Proofs and Programs about Open Terms

Francisco Ferreira Ruiz
School of Computer Science
McGill University, Montréal

December 2017

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctor of Philosophy

Copyright © 2017 by Francisco Ferreira Ruiz

Abstract

Formal deductive systems are very common in computer science. They are used to represent logics, programming languages, and security systems. Moreover, writing programs that manipulate them and that reason about them is important and common. Consider proof assistants, language interpreters, compilers and other software that process input described by formal systems. This thesis shows that contextual types can be used to build tools for convenient implementation and reasoning about deductive systems with binders. We discuss three aspects of this: the reconstruction of implicit parameters that makes writing proofs and programs with dependent types easier, the addition of contextual objects to an existing programming language that make implementing formal systems with binders easier, and finally, we explore the idea of embedding the logical framework LF using contextual types in fully dependently typed theory. These are three aspects of the same message: programming using the right abstraction allows us to solve deeper problems with less effort. In this sense we want: easier to write programs and proofs (with implicit parameters), languages that support binders (by embedding a syntactic framework using contextual types), and the power of the logical framework LF with the expressivity of dependent types.

Résumé

Les systèmes critiques comme les systèmes embarqués dans les avions requièrent un niveau de sécurité élevé qui peut seulement être obtenu par des systèmes formels. Les compilateurs certifiés ainsi que les assistants de preuve sont des programmes qui manipulent et qui raisonnent sur ces systèmes. Dans cette thèse, nous utilisons les types contextuels afin de construire de tels outils permettant notamment de raisonner sur des systèmes formels avec lieurs (binders). En particulier, nous abordons les trois points suivants : la reconstruction des paramètres implicites, ce qui simplifie l'écriture des preuves et programmes avec types dépendants ; l'ajout d'objets contextuels à un langage de programmation existant qui facilitent la mise en œuvre des systèmes formels avec des lieurs ; et l'intégration des types contextuels avec des types dépendants au-dessus du cadre logique (logical frameworks) LF. Ces trois facettes reflètent le message suivant : le choix d'une abstraction adéquate nous permet de construire des outils capables de résoudre des problèmes plus complexes plus facilement. Ceci se traduit par la construction de langages de programmation utilisant des paramètres implicites, supportant des lieurs (en intégrant un cadre syntaxique utilisant des types contextuels), et qui utilisent la puissance du cadre logique LF jointe à une théorie des types a la Martin-Löf.

A Juan, a la Nanana y a la Chongola.

Acknowledgements

Oh, the places I've seen! But getting to this point in the adventure would not have been possible or as much fun without the great people that supported me. First, I want to thank my doctoral advisor Brigitte Pientka whose passion and drive will always be examples to me. And Stefan Monnier for his confidence and always relevant guidance. I want to thank Prakash Panangaden for showing me the excitement that new ideas can provide. Laurie Hendren, a person that makes every interaction an insightful occasion, and whose laughter can cheer you up even through two doors!

Andrew Cave and I started at the same time, I will be for ever in debt to all the insights I get from each discussion. Andrew's discipline and empathy are a treasure to me. David Thibodeau, we shared so many discussions, pair programming, so many ideas. I could have not asked for a better friend to share these adventures with.

Annie Ying who offered support and understanding that you can only get from shared experiences. Pablo Duboue a dear friend that always has an interesting project or two and he was kind enough to let me participate in some.

The many friends that kept me sane: Stefan Knudsen, Shawn Otis, Rohan Jacob-Rao, Steven Thephsourinthone, Agata Murawska and François Thiré, Milena Scaccia, Caroline Berger, Yam Chhetri, Gayane Petrosyan, Aïna Linn Georges, Vincent Foley, Eric Lavoie, Larry Diehl, Alanna De Bortoli, Mathieu Boespflug, Tao Xue, . . .

I do not have space to enumerate my Argentine chosen family, your love and support keep me going every day of my life!

Getting to this point was difficult. But when the night was darkest I knew that Juan Buzzetti would be unconditionally there, no questions asked and that made all the difference in the world.

Finally, I want to thank my fantastic defense comittee: Prakash Panangaden, Clark Verbrugge, Stefan Monnier and Joerg Kienzle for their kind and insightful questions. Finally, special thanks to the external reviewer James Cheney.

I could not have done it without all of you. Thank you, so very much!

Contributions of the Author

- Chapter 3 is an extended version [Ferreira and Pientka, 2014] where I am first author and I developed the ideas under my co-author supervision.
- Chapter 4 is an extended version [Ferreira and Pientka, 2017] where I am also first author and where I developed the idea and prototype under my advisor's (also my co-author) supervision.
- Chapter 5 is a heavily extended version of [Ferreira et al., 2017], work that I did with David Thibodeau and Brigitte Pientka. The idea started as a consequence of the work on chapter 4 and we then developed together the implementation and the ideas that lead to the theory. This chapter spells out the theory for the first time.

Contents

Contents **vi**

List of Figures **viii**

1 Introduction **1**

2 Deductive Systems with Binders **9**

2.1 Introduction 9
2.2 The Representation of Binders 10
2.3 Higher Order Abstract Syntax 11
2.4 Logical Framework LF 14
2.5 Programming with Proofs: The Beluga System 20

3 Reconstruction of Implicit Parameters **28**

3.1 Introduction 28
3.2 Source Language 31
3.3 Target Language 46
3.4 Description of Elaboration 53
3.5 Soundness of Elaboration 73
3.6 Related Work 74
3.7 Conclusion 77

4 Contextual Types and Programming Languages **79**

4.1 Introduction 79
4.2 Main Ideas 82
4.3 Core-ML: A Small Functional Language 95
4.4 A Syntactic Framework 99
4.5 Core-ML with Contextual Types 106
4.6 Core-ML with GADTs 111

4.7 Deep Embedding of SF 115
4.8 From Contextual Types to GADTs 121
4.9 A Proof of Concept Implementation 128
4.10 Related Work 131
4.11 Conclusion 132

5 Contextual Types and Type Theory 134
5.1 Introduction 134
5.2 Example: Translating Boolean Types 135
5.3 Orca's Core Calculus 142
5.4 Definitions and Pattern Matching 155
5.5 The Prototype Implementation 159
5.6 Related Work 162
5.7 Conclusion 164

6 Conclusion 166
6.1 Future Work 166

A Proof of Soundness of Reconstruction 169

B Babybel's Translation Meta-theory 178

C Translating Booleans in Beluga 182

Bibliography 187

List of Figures

2.1 The STLC in LF 15
2.2 Well formed LF signatures and contexts 17
2.3 Well formed LF types and kinds 17
2.4 Typing LF 18
2.5 Type Approximations 19
2.6 Hereditary Substitutions on Types 20
2.7 Hereditary Substitutions on Terms 21
2.8 Type Uniqueness Proof 24
2.9 The Beluga Language 25
2.10 Contextual Objects 25
2.11 Typing of Contextual Objects 26
2.12 Reasoning Language 27

3.1 Well-formed Source Expressions 34
3.2 Well-formed Kinds and Types 35
3.3 Example: A Simply-typed λ-calculus 43
3.4 Target Language 47
3.5 Typing of Computational Expressions 48
3.6 Elaborating Declarations 54
3.7 Elaborating Kinds and Types in Declarations 61
3.8 Elaboration of Expressions (Checking Mode) 63
3.9 Elaboration of Expressions (Synthesizing Mode) 67
3.10 Branches and Patterns 69
3.11 Elaboration of Patterns and Pattern Spines 72

4.1 Adding Contextual Types to ML 81
4.2 Closure Converted Language 90
4.3 Core-ML Typing Rules 97

4.4 Core-ML Big-Step Operational Semantics 98
4.5 First-order Matching 100
4.6 Syntactic Framework Typing 102
4.7 Typing Rules for SF Patterns 107
4.8 Extended Operational Semantics 110
4.9 The Typing of Core-ML^{gadt} 113
4.10 Zip in Core-ML^{gadt} 115
4.11 Core-ML^{gadt} Big-Step Operational Semantics 116
4.12 Syntactic Framework Definition 118
4.13 Translating Types, Signatures, and Contexts 126
4.14 Translating Computational Expressions 127

5.1 The Typing of the Source Language 136
5.2 The Typing of the Target Language 137
5.3 Computation Substitution 146
5.4 Computation Substitution in Specifications 147
5.5 Computation Substitution in Contexts and Substitutions . . . 147
5.6 Specification Substitution 148
5.7 Typing for Computations 149
5.8 Typing Rules for Specifications (I) 151
5.9 Typing Rules for Specifications (II) 152
5.10 Equality Rules for Computations 153
5.11 Equality Rules for Specification Terms 153
5.12 Equality Rules for Specification Types and Kinds 154
5.13 Equality Rules for Contexts 154
5.14 Typing Rules for Patterns 157
5.15 Typing Equations 158
5.16 The Orca Pipeline 159
5.17 Boolean Translation After Box Inference 161

1 Introduction

Proofs are fundamental in mathematics and computer science. For the purpose of this thesis we will consider that a proof is an irrefutable argument in favour of a statement (i.e. a theorem). A formal proof is one that is presented as a step by step argument in some foundational system like ZFC set theory. The validity of a formal proof is reduced to mechanically checking all of its steps in regards to the rules established in its foundational system. On the one hand, formal proofs are, as one might expect, very verbose and long to validate. On the other hand, because they are a sequence of elementary steps in some theory, they are easy to verify by computers.

A formal proof is straightforward to check but laborious to construct. Therefore, proof assistant software was written as soon as computers became fast enough to validate them. The AUTOMATH [de Bruijn, 1983] system was a trailblazer in this field. The system was meant as a language to formalize mathematics and it introduced many ideas that we take for granted today. It pioneered, for example, ideas like using strongly typed λ-calculi as a formalism. Today there are many proof assistants that are in use. They are based on diverse formalisms, for example the Isabelle [Paulson, 1988] system that can use ZFC or Higher-Order Logic, or systems based in type theory like Coq The Coq Developement Team [2016], or Agda [Norell, 2007] and others.

The job of a proof assistant is not only to validate proofs but also to assist the user in describing the full formal proof. This is usually achieved by allowing one to omit parts that can be automatically reconstructed, invoking an automated decision procedure, or tactics (a mechanism for defining programs that compute proofs). Mechanizing a theorem is describing a theorem in a way that a proof assistant is able to generate and validate the complete formal proof.

The history of proof assistants or interactive theorem provers while not long is rich and eventful. A good reference is offered by Harrison et al. [2014]. As mentioned before, there are proof assistants based on several formalisms, among them several variations on type theory. Some (non-exhaustive) examples and their formalisms are:

- Church's simple theory of types [Church, 1940]
  - HOL4 [Slind and Norrish, 2008],
  - HOL light [Harrison, 2009] and
  - Isabelle/HOL [Nipkow et al., 2002]),
- Martin-Löf's type theory [Martin-Löf, 1984] (MLTT)
  - the Nuprl [Constable, 1986] proof assistant that is based on an extension of MLTT.
  - Coq [The Coq Developement Team, 2016] that uses an extension of the calculus of constructions [Coquand and Huet, 1988],
  - Agda [Norell, 2007] roughly based on UTT [Luo, 1994]
- the logical framework LF [Harper et al., 1993]
  - Twelf [Pfenning and Schürmann, 1999] that uses logic programming with LF definitions
  - Beluga [Pientka and Cave, 2015] that implements a reasoning language based on first order logic with induction on LF specifications.

In this thesis, we will mostly discuss systems that take advantage of the Curry-Howard correspondence [Howard, 1980], where propositions correspond to types and proofs correspond to terms (some proof assistants based on this idea are Agda, Coq and Beluga). In these systems, higher-order statements become dependent functions, that is, functions where

the resulting type depends on the value of the parameter. Theorems are then represented as function types and proofs by induction are represented by well-founded recursion and pattern matching for case analysis. Because type theory can be seen as a programming language [Nordström et al., 1990] these proof assistants can be used simultaneously as provers and as programming environments. This is a key advantage of constructive logics and type theory in particular, and it is instrumental to the subject matter of this thesis.

Proof assistants based on MLTT or extensions of the Calculus of Constructions like Coq are highly expressive and significant mathematical results have been formalized in them. For example the formalization of the odd order theorem by Gonthier et al. [2013] not only fully mechanizes the existing proof, but in order to do so it provides a library of algebraic definitions and theorems. With many logics, expressivity is not a problem; however, having the right setting and properties is crucial. For example, inductive types can be internally encoded in the pure calculus of constructions [Pfenning and Paulin-Mohring, 1990]. Nevertheless, the calculus of inductive constructions was designed to have inductive types as a first-class construct in order to have nicer computational behaviour and better usability [Paulin-Mohring, 1993]. A theme of this work is to take advantage of good representations (namely, using the logical framework LF and contextual types) to allow for a straightforward representation of theorems and programs about structures with binders and hypothetical judgments.

The kind of systems we want to represent are deductive systems. These are systems specified by axioms and deduction rules and they often include the notion of variables and binders. This class of systems is very large, and includes, for example: programming languages, logics, and their meta-theory together with their implementation. When dealing with deductive systems we distinguish two related activities: the first

one is implementing formal proofs about deductive systems (e.g.: a language is type safe, or a logic is normalizing). When talking about some object language (i.e.: the programming language or logic under study) we construct proofs about aspects of its meta-theory. We refer to this as *reasoning about the specification of the language* or working on the meta-theory of the specified deductive system. Second, we want to specify and perform computations over these systems, for example: compilers, evaluators and normalizers. We will refer to this as *computing with specification language* or simply writing programs that manipulate objects in the deductive systems.

Deductive systems that represent programming languages and logics typically have the idea of bound variables, that is some terms introduce new variables. When writing proofs or programs about such specifications one usually needs to recursively inspect the expression under a binder. Consider the first order formula: $\forall x.P(x)$ where one finds the predicate P that may contain free occurrences of x a variable that is bound outside by the universal quantifier. In this situation the sub-term might have free variables (i.e.: a variable bound outside of the term). Such terms are called open terms. Implementing and reasoning about open terms of systems with variables and binders is a common activity for computer scientists. Some typical examples of this are implementing new programming languages, reasoning about the meta-theory of languages and logics, and implementing proof assistants. Therefore, when working with open objects (terms with free variables) one finds oneself needing to represent variables, binders and substitutions.

The meaning of free variables cannot be ignored. One solution commonly used while reasoning both on paper and formally with a proof assistant, is to track free variables with a context that gives meaning to all the free variables in the term. Here we say the context binds the variables. This thesis explores a particular approach to this problem. Concretely,

we will explore the use of contextual types [Nanevski et al., 2008], that represent the type of an expression together with the context that gives meaning to all its free variables, to program and reason about higher-order abstract syntax (HOAS). Higher-order representations like HOAS, are used to represent binders reusing the function space, and function application to represent substitution, this approach is exemplified by the logical framework LF. While contextual types and the logical framework LF provide support for specifying formal systems, one can reason about these structures by implementing pattern matching and recursion over them. This has been done before, for example languages like Beluga [Pientka and Cave, 2015] and other languages like Delphin [Poswolsky and Schürmann, 2009]. Actually, Delphin does not have an idea of first class contexts but it does manipulate terms using HOAS.

This thesis shows that contextual types can be used to build tools for convenient implementation and reasoning about deductive systems with binders. We discuss the specification, reasoning and programming with open terms from the following points of view:

- How to reconstruct types in dependently typed systems. This is important to make the system accessible. Otherwise dependently typed programs are very verbose. We describe a formal algorithm for the sound reconstruction of implicit parameters (i.e.: parameters that the user does not write and the system infers) in systems with dependent types and a rich index domain like Beluga.

- How to integrate contextual types in an industrial strength functional programming language (in this case OCaml [Leroy et al., 2016b]) to allow for type safe programming with binders.

- How to extend Martin-Löf's type theory [Martin-Löf, 1984] (MLTT) as the reasoning/programming language for a system with contextual types.

These three points of view are intimately related. The objective of implicit parameter reconstruction is to simplify writing proofs about specifications and dependent pattern matching (which can be seen as case analysis). Integrating contextual object with existing programming languages allows for ease in writing programs that manipulate objects with variables and binders. To conclude, combining contextual objects and specifications with fully dependently typed language allows for implementing proofs about specifications, implementing programs over specifications, and particularly for implementing proofs about said programs.

Main Contributions

Reconstruction of Implicit Parameters

This chapter presents the design of a source language with index types and dependent pattern matching together with an elaboration phase that reconstructs omitted arguments. We differentiate between implicit arguments; those that the user does not write, and explicit arguments, which must be provided by the user. This language and its elaboration describe the corresponding reconstruction of the computational language of the Beluga system.

The elaboration is type directed and it infers omitted arguments to produce a closed well-typed program. Notably, we describe the reconstruction of pattern matching expressions. Specifically, using the design of pattern matching inspired by Beluga that provides nested dependent pattern matching without type annotations. This is an important distinction from other systems that either do not provide nested dependent matching statements (such as Agda [Norell, 2007] or Idris [Brady, 2013]) or require annotations for the return type (as the Coq [The Coq Developement Team, 2016] proof assistant).

Finally, we prove that this is sound, that is, that the successful elaboration of a term implies that it is well-typed in the target language. Part of this work was published in Ferreira and Pientka [2014].

Contextual Types and Programming Languages

Implementing contextual types in a current and existing programming language brings some of the power of Beluga and HOAS or λ-tree definitions [Miller and Palamidessi, 1999] to existing (simply-typed) functional programming languages. Using contextual types and a syntactic framework (SF) based on modal S4 [Nanevski et al., 2008, Davies and Pfenning, 2001], programmers can manipulate open objects by pattern matching with a type system that guarantees the binders do not escape their scopes. We show that this language extension can be translated into a language that supports Generalized Abstract Data Types (i.e.: GADTs) using a deep embedding of SF.

The other contribution is Babybel, a prototype implementation of these ideas. Babybel is implemented as a syntax extension of the OCaml language. It takes advantage of OCaml's type system to ensure that the translation is type preserving and allows the users to take advantage of GADTs to represent some inductive predicates over syntax, like context relations. Finally, part of this work was published in Ferreira and Pientka [2017].

Contextual Types and Type Theory

The main contribution of this chapter is a calculus that integrates the logical framework LF for specifications with Martin-Löf's type theory as a reasoning/computation language. This is achieved using contextual types and it can be seen as an extension of the technique for adding contextual types to programming languages. Morally it presents an extended ver-

sion of the Beluga language with fully dependent typing. MLTT does not have a phase separation between type-checking and evaluation allowing for the interleaving of computation and specification. Additionally, because MLTT allows for reasoning about functions, the theory permits writing computations and proving properties about the computations.

A further contribution, is the Orca prototype that implements these ideas. The design of the Orca language also provides an interesting type directed syntax reconstruction to be able to disambiguate between computational terms and specification terms which provides a nicer user experience. Some of these ideas were presented at Ferreira et al. [2017].

2 Deductive Systems with Binders

2.1 Introduction

Deductive systems presented using axioms and deduction rules are designed to formally describe logics, programming languages, and proof assistants. Therefore they are widely used in the study of programming languages and their properties. Examples abound: from the description of modal logic systems in Pfenning and Davies [2001], to the presentation of the core calculus of a reactive programming language [Cave et al., 2014], to the specification of the addition of dependent types to Haskell, a real world general purpose language [Weirich et al., 2017]

Most of the formal systems that we discuss in this thesis require variables and binders (the place where new variables are introduced). This a delicate aspect of the presentation of a formal system, where one must be aware of issues like variables not escaping their scopes, comparing terms up-to the renaming of bound variables (i.e.: α-equivalent terms), and that the substitution operation shall not capture free variables. Consider for example, the untyped λ-calculus, its syntax is:

$$
\begin{array}{llcll}
\text{Terms} & M, N & ::= & x & \text{variables} \\
 & & \mid & \lambda\, x.M & \text{function abstraction} \\
 & & \mid & M\,N & \text{application}
\end{array}
$$

Note that x is a name from a set that contains countably many distinct variable names. An important operation in the λ-calculus is that of substitution, that allows the instantiation of a variable with a term. We write $[M/x]\ N$ to say replace every occurrence of variable x for term M in term N. The usual definition is done inductively on the structure of N in the following way:

$$
\begin{array}{llcll}
[M/x] & y & = & M & \text{when } x = y \\
[M/x] & y & = & y & \text{when } x \neq y \\
[M/x] & \lambda z.N & = & \lambda z.[M/x]\ N & \text{with } z \text{ not free in } M \\
[M/x] & N\,N' & = & ([M/x]\ N)\,([M/x]\ N') &
\end{array}
$$

The definition is straightforward except that at first sight, the reader might think that it is not a total operation. After all, in the abstraction case, what is one supposed to do if z does appear free in M? (i.e.: a variable is free if its binder is not part of the term). The answer is that the substitution needs to be applied to a term where z has been renamed (i.e.: the substitution continues with an α-equivalent version of the term). This issue is the central idea in capture avoiding substitution. While this is usually left implicit in a description, it remains an issue in implementations and mechanized formal presentations.

2.2 The Representation of Binders

Representing variables as string and adding side conditions when necessary for substitution works well enough for paper or black-board presentations. However, names as strings are less common in computer implementations. For example, N. G. de Bruijn, when designing Automath [de Bruijn, 1991] (one of the first proof assistants) proposed a nameless representation [de Bruijn, 1972] where a variable is the distance, in number of binders, to the place where the variable is bound. On one hand this eliminates issues with name capture, but on the other hand, humans usually find these terms very difficult to understand. This idea was later refined by Altenkirch [1993] as well-scoped de Bruijn indices where they use dependent types to enforce scoping invariants.

An alternative approach is using nominal logic [Gabbay and Pitts, 1999] or categorical approaches such as [Fiore and Hur, 2008], to give a

precise mathematical definition the ideas of α-equality and name capture.

A final approach, and the one that is discussed in this thesis the most, is using different versions of the λ-calculus as representation frameworks and reusing their function space to introduce binders. This allows for a simple implementation of substitution as one simply reuses the notion of substitution of the underlying calculus. This was first used by Church in [Church, 1940], but the logical framework LF [Harper et al., 1993] takes full advantage of the idea. LF proposes a dependently typed typed λ-calculus as a representation logic that is able to encode syntax together with judgments. This technique is usually known as Higher-Order Abstract Syntax (HOAS) because of the reutilization of the the function space to implement binders. A related technique with similar presentation is the use of λ-trees [Miller and Palamidessi, 1999].

2.3 Higher Order Abstract Syntax

The idea of representing binders using the function space of λ-calculi to represent binders and dependent types to represent judgments is a key insight provided by LF and allows for the mechanization of deductive systems in a direct and high-level way. The use of the function space of LF frees the user from thinking about the representation of binders, the implementation of substitution and even proving some substitution lemmas as all this infrastructure is inherited from the framework itself.

In regular programming languages (that provide higher-order functions) it is possible to represent binders using the function space of the language, like in this OCaml example:

```
type exp =
  | Lam of (exp → exp) (* abstractions *)
  | App of exp * exp (* applications *)

let omega = Lam (fun x → App (x, x)) (* little omega *)

let rec eval : exp → exp = function
  | Lam f → Lam f (* abstractions are values *)
  | App (e1, e2) →
     begin match eval e1 with
     | Lam f → eval (f e2) (* function application is
    substitution *)
     | stuck → App (e1, e2)
     end
```

This short example is enough to show two of the crucial aspects of HOAS. The first appears in the declaration of the type `exp`, where the constructor for abstractions uses a function to represent the body of the λ-expression that contains a variable. For that reason it is introduced as a higher-order function. The second aspect is that in this setting the function application represents substitution, and we can notice this when reducing applications. To perform the substitution it suffices to apply the function because binders are implemented by functions.

This short example is also enough to show two important problems: the first is that the OCaml function space is too rich and there are many functions that do not represent terms in the λ-calculus. For example:

```
let exotic : exp =
  Lam (fun x → match x with App (_, e) → e | e → e)
```

This exotic term pattern matches on the shape of its argument (dropping all the terms in function position) while in the λ-calculus variables can

only be substituted for a term and cannot analyze the shape of any term. Thus, this representation, while it might be convenient, is not adequate and it is not a replacement for LF. The right framework should provide an adequate representation (i.e. a function space that is weak enough so that no exotic terms exist) and should allow for the intensional inspection of the represented functions. LF is designed with these features in mind (together with the ability to represent judgments and completely represent deductive systems).

The other problem is that because functions operate as black boxes, it is not easy to operate on open terms as the only operation on functions is application, and thus the only way to get to the body of the abstraction is to perform a substitution. The issue is that the function space in OCaml behaves like an extensional function (i.e.: one can only observe the result of a function application). When operating with open terms one would need an intensional function space that gives access to the structure of the implementation of the function [Pfenning, 2001].

One possible solution is to separate the language that describes computation from the language that is used for specifications. This approach was started by Schürmann et al. [2001] when they proposed a computation language with primitive recursion and a simply typed language for specifications. This idea was pushed forward in Delphin [Poswolsky and Schürmann, 2009] when they added support for dependent types and the logical framework LF. Finally, the Beluga [Pientka, 2008] system uses contextual types together with logical framework LF. Contextual types describe potentially open terms (terms with free variables) together with their contexts (that provide a binding to all the free variables of a term). This allows for proofs and programs about open objects and that inspect terms under binders by keeping track of their contexts. This thesis explores some extensions and implementation concerns for these ideas.

2.4 Logical Framework LF

The logical framework LF provides the means to represent the syntax and judgments of deductive systems through the use of a dependently typed calculus related to Martin-Löf's system of arities [Nordström et al., 1990].

In LF, judgment and syntactic categories are represented by types, and constructors represent respectively the inference rules and terms of the object language. For example Figure 2.1 shows how to represent the simply typed λ-calculus (STLC) using the concrete syntax of the Beluga system. There are two types, one for each syntactic category (i.e.: `tp` for types and `tm` for term) and the typing judgment is represented by a dependent type (`oft`). In the representation of *lambda* expressions in terms, notice how binding is represented by a function (HOAS). The most interesting case is the `oft` judgment that relates a term to its type. The judgment contains three constructors that correspond to the typing rules for applications, abstractions and the constant respectively. Because of the use of HOAS, there is no need for a rule for variables, as the typing assumptions for variables are added to the context by the rule `t-lam`.

The logical framework LF is a dependently typed theory related to the λP vertex of the λ-cube [Barendregt, 1992]. The particular presentation we use is often called Canonical LF because it only allows for normal (i.e.: canonical) forms. Regular substitution might introduce non-normal forms, so substitution is done in an "hereditary" way. Hereditary substitutions continue reducing to avoid the introduction of non-normal forms. This technique was introduced in the context of Concurrent LF by [Watkins et al., 2002] and [Cervesato et al., 2002]. In particular this presentation is based on work by Harper and Licata [2007].

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

LF tm : type =
| app : tm → tm → tm
| lam : (tm → tm) → tm
| c : tm
;

LF oft : tm → tp → type =
| t-app : oft M (arr S T) → oft N S → oft (app M N) T
| t-lam : ({x:tm} oft x S → oft (M x) T) →
          oft (lam M) (arr S T)
| t-c : oft c b
;
```

Figure 2.1: The simply typed λ-calculus in LF

The syntax is as follows:

Kinds	K	::=	$\texttt{type} \mid \Pi x{:}A.K$
Base Types	P	::=	$\mathbf{a} \mid P\,M$
Types	A, B	::=	$P \mid \Pi x{:}A.B$
Normal Terms	M	::=	$\lambda x.M \mid R$
Neutral Terms	R	::=	$\mathbf{c} \mid x \mid R\,M$
Contexts	Ψ	::=	$\cdot \mid \Psi, x{:}A$
Signature	Σ	::=	$\cdot \mid \Sigma, \mathbf{a}{:}K \mid \Sigma, \mathbf{c}{:}A$

Kinds classify the types and are either `type` for non-dependent kinds (used to represent our syntactic categories) and $\Pi x{:}A.K$ for judgments. Base types are either atomic types $\mathbf{a}$ or type constructors applied to terms $P\,M$. Types classify terms and are: base types P, or $\Pi x{:}A.B$ a dependent function space, when x does not appear in B we write $A \to B$ to indicate the simply typed function space. Terms are split between normal and

neutral terms, with the objective of preventing β reducible terms. So normal terms contain function abstractions $\lambda x.M$ and neutral terms. And neutral terms are constructors $\mathbf{c}$, variables x bound in abstractions or dependent functions, and applications of neutral terms to normal terms $R\,M$. Finally, contexts Ψ store typing assumptions for bound variables, and the signature Σ contains the user definitions (that represent the object language as in Figure 2.1).

Well typed LF terms are defined by these judgments:

- Well formed signatures, contexts and kinds and types are given by:
  - $\boxed{\vdash \Sigma\ \mathtt{sig}}$: Σ is a valid signature.
  - $\boxed{\vdash_\Sigma \Psi\ \mathtt{ctx}}$: Ψ is a well formed context in signature Σ.
  - $\boxed{\Psi \vdash_\Sigma K\ \mathtt{kind}}$: Kind K is well formed in context Ψ.
  - $\boxed{\Psi \vdash_\Sigma A\ \mathtt{type}}$: Type A is well formed in context Ψ.
- Well kinded base types and well typed terms are given by:
  - $\boxed{\Psi \vdash P \Rightarrow K}$: Base type P synthesizes kind K in Ψ.
  - $\boxed{\Psi \vdash M \Leftarrow A}$: Normal term M checks against type A in Ψ.
  - $\boxed{\Psi \vdash R \Rightarrow A}$: Neutral term P synthesizes type A in Ψ.

We assume signatures are well formed, and we omit them in the rules because they do not change during typing. Similarly, for contexts, we remark that the rules only extend contexts with well formed assumptions, so we assume that one starts with a well-formed context (empty or otherwise) and the rules preserve that property. We refer to Harper et al. [1993] and Harper and Licata [2007] for a more detailed discussion of these issues.

Figures 2.2, 2.3, and 2.4 show the deduction rules for each judgment. They are needed to establish when a term is well formed and well typed. Logics and deductive systems are represented by the canonical forms (i.e.:

$\boxed{\vdash \Sigma\ \mathtt{sig}}$: Σ is a valid signature.

$$\frac{}{\vdash \cdot\ \mathtt{sig}}\ \mathtt{s\text{-}empty} \qquad \frac{\vdash \Sigma\ \mathtt{sig} \quad \vdash_\Sigma A\ \mathtt{type}}{\vdash \Sigma, \mathbf{c}{:}A\ \mathtt{sig}}\ \mathtt{s\text{-}type}$$

$$\frac{\vdash \Sigma\ \mathtt{sig} \quad \cdot \vdash_\Sigma K\ \mathtt{kind}}{\vdash \Sigma, \mathbf{a}{:}K\ \mathtt{sig}}\ \mathtt{s\text{-}con}$$

$\boxed{\vdash_\Sigma \Psi\ \mathtt{ctx}}$: Ψ is a well formed context in signature Σ.

$$\frac{}{\vdash_\Sigma \cdot\ \mathtt{ctx}}\ \mathtt{c\text{-}empty} \qquad \frac{\vdash_\Sigma \Psi\ \mathtt{ctx} \quad \cdot \vdash A\ \mathtt{type}}{\vdash_\Sigma \Psi, x{:}A\ \mathtt{ctx}}\ \mathtt{c\text{-}hyp}$$

Figure 2.2: Well formed LF signatures and contexts

$\boxed{\Psi \vdash_\Sigma K\ \mathtt{kind}}$: Kind K is well formed in context Ψ.

$$\frac{}{\Psi \vdash_\Sigma \mathtt{type}\ \mathtt{kind}}\ \mathtt{k\text{-}type} \qquad \frac{\Psi \vdash_\Sigma A\ \mathtt{type} \quad \Psi, x{:}A \vdash_\Sigma K\ \mathtt{kind}}{\Psi \vdash_\Sigma \Pi x{:}A.K\ \mathtt{kind}}\ \mathtt{k\text{-}pi}$$

$\boxed{\Psi \vdash_\Sigma A\ \mathtt{type}}$: Type A is well formed in context Ψ.

$$\frac{\Psi \vdash P \Rightarrow \mathtt{type}}{\Psi \vdash_\Sigma P\ \mathtt{type}}\ \mathtt{t\text{-}base} \qquad \frac{\Psi \vdash_\Sigma A\ \mathtt{type} \quad \Psi, x{:}A \vdash_\Sigma B\ \mathtt{type}}{\Psi \vdash_\Sigma \Pi x{:}A.B\ \mathtt{type}}\ \mathtt{t\text{-}pi}$$

Figure 2.3: Well formed LF types and kinds

$\boxed{\Psi \vdash P \Rightarrow K}$: Base type P synthesizes kind K in Ψ.

$$\frac{a : K \in \Sigma}{\Psi \vdash \mathbf{a} \Rightarrow K}\ \texttt{p-con} \qquad \frac{\Psi \vdash P \Rightarrow \Pi x : A.K \quad \Psi \vdash M \Leftarrow A}{\Psi \vdash P\, M \Rightarrow [M/x]^k_{(A)^-} K}\ \texttt{p-app}$$

$\boxed{\Psi \vdash M \Leftarrow A}$: Normal term M checks against type A in Ψ.

$$\frac{\Psi, x : A \vdash M \Leftarrow B}{\Psi \vdash \lambda x.M \Leftarrow \Pi x : A.B}\ \texttt{t-lam} \qquad \frac{\Psi \vdash R \Rightarrow A}{\Psi \vdash R \Leftarrow A}\ \texttt{t-neu}$$

$\boxed{\Psi \vdash R \Rightarrow A}$: Neutral term P synthesizes type A in Ψ.

$$\frac{\mathbf{c} : A \in \Sigma}{\Psi \vdash \mathbf{c} \Rightarrow A}\ \texttt{t-con} \qquad \frac{x : A \in \Psi}{\Psi \vdash x \Rightarrow A}\ \texttt{t-var}$$

$$\frac{\Psi \vdash R \Rightarrow \Pi x : A.B \quad \Psi \vdash M \Leftarrow A}{\Psi \vdash R\, M \Rightarrow [M/x]^a_{(A)^-} B}\ \texttt{t-app}$$

Figure 2.4: Typing LF

normal forms) of LF terms. There are at least two approaches to compare the normal forms:

- Defining a type directed equality to compare terms up-to β- reduction and η-expansion[Harper and Pfenning, 2005].
- Defining Canonical LF where all terms are in canonical form, and using hereditary substitutions [Watkins et al., 2002] to preserve this invariant. Harper and Licata [2007] explain this approach and show a proof of adequacy for these encodings. As we already mentioned, this is the approach we present here.

$$
\begin{array}{rcl}
(\Pi x : A.B)^- & = & (A)^- \rightarrow (B)^- \\
(P\,M)^- & = & (P)^- \\
(\mathbf{a})^- & = & \mathbf{a}
\end{array}
$$

Figure 2.5: Type Approximations

We define hereditary substitutions for kinds, types and terms. This requires defining six operations:

- $\boxed{[M/x]^k_\alpha\, K = K'}$: Hereditary substitution in kinds.
- $\boxed{[M/x]^a_\alpha\, A = A'}$: Hereditary substitution in types.
- $\boxed{[M/x]^p_\alpha\, P = P'}$: Hereditary substitution in base types.
- $\boxed{[M/x]^m_\alpha\, M' = M''}$: Hereditary substitution in normal terms.
- $\boxed{[M/x]^r_\alpha\, R = M' : \alpha'}$: Hereditary substitution in a neutral term producing a normal term.
- $\boxed{[M/x]^r_\alpha\, R = R'}$: Hereditary substitution in a neutral term producing a neutral term.

The superscript in the substitution indicates the syntactic domain (and it can often be omitted) where it applies, and the subscript is a type approximation of the resulting types that guides the process and the termination argument. The type approximation is defined in Figure 2.5. The computation rules for the hereditary substitution in types are presented in Figure 2.6 and Figure 2.7 presents the rules for terms. The definition is similar to regular substitution, but it critically differs when applying a substitution to a neutral term produces a normal term. In this situation, if the neutral term is an application then the substitution needs to continue until a normal form is reached (otherwise an unrepresentable β-reduction

$\boxed{[M/x]^k_\alpha\, K = K'}$: Hereditary substitution in kinds.

$$
\begin{array}{lll}
[M/x]^k_\alpha \;\; \texttt{type} & = & \texttt{type} \\
[M/x]^k_\alpha \;\; \Pi\, y : A.K & = & \Pi\, y : ([M/x]^a_\alpha A).([M/x]^k_\alpha K) \\
 & & \text{with } x \neq y \text{ and } y \notin FV(M)
\end{array}
$$

$\boxed{[M/x]^a_\alpha\, A = A'}$: Hereditary substitution in types.

$$
\begin{array}{lll}
[M/x]^a_\alpha \;\; P & = & [M/x]^p_\alpha P \\
[M/x]^a_\alpha \;\; \Pi\, y : A.B & = & \Pi\, y : ([M/x]^a_\alpha A).([M/x]^a_\alpha B) \\
 & & \text{with } x \neq y \text{ and } y \notin FV(M)
\end{array}
$$

$\boxed{[M/x]^p_\alpha\, P = P'}$: Hereditary substitution in base types.

$$
\begin{array}{lll}
[M/x]^p_\alpha \;\; \mathbf{a} & = & \mathbf{a} \\
[M/x]^p_\alpha \;\; P\, M' & = & ([M/x]^p_\alpha P)\, ([M/x]^m_\alpha M')
\end{array}
$$

Figure 2.6: Hereditary Substitutions on Types

would be created), notice how the type approximations get progressively smaller.

2.5 Programming with Proofs: The Beluga System

The Beluga [Pientka and Dunfield, 2010b] system[1] is a proof assistant/programming language that manipulates LF terms and inductive data to implement proofs the meta-theory of formal systems specified in LF, and to implement programs that manipulate these terms, shining examples are translators and compilers, for example a type safe closure conversion

[1] Available at: `https://github.com/Beluga-lang/Beluga`

$\boxed{[M/x]^m_\alpha\, M' = M''}$: Hereditary substitution in normal terms.

$$
\begin{array}{lcl}
[M/x]^m_\alpha \quad \lambda\, y.M' & = & \lambda\, y.([M/x]^m_\alpha M') \\
 & & \text{with } x \neq y \text{ and } y \notin FV(M) \\
[M/x]^m_\alpha \quad R & = & M' \text{ where } [M/x]^r_\alpha R = M' : \alpha'
\end{array}
$$

$\boxed{[M/x]^r_\alpha\, R = M' : \alpha'}$: H. subst. in a neutral producing a normal term.

$$
\begin{array}{lcl}
[M/x]^r_\alpha \quad x & = & M : \alpha \\
[M/x]^r_\alpha \quad R\, M_1 & = & [([M/x]^m_\alpha M_1)/y]^m_{\alpha_1} M_2 : \alpha_2 \\
 & & \text{when } [M/x]^r_\alpha R = \lambda\, y.M_2 : \alpha_1 \to \alpha_2
\end{array}
$$

$\boxed{[M/x]^r_\alpha\, R = R'}$: H. subst. in a neutral producing a neutral term.

$$
\begin{array}{lcl}
[M/x]^r_\alpha \quad y & = & y \quad \text{with } x \neq y \\
[M/x]^r_\alpha \quad R\, M' & = & R'\ ([M/x]^m_\alpha M') \\
 & & \text{when } [M/x]^r_\alpha R = R'
\end{array}
$$

Figure 2.7: Hereditary Substitutions on Terms

and hoisting implementation [Belanger et al., 2013], or normalization by evaluation for the simply typed λ-calculus [Cave and Pientka, 2012].

Beluga is a dependently typed programming language where programs directly correspond to first-order logic proofs over a specific domain. More specifically, a proof by cases (and more generally by induction) corresponds to a (total) functional program with dependent pattern matching. We hence separate the language of programs from the language of our specific domain about which we reason. The language is similar to indexed type systems (see [Zenger, 1997, Xi and Pfenning, 1999]); however, unlike these aforementioned systems, Beluga's index domain is much richer (contextual LF terms) and it allows pattern matching on index objects, i.e. we support case-analysis on objects in our domain.

Figure 2.8 shows the type uniqueness theorem for the calculus from

Figure 2.1. The property we want to establish is that if a term M has a derivation for type S and one for type T then S and T are necessarily equal. To be able to state the theorem first we define `eq-tp` to formally say when two types are equal. And then we define a context schema (that is the classifier or "type" of contexts) where we say that each entry contains a variable together with its type derivation. The actual proof is a total recursive function (following the Curry-Howard correspondence) where the statement of the theorem is the type of the function, and the proof is its implementation. In this case, the type is:

```
(g : ctx) [g ⊢ oft M S[]] → [g ⊢ oft M T[]] →
  [⊢eq-tp S T]
```

That can be read as: give a context `g` where each variable is stored together with its type derivation, and a proof that term `M` has type `S` and `T` (where the square brackets around the types `T` and `S` mean that they are closed objects and do not depend on the context) the function shows that both types are equal. We see in Figure 2.8 that to implement the proof, the function uses case analysis and well-founded recursion to implement induction, recursive function calls on smaller arguments to appeal to the induction hypothesis and let expressions, or irrefutable pattern matching to implement inversion. The application case follows by inversion and the invoking the induction hypothesis, the case for the constructor (i.e.: `t-c`) is the base case and it just needs to perform inversion on the other derivation. The more interesting cases are `t-lam` for functions and the variable case (`#p.u`). The case for λ derivation uses an inductive call done in an extended context where we have to be careful extend the context appropriately, which is what we do in the substitution we apply to the variables `D` and `E`. The resulting recursive call is:

```
unique [g, b:block x:tm, u:oft x _ ⊢ D[..,b.x,b.u]]
       [g, b                       ⊢ E[..,b.x,b.u]]
```

Finally, the variable case also follows by inversion using the information stored in the context.

The Beluga language reasons about *specifications* in the logical framework LF, by embedding them using *contextual objects* in a functional programming language. The idea, illustrated in Figure 2.9, is that proofs are represented by programs, as in the proofs as programs methodology [Howard, 1980], and the specifications are embedded using contextual objects [Nanevski et al., 2008] and contextual types that combine the type of LF objects with the context they are valid in.

One of the crucial features of Beluga is that it handles open objects (i.e.: objects with free variables bound in a context). The technical device to achieve this is contextual objects and types and it is based on contextual modal type theory [Nanevski et al., 2008]. A contextual object is a term together with the context that describes all the free variables of the term. In a sense, one could say that there are no free variables, just variables bound in some context. As a consequence, LF terms and types are combined with a context (in the case of terms, the context just describes the names of the assumptions that appear on the type, this is meant to support α-conversions and it is denoted with the hatted contexts as in $\hat{\Psi}$).

Figure 2.10 shows the syntax of the contextual objects, together with the addition of meta-variables to be able to describe incomplete LF terms. We use X to represent meta-variables, and u and $\#p$ when talking about a specific one. Moreover, we use $u\,[\sigma]$ where the substitution expresses how u relates to its context. The case for $\#p$ is similar, but we use the sharp sign to represent parameter variables that are a kind of meta-variable that can only be instantiated by variables from the context Ψ. We write contextual terms together with a context where types have been erased and only names remain. The typing rule for contextual term indicated

```
LF eq-tp: tp → tp → type =
% two types are equal only if they are the same
| refl: eq-tp T T;

% the context stores together the variable x
% and its type derivation
schema ctx = some [t:tp] block (x:tm, u:oft x t);

% Thm: If a term M has types S and also T, then S = T
rec unique : (g : ctx) [g ⊢ oft M S[]] → [g ⊢ oft M T[]] →
    [⊢ eq-tp S T] =
/ total d (unique _ _ _ _ d _) / % requires well founded
    recursive calls and totality
fn d ⇒ fn e ⇒ case d of

| [g ⊢ t-app D1 D2] ⇒
  % by inversion on the other derivation
  let [g ⊢ t-app E1 E2] = e in
  let [⊢ refl] = unique [g ⊢ D1] [g ⊢ E1] in % by i.h.
  [⊢ refl]

| [g ⊢ t-c] ⇒
  % by inversion
  let [g ⊢ t-c] = e in
  [⊢ refl]

| [g ⊢ t-lam λx.λu. D] ⇒
  let [g ⊢ t-lam λx.λu.E] = e in % by inversion
  let [⊢ refl] =
      % by i.h. in the extended ctx
      unique [g, b:block x:tm, u:oft x _ ⊢ D[..,b.x,b.u]]
             [g, b ⊢  E[..,b.x,b.u]]
  in
  [⊢ refl]

| [g ⊢ #p.u] ⇒ % a variable in the context
  let [g ⊢ #p.u] = e in % the type derivation in the context
  [⊢ refl];
```

Figure 2.8: Type Uniqueness Proof

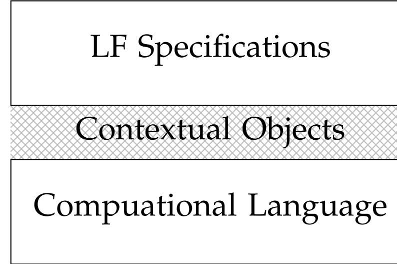

Figure 2.9: The Beluga Language

Contextual Objects	C	::=	$\hat{\Psi} \vdash M \mid \Psi$
Contextual Types	U	::=	$\Psi \vdash A \mid \Psi \vdash \#A \mid G$
Contextual Schemas	G	::=	$\exists\overrightarrow{(x : A)}\ .\ B \mid G + \exists\overrightarrow{(x : A)}\ .\ B$
Substitutions	σ	::=	$\cdot \mid \mathtt{id} \mid \sigma, M$
Terms w/Meta vars.	M	::=	$\cdots \mid u\,[\sigma] \mid \#p\,[\sigma]$
Meta Substitutions	θ	::=	$\cdot \mid \theta, C/X$
Meta Context	Δ	::=	$\cdot \mid \Delta, X : U$

Figure 2.10: Contextual Objects

that when a term: $\hat{\Psi} \vdash M$ is matched against a contextual type: $\Psi \vdash A$, typing continues with the LF typing judgment using the adequate contexts, namely: $\Delta; \Psi \vdash M \Leftarrow A$.

Finally, contexts are also first-class objects, and they are classified by contextual schemas[2]. Context schemas are formed by elements $\exists\overrightarrow{(x : A)}\ .\ B$ where we say that assumptions are of type B. If B is a type family its indices are provided by the existential. When schemas describe contexts that may contain assumptions of different type, they appear as sum types. Similarly one could, as the implementation has, add products to express contexts that grow by blocks of assumptions. Figure 2.11 shows the typing for contextual terms and contexts.

With the machinery to represent LF objects together with their con-

[2]Contextual schemas play the role of types for contexts.

$\boxed{\Delta \vdash C : U}$: C is of type U in Δ

$$\frac{\Delta; \Psi \vdash M \Leftarrow A}{\Delta \vdash (\hat{\Psi} \vdash M) : (\Psi \vdash A)} \qquad \frac{}{\Delta \vdash \cdot : G}$$

$$\frac{\Delta \vdash \Psi : G \quad \text{exists } \overrightarrow{(x : B')} \;.\; B \in G \text{ and } \Delta; \Psi \vdash \overrightarrow{M} : \overrightarrow{B} \text{ s.t. } A = B\,[\overrightarrow{M}/\overrightarrow{x}\,]}{\Delta \vdash \Psi, x : A : G}$$

Figure 2.11: Typing of Contextual Objects

text we can describe now the reasoning language. This language can be seen as as a term assignment of first-order logic with inductive types. Its syntax is presented in Figure 2.12. It supports inductive type families [Dybjer, 1994] where the indices come from the index domain and two kinds of functions, dependent function space from contextual types to computational types, an arrow type for computational functions, a box type to represent contextual-LF objects and finally, the type for fully applied type families from the context.

The terms contain abstractions and applications for each function space and notable pattern matching through the case expressions to analyze inductive data and contextual objects.

Finally, the signature stores the definition of types and constructors and recursive functions. Beluga supports general recursion, but it implements a termination checker [Pientka and Abel, 2015] to ensure that functions and therefore proofs are well founded.

Kinds	K	::=	$\Pi X{:}U.K \mid \mathtt{ctype}$
Types	T	::=	$\Pi X{:}U.T \mid T_1 \rightarrow T_2 \mid [U] \mid \mathbf{a}\,\vec{C}$
Expressions	E	::=	$\mathtt{fn}\,x \Rightarrow E \mid \mathtt{mlam}\,X \Rightarrow E \mid E_1\,E_2$
			$\mid E_1\,[C] \mid [C] \mid \mathtt{case}\,E\,\mathtt{of}\,\vec{B} \mid E : T \mid x \mid \mathbf{c}$
Branches	$\vec{B}$	::=	$B \mid (B \mid \vec{B})$
Branch	B	::=	$\Pi\Delta;\Gamma.Pat : \theta \mapsto E$
Pattern	Pat	::=	$x \mid [C] \mid \mathbf{c}\,\overrightarrow{Pat}$
Signature	Σ	::=	$\mathbf{c} : T \mid \mathbf{a} : K \mid \mathtt{rec}\,f : T = E$
Context	Γ	::=	$\cdot \mid \Gamma, x : T$

Figure 2.12: Reasoning Language

3 Reconstruction of Implicit Parameters

3.1 Introduction

Dependently typed programming languages allow programmers to express a rich set of properties and statically verify them via type checking. To make programming with dependent types practical, these systems provide a source language where programmers can omit (implicit) arguments which can be reasonably easy inferred and elaborate the source language into a well-understood core language, an idea going back to Pollack [1990]. However, this elaboration is rarely specified formally for dependently typed languages which support recursion and pattern matching. For Agda, a full dependently typed programming language based on Martin Löf type theory, Norell [2007, Chapter 3] describes a bi-directional type inference algorithm, but does not treat the elaboration of recursion and pattern matching. For the fully dependently typed language Idris, Brady [2013] describes the elaboration between source and target, but no theoretical properties such as soundness are established. A notable exception is Asperti et al. [2012] that describes a sound bi-directional elaboration algorithm for the Calculus of (Co)Inductive Constructions (CCIC) implemented in Matita.

In this chapter, we concentrate on a computational language with index types. Specifically, our source language is inspired by the Beluga language [Pientka and Cave, 2015, Pientka, 2008, Pientka and Dunfield, 2010b, Cave and Pientka, 2012] (presented in Section 2.5 where we specify formal systems in the logical framework LF [Harper et al., 1993] (our index language) and write proofs about LF objects as total recursive functions using pattern matching.

More generally, our language may be viewed as a smooth extension of simply typed languages, like Haskell or OCaml to nested dependent pattern matching. Moreover, taking advantage of the separation between types and terms, it is easy to allow impure programs, for example to allow non-termination, partial functions, and polymorphism. All this while reaping some of the benefits of dependent types.

Viewing our language through the Curry-Howard correspondence, it closely corresponds to a proof term assignment for first-order logic over a specific domain. Our dependent pattern matching construct corresponds to case-analysis on predicates in first-order logic. Pattern matching on index objects corresponds to case-analysis on an object from the domain. Writing total functions in this language corresponds to inductive proofs in first-order logic over a given domain. This domain is kept somewhat abstract, but in the case of Beluga it is the logical framework LF. The reconstruction algorithm for LF is presented in [Pientka, 2013]. However, the logical framework LF is not a programming language and it does not contain features like pattern matching that we consider in this work.

The main contribution of this chapter is the design of a source language for dependently typed programs where we omit implicit arguments together with a sound bi-directional elaboration algorithm from the source language to a fully explicit core language. This language supports dependent pattern matching without requiring type invariant annotations, and dependently-typed case expressions can be nested as in simply-typed pattern matching. Throughout our development, we will keep the index language abstract and state abstractly our requirements such as decidability of equality and typing. There are many interesting choices of index languages. For example choosing arithmetic would lead to a DML [Xi, 2007] style language ; choosing an authorization logic would let us manipulate authorization certificates (similar to *Aura* [Jia et al., 2008]); choosing LF style languages (like Contextual LF [Nanevski

et al., 2008]) we obtain Beluga; choosing substructural variant of it like CLF [Watkins et al., 2002] we are in principle able to manipulate and work with substructural specifications.

A central question when elaborating dependently typed languages is what arguments may the programmer omit. In dependently-typed systems such as Agda or Coq, the programmer declares constants of a given (closed) type and labels arguments that can be freely omitted when subsequently using the constant. Both, Coq and Agda, give the user the possibility to locally override the implicit arguments and provide instantiations explicitly.

In contrast, we follow here a simple, lightweight recipe which comes from the implementation of the logical framework *Elf* [Pfenning, 1989] and its successor *Twelf* [Pfenning and Schürmann, 1999]: *programmers may leave some index variables free when declaring a constant of a given type; elaboration of the type will abstract over these free variables at the outside; when subsequently using this constant, the user must omit passing arguments for those index variables which were left free in the original declaration.* Following this recipe, elaboration of terms and types in the logical framework has been described in Pientka [2013]. Here, we will consider a dependently typed functional programming language which supports pattern matching on index objects.

The key challenge in elaborating recursive programs which support case-analysis is that pattern matching in the dependently typed setting refines index arguments and hence refines types. In contrast to systems such as Coq and Agda, where we must annotate case-expressions with an invariant, i.e. the type of the scrutinee, and the return type, our source language does not require such annotations. Instead we will infer the type of the scrutinee and for each branch, we infer the type of the pattern and compute how the pattern refines the type of the scrutinee. This makes our source language lightweight and closer in spirit to simply-

typed functional languages. Our elaboration of source expressions to target expressions is type-directed, inferring omitted arguments and producing a closed well-typed target program. Finally, we prove soundness of our elaboration, i.e. if elaboration succeeds our resulting program type checks in our core language. Our framework provides post-hoc explanation for elaboration found in the programming and proof environment, Beluga [Pientka and Dunfield, 2010b], where we use as the index domain terms specified in the logical framework LF [Harper et al., 1993].

We describe reconstruction as follows: We first give the grammar of our source language. Showing example programs, we explain informally what elaboration does. We then revisit our core language, describe the elaboration algorithm formally and prove soundness.

3.2 Source Language

We consider here a dependently typed language where types are indexed by terms from an index domain. Our language is similar to Beluga a dependently typed programming environment where we can embed LF objects into computation-level types and computation-level programs which analyze and pattern match on LF objects. However, in our description, as in for example Cave and Pientka [2012], we will keep the index domain abstract, but only assume that equality in the index domain is decidable and unification algorithms exist. This will allow us to focus on the essential challenges when elaborating a dependently typed language in the presence of pattern matching.

The syntax of our source language that allows programmers to omit

some arguments is as follows:

$$
\begin{array}{llcl}
\text{Kinds} & k & ::= & \mathbf{ctype} \mid \{X : u\}\, k \\
\text{Atomic Types} & p & ::= & \mathbf{a}\, \overrightarrow{[c]} \\
\text{Types} & t & ::= & p \mid [u] \mid \{X : u\}\, t \mid t_1 \rightarrow t_2 \\
\text{Expressions} & e & ::= & \mathtt{fn}\, x \Rightarrow e \mid \mathtt{mlam}\, X \Rightarrow e \mid x \mid \mathbf{c} \mid [c] \mid \\
 & & & e_1\, e_2 \mid e_1\, [c] \mid e\ _ \mid \mathtt{case}\, e\, \mathtt{of}\, \vec{b} \mid e : t \\
\text{Branches} & \vec{b} & ::= & b \mid (b \mid \vec{b}) \\
\text{Branch} & b & ::= & pat \mapsto e \\
\text{Pattern} & pat & ::= & x \mid [c] \mid \mathbf{c}\, \overrightarrow{pat} \mid pat : t \\
\text{Declarations} & d & ::= & \mathtt{rec}\, f : t = e \mid \mathbf{c} : t \mid \mathbf{a} : k
\end{array}
$$

As a convention we will use lowercase c to refer to index level objects, lowercase u for index level types, and upper case letters X, Y for index-variables. Index objects can be embedded into computation expressions by using a box modality written as $[c]$. Our language supports functions ($\mathtt{fn}\, x \Rightarrow e$), dependent functions ($\mathtt{mlam}\, X \Rightarrow e$), function application ($e_1\, e_2$), dependent function application ($e_1\, [c]$), and case-expressions. We also support writing underscore (_) instead of providing explicitly the index argument in a dependent function application (e _). Note that we are overloading syntax: we write $e\ [c]$ to describe the application of the expression e of type $[u] \rightarrow t$ to the expression $[c]$; we also write $e\ [c]$ to describe the dependent application of the expression e of type $\{X : u\}t$ to the (unboxed) index object c.This ambiguity can be easily resolved using type information. Note that in our language the dependent function type and the non-dependent function type do not collapse, since we can only quantify over objects of our specific domain instead of arbitrary propositions (types).

We may write type annotations anywhere in the program ($e : t$ and in patterns $pat : t$); type annotations are particularly useful to make explicit the type of a sub-expression and name index variables occurring in the

type. This allows us to resurrect index variables which are kept implicit. In patterns, type annotations are useful since they provide hints to type elaboration regarding the type of pattern variables.

A program signature Σ consists of kind declarations ($\mathbf{a} : k$), type declarations ($\mathbf{c} : t$) and declarations of recursive functions ($\texttt{rec}\ f : t = e$). This can be extended to allow mutual recursive functions in a straightforward way.

One may think of our source language as the language obtained after parsing where for example let-expressions have been translated into case-expression with one branch.

Types for computations include non-dependent function types ($t_1 \to t_2$) and dependent function types ($\{X : u\}t$); we can also embed index types into computation types via $[u]$ and indexed computation-level types by an index domain written as $\mathbf{a}\,\overrightarrow{[c]}$. We also include the grammar for computation-level kinds which emphasizes that computation-level types can only be indexed by terms from an index domain u. We write **ctype** (that reads as "computational type") for the base kind, since we will use **type** for kinds of the index domain.

We note that we only support one form of dependent function type $\{X : u\}t$; the source language does not provide any means for programmers to mark a given dependently typed variable as implicit as for example in Agda. Instead, we will allow programmers to leave some index variables occurring in computation-level types free; elaboration will then infer their types and abstract over them explicitly at the outside. The programmer must subsequently omit providing instantiation for those "free" variables. We will explain this idea more concretely below.

3.2.1 Well-formed Source Expressions

Before elaborating source expressions, we state when a given source expression is accepted as a well-formed expression. In particular, it will

$\boxed{\vdash d\,\mathtt{wf}}$ Declaration d is well-formed

$$\frac{\cdot;\ f \vdash e\,\mathtt{wf} \quad \cdot \vdash_{\mathsf{f}} t\,\mathtt{wf}}{\vdash \mathtt{rec}\ f : t = e\,\mathtt{wf}}\ \mathtt{wf\text{-}rec} \qquad \frac{\cdot \vdash_{\mathsf{f}} t\,\mathtt{wf}}{\vdash c : t\,\mathtt{wf}}\ \mathtt{wf\text{-}types} \qquad \frac{\cdot \vdash_{\mathsf{f}} k\,\mathtt{wf}}{\vdash a : k\,\mathtt{wf}}\ \mathtt{wf\text{-}kinds}$$

$\boxed{\delta;\gamma \vdash_N e\,\mathtt{wf}}$ Normal expression e is well-formed in context δ and γ

$$\frac{\delta;\gamma, x \vdash_N e\,\mathtt{wf}}{\delta;\gamma \vdash_N \mathtt{fn}\, x \Rightarrow e\,\mathtt{wf}}\ \mathtt{wf\text{-}fn} \qquad \frac{\delta, X;\gamma \vdash_N e\,\mathtt{wf}}{\delta;\gamma \vdash_N \mathtt{mlam}\, X \Rightarrow e\,\mathtt{wf}}\ \mathtt{wf\text{-}mlam}$$

$$\frac{\delta \vdash c\,\mathtt{wf}}{\delta;\gamma \vdash_N [c]\,\mathtt{wf}}\ \mathtt{wf\text{-}box} \qquad \frac{\delta;\gamma \vdash_n e\,\mathtt{wf} \quad \text{for all } b_n \text{ in } \vec{b}\,.\ \delta;\gamma \vdash b_n\,\mathtt{wf}}{\delta;\gamma \vdash_N \mathtt{case}\, e\, \mathtt{of}\, \vec{b}\,\mathtt{wf}}\ \mathtt{wf\text{-}case} \qquad \frac{\delta;\gamma \vdash_n e\,\mathtt{wf}}{\delta;\gamma \vdash_N e\,\mathtt{wf}}\ \mathtt{wf\text{-}neu}$$

$\boxed{\delta;\gamma \vdash_n e\,\mathtt{wf}}$ Neutral expression e is well-formed in context δ and γ

$$\frac{\delta;\gamma \vdash_N e\,\mathtt{wf} \quad \delta \vdash t\,\mathtt{wf}}{\delta;\gamma \vdash_n e : t\,\mathtt{wf}}\ \mathtt{wf\text{-}ann} \qquad \frac{\delta;\gamma \vdash_n e_1\,\mathtt{wf} \quad \delta;\gamma \vdash_N e_2\,\mathtt{wf}}{\delta;\gamma \vdash_n e_1\, e_2\,\mathtt{wf}}\ \mathtt{wf\text{-}app}$$

$$\frac{\delta;\gamma \vdash_n e\,\mathtt{wf} \quad \delta \vdash c\,\mathtt{wf}}{\delta;\gamma \vdash_n e\,[c]\,\mathtt{wf}}\ \mathtt{wf\text{-}appe} \qquad \frac{\delta;\gamma \vdash_n e_1\,\mathtt{wf}}{\delta;\gamma \vdash_n e_1\,_\,\mathtt{wf}}\ \mathtt{wf\text{-}apph}$$

$\boxed{\delta;\gamma \vdash pat \mapsto e\,\mathtt{wf}}$ Branch is well-formed in δ and γ

$$\frac{\delta';\gamma' \vdash pat\,\mathtt{wf} \quad \delta,\delta';\gamma,\gamma' \vdash_n e\,\mathtt{wf} \quad \text{where } \delta',\gamma' \text{ introduce fresh vars}}{\delta;\gamma \vdash pat \mapsto e\,\mathtt{wf}}\ \mathtt{wf\text{-}branch}$$

$\boxed{\delta;\gamma \vdash pat\,\mathtt{wf}}$ Pattern pat is well-formed in δ and γ

$$\frac{}{\delta; x \vdash x\,\mathtt{wf}}\ \mathtt{wf\text{-}p\text{-}var} \qquad \frac{\text{for all } p_i \text{ in } \overrightarrow{Pat}.\ \ \delta_i;\gamma_i \vdash p_i\,\mathtt{wf}}{\delta_1,\ldots,\delta_n;\gamma_1,\ldots,\gamma_n \vdash \mathbf{c}\,\overrightarrow{Pat}\,\mathtt{wf}}\ \mathtt{wf\text{-}p\text{-}con}$$

$$\frac{\delta \vdash c\,\mathtt{wf}}{\delta;\cdot \vdash [c]\,\mathtt{wf}}\ \mathtt{wf\text{-}p\text{-}i} \qquad \frac{\delta;\gamma \vdash pat\,\mathtt{wf} \quad \delta \vdash t\,\mathtt{wf}}{\delta;\gamma \vdash pat : t\,\mathtt{wf}}\ \mathtt{wf\text{-}p\text{-}ann}$$

Figure 3.1: Well-formed Source Expressions

$\boxed{\delta \vdash k\,\mathtt{wf}}$ Kind k is well-formed and closed with respect to δ

$$\frac{}{\delta \vdash \mathbf{ctype}\,\mathtt{wf}} \qquad \frac{\delta \vdash u\,\mathtt{wf} \quad \delta, X{:}u \vdash k\,\mathtt{wf}}{\delta \vdash \{X{:}u\}\,k\,\mathtt{wf}}$$

$\boxed{\delta \vdash t\,\mathtt{wf}}$ Type t is well-formed and closed with respect to δ

$$\frac{\delta; \vdash c_i\,\mathtt{wf} \quad \text{for all } c_i \text{ in } \overrightarrow{c}}{\delta \vdash \mathsf{a}[\overrightarrow{c}]\,\mathtt{wf}} \qquad \frac{\delta \vdash u\,\mathtt{wf}}{\delta \vdash [u]\,\mathtt{wf}}$$

$$\frac{\delta \vdash u\,\mathtt{wf} \quad \delta, X{:}u \vdash t\,\mathtt{wf}}{\delta \vdash \{X{:}u\}\,t\,\mathtt{wf}} \qquad \frac{\delta \vdash t_1\,\mathtt{wf} \quad \delta \vdash t_2\,\mathtt{wf}}{\delta \vdash t_1 \to t_2\,\mathtt{wf}}$$

Figure 3.2: Well-formed Kinds and Types

highlight that free index variables are only allowed in declarations when specifying kinds and declaring the type of constants and recursive functions. We use δ to describe the list of index variables and γ the list of program variables. We rely on two judgments from the index language:

$\delta \vdash\ c\,\mathtt{wf}$	Index object c is well formed and closed with respect to δ
$\delta \vdash_{\mathsf{f}} c\,\mathtt{wf}$	Index object c is well formed with respect to δ and may contain free index variables

We describe declaratively the well-formedness of declarations and source expressions in Fig. 3.1. The distinction between normal and neutral expressions forces a type annotation where a non-normal program would occur. The normal vs. neutral term distinction is motivated by the bidirectional type-checker presented in Section 3.3.1. The rules for well-formed types and kinds are given in Figure 3.2.

In branches, pattern variables from γ must occur linearly while we put no such requirement on variables from our index language listed in δ. The judgment for well formed patterns synthesizes contexts δ and

γ that contain all the variables bound in the pattern (this presentation is declarative, but algorithmically the two contexts result from the well-formed judgment). Notice that the rules `wp-p-i` and `wp-con` look similar but they operate on different syntactic categories and refer to the judgment for well-formed index terms provided by the index level language. They differ in that the one for patterns synthesizes the δ context that contains the meta-variables bound in the pattern.

3.2.2 Some Example Programs

We next illustrate writing programs in our language and explain the main ideas behind elaboration. We use Beluga syntax in our examples, in these examples we want to focus on the elaboration of computations not the index language.

3.2.2.1 Translating Untyped Terms to Intrinsically Typed Terms

We implement a program to translate a simple language with numbers, booleans and some primitive operations to its intrinsically typed counterpart. This illustrates declaring an index domain, using index computation-level types, and explaining the use and need to pattern match on index objects. We translate a source language that we will call *Untyped* into its typed counter part, we call this representation *Typed*. This translation is basically a type-checker for terms written in *Untyped*.

We first define the syntax of *Untyped* using the recursive datatype `UTm`. Note the use of the keyword **`ctype`** to define a computation-level recursive data-type.

```
inductive UTm : ctype =
| UNum  : Nat → UTm
| UPlus : UTm → UTm → UTm
| UTrue : UTm
| UFalse: UTm
| UNot  : UTm → UTm
| UIf   : UTm → UTm → UTm → UTm
;
```

Terms can be of type `nat` for numbers or `bool` for booleans. Our goal is to define *Typed* our language of typed terms using a computation-level type family `Tm` which is indexed by objects `nat` and `bool` which are constructors of our index type `tp`. Note that `tp` is declared as having the kind **`type`** which implies that this type lives at the index level and that we will be able to use it as an index for computation-level type families.

```
LF tp : type =
| nat  : tp
| bool : tp
;
```

Using indexed families we can now define the type `Tm` that specifies only type correct terms of the language *Typed*, by indexing terms by their type using the index level type `[⊢ tp]`. The square brackets define a box for index language terms and types and because Beluga's index language contains contextual types the empty turnstyle indicates that types are closed.

```
inductive Tm : [⊢ tp] → ctype =
| Num       : Nat → Tm [⊢ nat]
| Plus      : Tm [⊢ nat] → Tm [⊢ nat] → Tm [⊢ nat]
| True      : Tm [⊢ bool]
| False     : Tm [⊢ bool]
| Not       : Tm [⊢ bool] → Tm [⊢ bool]
| If        : Tm [⊢ bool] → Tm [⊢ T] → Tm [⊢ T] → Tm [⊢ T]
;
```

When the `Tm` family is elaborated, the free variable `T` in the `If` constructor will be abstracted over by an implicit Π-type, as in the Twelf [Pfenning and Schürmann, 1999] tradition. Because `T` was left free by the programmer, the elaboration will add an implicit quantifier; when we use the constant `If`, we now must omit passing an instantiation for `T`. For example, we must write `(If True (Num 3) (Num 4))` and elaboration will infer that `T` must be `nat`.

One might ask how we can provide the type explicitly - this is possible indirectly by providing type annotations. For example:

```
If e (e1:TM[⊢ nat]) e2
```

will fix the type of `e1` to be `Tm [⊢ nat]`.

Our goal is to write a program to translate an untyped term `UTm` to its corresponding typed representation. Because this operation might fail for ill-typed `UTm` terms we need an option type to reflect the possibility of failure.

```
inductive TmOpt : ctype =
| None : TmOpt
| Some : {T : [⊢ tp]} Tm [⊢ T] → TmOpt
;
```

A value of type `TmOpt` will either be empty (i.e. `None`) or some term of type `T`. We chose to make `T` explicit here by quantifying over it explicitly using the curly braces. When returning a `Tm` term, the program must now provide the instantiation of `T` in addition to the actual term.

So far we have declared types and constructors for our language. These declarations will be available in a global signature. The next step is to declare a function that will take *Untyped* terms into *Typed* terms if at all possible. Notice that for the function to be type correct it has to respect the specification provided in the declaration of the type `Tm`. We only show a few interesting cases below.

```
rec typecheck : UTm → TmOpt =
fn e ⇒ case e of
| UNum n ⇒ Some [⊢ nat] (Num n)
| UNot e ⇒ (case typecheck e of
  | Some [⊢ bool] x ⇒ Some [⊢ bool] (Not x)
  | other ⇒ None)
| UIf c e1 e2 ⇒  (case typecheck c of
  | Some [⊢ bool] c’ ⇒ (case (typecheck e1 , typecheck e2)
   of
    | (Some [⊢ T] e1’ , Some [⊢ T] e2’) ⇒
      Some [⊢ T] (If c’ e1’ e2’)
    | other ⇒ None)
  | other ⇒ None)
% ... the cases for UPlus, UTrue and UFalse are similar
;
```

In the `typecheck` function the cases for numbers, plus, true and false are completely straightforward. The case for negation (i.e. constructor `UNot`) is interesting because we need to pattern match on the result of type-checking the sub-expression `e` to match its type to `bool` otherwise we cannot construct the intrinsically typed term, i.e. the constructor `Not` requires a boolean term, this requires matching on index level terms. Additionally the case for `UIf` is also interesting because we not only need a boolean condition but we also need to have both branches of the `UIf` term to be of the same type. Again we use pattern matching on the indices to verify that the condition is of type `bool` but notably we use non-linear pattern matching to ensure that the type of the branches coincides. Therefore, by using non-linear patterns we can force two meta-variables to match against the same term. Notably, note that `If` has an implicit argument (the type `T`) which will be inferred during elaboration and the fact that it is used in both branches implies that in an if expression the "then" branch and the "else" branch are of the same type.

In the definition of type `TmOpt` we chose to explicitly quantify over `T`, however another option would have been to leave it implicit. When pattern matching on `Some e`, we would need to resurrect the type of the argument `e` to be able to inspect it and check whether it has the appropriate type. We can employ type annotations, as shown in the code below, to constrain the type of `e`.

```
| UIf c e1 e2 ⇒  (case typecheck c of
  | Some (c’ : [⊢ bool]) ⇒ (case (typecheck e1, typecheck e2
   ) of
    | (Some (e1’ : [⊢ T]) , Some (e2’ : [⊢ T]) ⇒
      Some (If c’ e1’ e2’)
    | other ⇒ None)
  | other ⇒ None)
```

In this first example there is not much to elaborate. The missing argument in `If` and the types of variables in patterns are all that need to be elaborated.

3.2.2.2 Type-preserving Evaluation

Our previous program used dependent types sparingly; most notably there were no dependent types in the type declaration given to the function `typecheck`. We now discuss the implementation of an evaluator, which evaluates type correct programs to values of the same type, to highlight writing dependently typed functions. Because we need to preserve the type information, we index the values by their types in the following manner:

```
inductive Val : [⊢ tp] → ctype =
| VNum  : Nat → Val [⊢ nat]
| VTrue : Val [⊢ bool]
| VFalse: Val [⊢ bool]
;
```

We define a type preserving evaluator below:

```
rec eval : Tm [⊢ T] → Val [⊢ T] = fn e ⇒ case e of
| Num n ⇒ VNum n
| Plus e1 e2 ⇒ (case (eval e1 , eval e2) of
  | (VNum x , VNum y) ⇒ VNum (add x y))
| Not e ⇒ (case eval e of
  | VTrue ⇒ VFalse
  | VFalse ⇒ VTrue)
| If e e1 e2 ⇒  (case eval e of
  | VTrue ⇒ eval e1
  | VFalse ⇒ eval e2)
| True ⇒ VTrue
| False ⇒ VFalse
;
```

First, we specify the type of the evaluation function as:

```
Tm[⊢ T] → Val [⊢ T]
```

where `T` remains free. The user provided type has a free variable (i.e. `T`) that elaboration will abstract over after inferring its type. The type of the variable should be fixed by the places where it is used. Elaboration will therefore abstract over it in the outside by adding it as an implicit parameter (introduced by Π^i, which is an abstraction that we indicate as implicit since the user will not provide instantiations for and that elaboration will reconstruct). We then elaborate the body of the function against Π^i`T:⊢ tp. Tm [⊢ T] → Val [⊢ T]`. It will first need to introduce the appropriate dependent function abstraction in the program before we introduce the non-dependent function `fn` $x \Rightarrow e$. Moreover, we need to infer omitted arguments in the pattern in addition to inferring the type of pattern variables in the `If` case. Since `T` was left free in the type given to `eval`, we must also infer the omitted argument in the recursive calls to `eval`. Finally, we need to keep track of refinements the pattern match induces: our scrutinee has type `Tm [⊢ T]`; pattern matching against `Plus e1 e1` which has type `Tm [⊢ nat]` refines `T` to `nat`.

3.2.2.3 A Certifying Evaluator

So far in our examples, we have used a simply typed index language. We used our index language to specify natural numbers, booleans, and a tagged enumeration that contained labels for the `bool` and `nat` types. In this example we go one step further, and use a dependently typed specification, in fact we take advantage of LF as our index level language as used in Beluga. Using LF at the index language we specify the simply-typed lambda calculus and its operational semantics in Figure 3.3. These rules provide a call by name operational semantic chosen for no reason other than to save one evaluation rule compare to call by value. Using

$$
\begin{array}{llcl}
\text{Types} & T & ::= & \top \mid T_1 \to T_2 \\
\text{Terms} & M, N & ::= & () \mid x \mid \lambda x : T.M \mid MN \\
\text{Context} & \Gamma & ::= & \cdot \mid \Gamma, x : T \\
\text{Values} & V & ::= & \top \mid \lambda x : T.M
\end{array}
$$

$\boxed{\Gamma \vdash M : T}$ Term M has type T in context Γ

$$
\frac{}{\Gamma \vdash () : \top} \quad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x : T_1.M : T_2}
$$

$$
\frac{\Gamma \vdash M : T_1 \to T \quad \Gamma \vdash N : T_1}{\Gamma \vdash MN : T}
$$

$\boxed{M \Downarrow V}$ Term M steps to value V

$$
\frac{}{\top \Downarrow \top} \quad \frac{}{\lambda x : T.M \Downarrow \lambda x : T.M}
$$

$$
\frac{M \Downarrow \lambda x : T.M' \quad [N/x]M' \Downarrow N'}{MN \Downarrow N'}
$$

Figure 3.3: Example: A Simply-typed λ-calculus

these specifications we write a recursive function that returns the value of the program together with a derivation tree that shows how the value was computed. This example requires dependent types at the index level and consequently the elaboration of functions that manipulate these specifications has to be more powerful.

As in the previous example, we define the types of terms of our language using the index level language. As opposed to the type preserving evaluator, in this case we define our intrinsically typed terms also using the index level language (which will be LF for this example). We take advantage of LF to represent binders in λ-terms, and use dependent types to represent well-typed terms only.

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

LF term : tp → type =
| one : term unit
| lam : (term A → term B) → term (arr A B)
| app : term (arr A B) → term A → term B
;
```

These datatypes represent an encoding of well typed terms of a simply-typed lambda calculus with `unit` as a base type. Using LF we can also describe what constitutes a value and a big-step operational semantics. We use the standard technique in LF to represent binders with the function space (usually called Higher Order Abstract Syntax, HOAS [Pfenning and Elliott, 1988]) and type families to only represent well-typed terms, thus this representation combines the syntax for terms with the typing judgment from Figure 3.3.

```
LF value : tp → type =
| v-one : value unit
| v-lam : (term A → term B) → value (arr A B)
;

LF big-step : term T → value T → type =
| e-one : big-step one v-one
| e-lam : big-step (lam M) (v-lam M)
| e-app : big-step M (v-lam M') →
          big-step (M' N) N' →
          big-step (app M N) N'
;
```

The `value` type simply states that `one` and lambda terms are values, and the type `big-step` encodes the operational semantics where each constructor corresponds to one of the rules in Figure 3.3. The constructors

`e-one` and `e-lam` simply state that `one` and lambdas step to themselves. On the other hand rule `e-app` requires that in an application, the first term evaluates to a lambda expression (which is always the case as the terms are intrinsically well typed) and then it performs the substitution and continues evaluating the term to a value. Note how the substitution is performed by an application as we reuse the LF function space for binders as typically done with HOAS.

To implement a certifying evaluator we want the `eval` function to return a value and a derivation tree that shows how we computed this value. We encode this fact in the `Cert` data-type that encodes an existential or dependent pair that combines a value with a derivation tree.

```
inductive Cert : [⊢ term T] → ctype =
| Ex : {N: [⊢ value T]} [⊢ big-step M N] → Cert [⊢ M]
;
```

In the `Ex` constructor we have chosen to explicitly quantify over `N`, the value of the evaluation, and left the starting term `M` implicit. However another option would have been to leave both implicit, and use type annotations when pattern matching to have access to both the term and its value.

Finally the evaluation function simply takes a term and returns a certificate that contains the value the terms evaluates to, and the derivation tree that led to that value.

```
rec eval : {M : [⊢ term T]} Cert [⊢ M] =
mlam M ⇒ case [⊢ M] of
| [⊢ one] ⇒ Ex [⊢ v-one] [⊢ e-one]
| [⊢ lam (λx.M)] ⇒ Ex [⊢ v-lam (λx.M)] [⊢ e-lam]
| [⊢ app M N] ⇒
  let Ex [⊢ v-lam (λx. M’)] [⊢ D]  = eval [⊢ M] in
  let Ex [⊢ N’][⊢ D’] = eval [⊢ M’[N]] in
  Ex [⊢ N’] [⊢ e-app D D’]
;
```

Elaboration of `eval` starts by the type annotation. Inferring the type of variable `T` and abstracting over it, resulting in:

```
Πⁱ T:[⊢ tp]. {M : [⊢ term T]} Cert [⊢ M]
```

The elaboration proceeds with the body, abstracting over the inferred dependent argument with **mlam** `T ⇒ ...` When elaborating the case expression, the patterns in the index language will need elaboration. In this work we assume that each index language comes equipped with an appropriate notion of elaboration (described in [Pientka, 2013] for the logical framework LF). For example, index level elaboration will abstract over free variables in constructors and the pattern for lambda terms becomes `[lam A B (λx. M)]` when the types for parameters and body are added. Additionally, in order to keep the core language as lean as possible we desugar **let** expressions into **case** expressions. For example, in the certifying evaluator, the following code from `eval`:

```
let Ex [⊢ v-lam M'] [⊢ D]  = eval [⊢ M] in
let Ex [⊢ N'][⊢ D'] = eval [⊢ M' [N]] in
Ex [⊢ N'] [⊢ e-app D D']
```

is desugared into:

```
(case eval [⊢ M] of | Ex [⊢ v-lam M'] [⊢ D] ⇒
   (case eval [⊢ M' [N]] of
   | Ex [⊢ N'][⊢ D'] ⇒ Ex [⊢ N'] [⊢ λe-app D D']))
```

We will come back to this example and discuss the fully elaborated program in the next section.

3.3 Target Language

The *target language* is similar to the computational language described in Cave and Pientka [2012] which has a well developed meta-theory including descriptions of coverage [Dunfield and Pientka, 2009, Jacob-Rao, 2017] and termination [Pientka and Abel, 2015]. The target language

Kinds	K	::=	$\mathtt{\textbf{ctype}} \mid \Pi^e X : U.\, K \mid \Pi^i X : U.\, K$
Types	T	::=	$\Pi^e X : U.\, T \mid \Pi^i X : U.\, T$
			$P \mid U \mid T_1 \rightarrow T_2 \mid \mathbf{a} \mid T\, C$
Expressions	E	::=	$\mathtt{fn}\, x \Rightarrow E \mid \mathtt{mlam}\, X \Rightarrow E$
			$\mid E_1\, E_2 \mid E_1\, C \mid C \mid$
			$\mathtt{case}\, E\, \mathtt{of}\, \vec{B} \mid x \mid E : T \mid \mathbf{c}$
Branches	$\vec{B}$	::=	$B \mid (B \mid \vec{B})$
Branch	B	::=	$\Pi^i \Delta; \Gamma. \mathit{Pat} : \theta \mapsto E$
Pattern	Pat	::=	$x \mid C \mid \mathbf{c}\, \overrightarrow{\mathit{Pat}}$
Declarations	D	::=	$\mathbf{c} : T \mid \mathbf{a} : K \mid \mathtt{rec}\, f : T = E$
Context	Γ	::=	$\cdot \mid \Gamma, x : T$
Index-Var-Context	Δ	::=	$\cdot \mid \Delta, X : U$
Refinement	θ	::=	$\cdot \mid \theta, C/X \mid \theta, X/X$

Figure 3.4: Target Language

(see Fig. 3.4), which is similar to our source language, is indexed by fully explicit terms of the index level language; we use C for fully explicit index level objects, and U for elaborated index types; index-variables occurring in the target language will be represented by capital letters such as X, Y. Moreover, we rely on a substitution which replaces index variables X with index objects. The main difference between the source and target language is in the description of branches. In each branch, we make the type of the pattern variables (see context Γ) and variables occurring in index objects (see context Δ) explicit. We associate each pattern with a refinement substitution θ which specifies how the given pattern refines the type of the scrutinee.

$\boxed{\vdash D\,\mathsf{wf}}$ Target declaration D is well-formed

$$\frac{\cdot \vdash T \Leftarrow \mathbf{ctype} \quad \cdot\,;\, f : T \vdash E \Leftarrow T}{\vdash \mathsf{rec}\, f : T = E\,\mathsf{wf}}\ \texttt{t-rec} \qquad \frac{\cdot \vdash T \Leftarrow \mathbf{ctype}}{\vdash \mathbf{c} : T\,\mathsf{wf}}\ \texttt{t-type} \qquad \frac{\cdot \vdash K \Leftarrow \mathsf{kind}}{\vdash \mathbf{a} : K\,\mathsf{wf}}\ \texttt{t-kind}$$

$\boxed{\Delta \vdash T : K}$ T is a well-kinded type of kind K

$$\frac{\Delta \vdash T_1 : \mathbf{ctype} \quad \Delta \vdash T_2 : \mathbf{ctype}}{\Delta \vdash T_1 \to T_2 : \mathbf{ctype}}\ \texttt{k-arr} \qquad \frac{\Delta \vdash U : \mathbf{type} \quad \Delta, X : U \vdash T : \mathbf{ctype}}{\Delta \vdash \Pi^{\{e,i\}} X : U.T : \mathbf{ctype}}\ \texttt{k-pi}$$

$$\frac{\Delta \vdash U : \mathbf{type}}{\Delta \vdash U : \mathbf{ctype}}\ \texttt{k-idx} \qquad \frac{\Delta \vdash C \Leftarrow U \quad \Delta, X : U \vdash T : K}{\Delta \vdash T\,C : \Pi^{\{e,i\}} X : U.K}\ \texttt{k-app} \qquad \frac{\Sigma(\mathbf{a}) = K}{\Delta \vdash \mathbf{a} : K}\ \texttt{k-con}$$

$\boxed{\Delta; \Gamma \vdash E \Rightarrow T}$ E synthesizes type T

$$\frac{\Delta;\Gamma \vdash E_1 \Rightarrow S \to T \quad \Delta;\Gamma \vdash E_2 \Leftarrow S}{\Delta;\Gamma \vdash E_1\, E_2 \Rightarrow T}\ \texttt{t-app} \qquad \frac{\Delta;\Gamma \vdash E \Rightarrow \Pi^{*} X : U.T \quad * = \{i, e\} \quad \Delta \vdash C : U}{\Delta;\Gamma \vdash E\, C \Rightarrow [C/X]T}\ \texttt{t-app-index}$$

$$\frac{\Sigma(\mathbf{c}) = T}{\Delta;\Gamma \vdash \mathbf{c} \Rightarrow T}\ \texttt{t-const} \qquad \frac{\Gamma(x) = T}{\Delta;\Gamma \vdash x \Rightarrow T}\ \texttt{t-var} \qquad \frac{\Delta;\Gamma \vdash E \Leftarrow T}{\Delta;\Gamma \vdash E : T \Rightarrow T}\ \texttt{t-ann}$$

$\boxed{\Delta; \Gamma \vdash E \Leftarrow T}$ E type checks against type T

$$\frac{\Delta;\Gamma \vdash E \Rightarrow T}{\Delta;\Gamma \vdash E \Leftarrow T}\ \texttt{t-syn} \qquad \frac{\Delta;\Gamma, x : T_1 \vdash E \Leftarrow T_2}{\Delta;\Gamma \vdash (\mathsf{fn}\, x \Rightarrow E) \Leftarrow T_1 \to T_2}\ \texttt{t-fn}$$

$$\frac{\Delta, X : U;\Gamma \vdash E \Leftarrow T \quad * = \{i, e\}}{\Delta;\Gamma \vdash (\mathsf{mlam}\, X \Rightarrow E) \Leftarrow \Pi^{*} X : U.T}\ \texttt{t-mlam} \qquad \frac{\Delta;\Gamma \vdash E \Rightarrow S \quad \Delta;\Gamma \vdash \overrightarrow{B} \Leftarrow S \to T}{\Delta;\Gamma \vdash \mathsf{case}\, E\, \mathsf{of}\, \overrightarrow{B} \Leftarrow T}\ \texttt{t-case}$$

$\boxed{\Delta; \Gamma \vdash \Pi\Delta';\Gamma'.Pat : \theta \mapsto E \Leftarrow T}$ Branch $B = \Pi\Delta';\Gamma'.Pat : \theta \mapsto E$ checks against type T

$$\frac{\Delta' \vdash \theta : \Delta \quad \Delta';\Gamma' \vdash Pat \Leftarrow [\theta]S \quad \Delta';[\theta]\Gamma, \Gamma' \vdash E \Leftarrow [\theta]T}{\Delta;\Gamma \vdash \Pi\Delta';\Gamma'.Pat : \theta \mapsto E \Leftarrow S \to T}\ \texttt{t-branch}$$

$\boxed{\Delta; \Gamma \vdash Pat \Leftarrow T}$ Pattern Pat checks against T

$$\frac{\Delta \vdash C \Leftarrow U}{\Delta;\Gamma \vdash C \Leftarrow U}\ \texttt{t-pindex} \qquad \frac{\Gamma(x) = T}{\Delta;\Gamma \vdash x \Leftarrow T}\ \texttt{t-pvar} \qquad \frac{\Sigma(\mathbf{c}) = T \quad \Delta;\Gamma \vdash \overrightarrow{Pat} \Leftarrow T \rangle S}{\Delta;\Gamma \vdash \mathbf{c}\, \overrightarrow{Pat} \Leftarrow S}\ \texttt{t-pcon}$$

$\boxed{\Delta; \Gamma \vdash \overrightarrow{Pat} \Leftarrow T \rangle S}$ Pattern spine $\overrightarrow{Pat}$ checks against T and has result type S

$$\frac{\Delta \vdash C \Leftarrow U \quad \Delta;\Gamma \vdash \overrightarrow{Pat} \Leftarrow [C/X]T \rangle S}{\Delta;\Gamma \vdash C\, \overrightarrow{Pat} \Leftarrow \Pi^{e} X : U.T \rangle S}\ \texttt{t-spi} \qquad \frac{\Delta;\Gamma \vdash Pat \Leftarrow T_1 \quad \Delta;\Gamma \vdash \overrightarrow{Pat} \Leftarrow T_2 \rangle S}{\Delta;\Gamma \vdash Pat\, \overrightarrow{Pat} \Leftarrow T_1 \to T_2 \rangle S}\ \texttt{t-sarr} \qquad \frac{}{\Delta;\Gamma \vdash \cdot \Leftarrow S \rangle S}\ \texttt{t-snil}$$

Figure 3.5: Typing of Computational Expressions

3.3.1 Typing of the Target Language

The kinding and typing rules for our core language are given in Fig. 3.5. We use a bidirectional type system [Pierce and Turner, 2000] for the target language which is similar to the one in Cave and Pientka [2012] but we simplify the presentation by omitting recursive types. Instead we assume that constructors together with their types are declared in a signature Σ. We choose a bi-directional type-checkers because it minimizes the need for annotations by propagating known typing information in the checking phase (judgment $\Delta; \Gamma \vdash E \Leftarrow T$) and inferring the types when it is possible in the synthesis phase (judgment $\Delta; \Gamma \vdash E \Rightarrow T$).

We rely on the fact that our index domain comes with rules which check that a given index object is well-typed. This is described by the judgment: $\Delta \vdash C : U$.

We check the introductions forms for functions $\texttt{fn}\, x \Rightarrow e$ and dependent functions $\texttt{mlam}\, x \Rightarrow e$ against their respective types. Dependent functions check against both $\Pi^e X{:}U.\, T$ and $\Pi^i X{:}U.\, T$ where types are annotated with e for explicit quantification and i for implicit quantification filled in by elaboration. Their corresponding eliminations, application $E_1\, E_2$ and dependent application $E\,[C]$, synthesize their type. We rely in this rule on the index-level substitution operation and we assume that it is defined in such a way that normal forms are preserved[1].

To type-check a case-expressions $\texttt{case}\, E\, \texttt{of}\, \vec{B}$ against T, we synthesize a type S for E and then check each branch against $S \to T$. A branch $\Pi\Delta'; \Gamma'.Pat : \theta \mapsto E$ checks against $S \to T$, if: 1) θ is a refinement substitution mapping all index variables declared in Δ to a new context Δ', 2) the pattern Pat is compatible with the type S of the scrutinee, i.e. Pat has type $[\theta]S$, and the body E checks against $[\theta]T$ in the index context Δ' and the program context $[\theta]\Gamma, \Gamma_i$. Note that the refinement substitution

[1] In Beluga, this is for example achieved by relying on hereditary substitutions[Cave and Pientka, 2012].

effectively performs a context shift.

We present the typing rules for patterns in spine format which will simplify our elaboration and inferring types for pattern variables. We start checking a pattern against a given type and check index objects and variables against the expected type. If we encounter $\mathbf{c}\,\overrightarrow{Pat}$ we look up the type T of the constant **c** in the signature and continue to check the spine $\overrightarrow{Pat}$ against T with the expected return type S. Pattern spine typing succeeds if all patterns in the spine $\overrightarrow{Pat}$ have the corresponding type in T and yields the return type S.

3.3.2 Elaborated Examples

In Section 3.2.2.3 we give an evaluator for a simply typed lambda calculus that returns the result of the evaluation together with the derivation tree needed to compute the value. The elaborated version of function `eval` is:

```
rec eval : Πi.T:[⊢ tp]. {M : [⊢ term T]} Cert [⊢ T][⊢ M] =
mlam T ⇒ mlam M ⇒ case [⊢ M] of
| Πi.;. [⊢ one]: unit/T ⇒ Ex [⊢ unit] [⊢ v-one] [⊢ e-one]
| Πi.T1:[⊢ tp], T2:[⊢ tp], M : [x:T1⊢term T2];.
  [⊢ lam T1 T2 (λx.M)] : arr T1 T2/ T ⇒
  Ex [⊢ arr T1 T2]
     [⊢ v-lam T1 T2 (λx. M)]
     [⊢ e-lam (λx. M)]
| Πi.T1:[⊢ tp], T2:[⊢ tp],
  M:[⊢ term (arr T1 T2)], N:[⊢ term T1];.
  [⊢ app T1 T2 M N]: T2/T ⇒
    (case eval [⊢ arr T1 T2] [⊢ M] of
    | Πi.T1:[⊢ tp],T2:[⊢ tp],M':[x:T1⊢term T2],
      D: [⊢ big-step (arr T1 T2) M'
             (v-lam (λx.M))];.
      Ex [⊢ arr T1 T2][⊢ v-lam (arr T1 T2) M'][⊢ D]:.⇒
      (case eval [⊢ T2] [⊢ M' [N]] of
     | Πi.T2:[⊢ tp], N':[⊢ val T2],
       D':[⊢ big-step T2 (M' [N]) N'];.
       Ex [⊢ T2] [⊢ N'][⊢ D']:. ⇒
         Ex [⊢ T2] [⊢ N']
            [⊢ e-app M M' N N' D D']))
;
```

To elaborate a recursive declaration we start by reconstructing the type annotation given to the recursive function. In this case the user left the variable `T` free which becomes an implicit argument and we abstract over this variable with Π^i`. T:[⊢ tp]` marking it implicit. Notice however how the user explicitly quantified over `M` this means that callers of `eval`

have to provide the term `M` while parameter `T` will be omitted and inferred at each calling point. Next, we elaborate the function body given the fully elaborated type. We therefore add the corresponding abstraction
`mlam` `T⇒` for the implicit parameter.

Elaboration proceeds recursively on the term. We reconstruct the case-expression, considering first the scrutinee `[M]` and we infer its type to be `[term T]`. We elaborate the branches next. Recall that each branch in the source language consists of a pattern and a body. Moreover, the body can refer to any variable in the pattern or variables introduced in outer patterns. However, in the target language branches abstract over the context $\Delta;\Gamma$ and add a refinement substitution θ. The body of the branch refers to variables declared in the branch contexts only. In each branch, we list explicitly the index variables and pattern variables. For example in the branch for `[lam M]` we added `T1` and `T2` to the index context Δ of the branch, index-level reconstruction adds these variables to the pattern. The refinement substitution moves terms from the outer context to the branch context, refining the appropriate index variables as expressed by the pattern. For example in this branch, the substitution refines the type `[T]` to the type `[arr T1 T2]`. And in the `[one]` branch it refines the type `[T]` to `[unit]`.

As we mentioned before, elaboration adds an implicit parameter to the type of function `eval`, and the user is not allowed to directly supply an instantiation for it. Implicit parameters have to be inferred by elaboration. In the recursive calls to `eval`, we add the parameter that represents the type of the term being evaluated.

The output of the elaboration process is a target language term that can be type checked with the rules from Figure 3.5.

If elaboration fails it can either be because the source level program describes a term that would be ill-typed when elaborated, or in some cases, elaboration fails because it cannot infer all the implicit parameters.

This might happen if unification for the index language is undecidable, as is for example the case for contextual LF. In this case, annotations are needed when the term falls outside the strict pattern fragment where unification is decidable; this is rarely a problem in practice. For other index languages where unification is decidable, we do not expect such annotations to be necessary.

3.4 Description of Elaboration

Elaboration of our source-language to our core target language is guided by the expected target type using a bi-directional algorithm. Recall that we mark in the target type the arguments which are implicitly quantified (e.g.: $\Pi^i X : U. T$). This annotation is added when we elaborate a source type with free variables. If we check a source expression against $\Pi^i X : U. T$ we insert the appropriate mlam-abstraction in our target. When we switch between synthesizing a type S for a given expression and checking an expression against an expected type T, we will rely on unification to make them equal. A key challenge is how to elaborate case-expressions where pattern matching a dependently typed expression of type τ against a pattern in a branch might refine the type τ. Our elaboration is parametric in the index domain, hence we keep our definitions of holes, instantiation of holes and unification abstract and only state their definitions and properties.

3.4.1 Elaboration of Index Objects

To elaborate a source expression, we insert holes for omitted index arguments and elaborate index objects which occur in it. We characterize holes with contextual objects as in [Nanevski et al., 2008, Pientka, 2009]. Contextual objects encode the dependencies on the context that the hole

$$\frac{\cdot;\cdot \mid \cdot \vdash t \rightsquigarrow T/\Theta;\Delta;\cdot \quad \Delta_i \vdash \epsilon : \Theta}{\vdash \mathbf{c} : t \rightsquigarrow \Pi^i(\Delta_i, [\![\epsilon]\!]\Delta).\, [\![\epsilon]\!]T}\ \texttt{el-typ}$$

$$\frac{\cdot;\cdot \mid \cdot \vdash k \rightsquigarrow K/\Theta;\Delta;\cdot \quad \Delta_i \vdash \epsilon : \Theta}{\vdash \mathbf{a} : k \rightsquigarrow \Pi^i(\Delta_i, [\![\epsilon]\!]\Delta).\, [\![\epsilon]\!]K}\ \texttt{el-kind}$$

$$\frac{\begin{array}{c}\cdot;\cdot \mid \cdot \vdash t \rightsquigarrow T/\Theta;\Delta;\cdot \\ \Delta_i \vdash \epsilon : \Theta \\ \cdot; f : \Pi^i\Delta_i, [\![\epsilon]\!]\Delta.\, T \vdash \wr e\,;\,\cdot \wr : \Pi^i(\Delta_i, [\![\epsilon]\!]\Delta).\, [\![\epsilon]\!]T \rightsquigarrow E/\cdot;\cdot\end{array}}{\vdash \texttt{rec}\, f : t = e \rightsquigarrow \texttt{rec}\, f : \Pi^i(\Delta_i, [\![\epsilon]\!]\Delta).\, [\![\epsilon]\!]T = E}\ \texttt{el-rec}$$

Figure 3.6: Elaborating Declarations

might have. We hence make a few requirements about our index domain. We assume:

1. A function genHole ($?Y : \Delta \vdash U$) that generates a term standing for a hole of type U in the context Δ, i.e. its instantiation may refer to the index variables in Δ. If the index language is first-order, then we can characterize holes for example by meta-variables [Nanevski et al., 2008]. If our index language is higher-order, for example if we choose contextual LF as in Beluga, we characterize holes using meta^2-variables as described in Boespflug and Pientka [2011]. As is common in these meta-variable calculi, holes are associated with a delayed substitution θ which is applied as soon as we know what Y stands for.

2. A typing judgment for guaranteeing that index objects with holes are well-typed:

$$\Theta; \Delta \vdash C : U \quad \text{Index object } C \text{ has index type } U \text{ in context } \Delta \text{ and all holes in } C \text{ are declared in } \Theta$$

   where Θ stores the hole typing assumptions:

$$\text{Hole Context} \quad \Theta \quad ::= \quad \cdot \mid \Theta, ?X : \Delta \vdash U$$

3. Unification algorithm which finds the most general unifier for two index objects. In Beluga, we rely on the higher-order unification; more specifically, we solve eagerly terms which fall into the pattern fragment [Miller, 1991, Dowek et al., 1996] and delay others [Abel and Pientka, 2011]. A most general unifier exists if all unification constraints can be solved. Our elaboration relies on unifying computation-level types which in turn relies on unifying index-level terms; technically, we in fact rely on two unification judgments: one finding instantiations for holes in Θ, the other finding most general instantiations for index variables defined in Δ such that two index terms become equal. We use the first one during elaboration when unifying two computation-level types; the second one is used when computing the type refinement in branches.

$$\begin{array}{rcll} \Theta; \Delta & \vdash & C_1 \doteq C_2 / \Theta'; \rho & \quad \text{where: } \Theta' \vdash \rho : \Theta \\ \Delta & \vdash & C_1 \doteq C_2 / \Delta'; \theta & \quad \text{where: } \Delta' \vdash \theta : \Delta \end{array}$$

   where ρ describes the instantiation for holes in Θ. If unification succeeds, then we have $[\![\rho]\!]C_1 = [\![\rho]\!]C_2$ and $[\theta]C_1 = [\theta]C_2$ respectively.

4. Elaboration of index objects themselves. If the index language is simply typed, the elaboration has nothing to do; however, if as

in Beluga, our index objects are objects described in the logical framework LF, then we need to elaborate them and infer omitted arguments following [Pientka, 2013].

3.4.1.1 Index Language Elaboration

In this chapter we focus on the elaboration of the computational language, therefore we assume the existence of the counter part for the index domain. In this section we summarize the requirements on the index domain.

Well-typed Index Objects (target)

$\Delta \vdash C : U$ Index object C has index type U in context Δ

Substitution C/X in an index object C' is defined as $[C/X]C'$.

Well-typed Index Objects with Holes

$\Theta; \Delta \vdash C : U$ Index object C has index type U in context Δ and all holes in C are well-typed wrt Θ

Hole types		$::=$	$\Delta \vdash U$
Hole Contexts	Θ	$::=$	$\cdot \mid \Theta, ?X : \Delta \vdash U$
Hole Inst.	ρ	$::=$	$\cdot \mid \rho, \hat{\Delta} \vdash C/?X$

When we insert hole variables for omitted arguments in a given context Δ, we rely on the abstract function genHole $(?Y : \Delta \vdash U)$ which returns an index term containing a new hole variable. When instantiating a hole we only need the names of the variables in the context for α-conversion, we represent this as $\hat{\Delta}$. We assume the existence of an operation `id`(Θ) that computes the identity substitution on the hole context Θ by replacing each variable with itself.

$$\text{genHole}\ (?Y : \Delta \vdash U) \;=\; C \qquad \text{where } C \text{ describes a hole.}$$

Unification of index objects

The notion of unification that elaboration needs depends on the index level language. As we mentioned, we require that equality on our index domain is decidable; for elaboration, we also require that there is a decidable unification algorithm which makes two terms equal. In fact, we need two forms: one which allows us to infer instantiations for holes and another which unifies two index objects finding most general instantiations for index variables such that the two objects become equal. We use the first one during elaboration, the second one is used to make two index objects equal as for example during matching.

$$\begin{array}{rcll} \Theta;\Delta & \vdash & C_1 \doteq C_2/\Theta';\rho & \text{where: } \Theta' \vdash \rho : \Theta \\ \Delta & \vdash & C_1 \doteq C_2/\Delta';\theta & \text{where: } \Delta' \vdash \theta : \Delta \end{array}$$

where ρ describes the instantiation for holes in Θ. If unification succeeds, then we have $[\![\rho]\!]C_1 = [\![\rho]\!]C_2$ and $[\theta]C_1 = [\theta]C_2$ respectively.

3.4.1.2 Elaboration of index objects

We describe the elaboration of index objects themselves. There are two related judgements of elaboration for index objects that we use:

$$\begin{array}{ll} \Theta;\Delta & \vdash c : U \rightsquigarrow C/\Theta';\Delta';\rho \\ \Theta;\Delta & \vdash \wr c\ ;\theta \int : U \rightsquigarrow C/\Theta';\rho \end{array}$$

The first judgment elaborates the index object c by checking it against U. We thread through a context Θ of holes and a context of index variables Δ, we have seen so far. The object c however may contain additional free index variables whose type we infer during elaboration. All variables occurring in C will be eventually declared with their corresponding type in Δ'. As we elaborate c, we may refine holes and add additional holes. ρ describes the mapping between Θ and Θ', i.e. it records refinement of holes. Finally, we know that $\Delta' = [\![\rho]\!]\Delta, \Delta_0$, i.e. Δ' is an extension of Δ.

We use the first judgment in elaborating patterns and type declarations in the signature.

The second judgment is similar to the first, but does not allow free index variables in c. We elaborate c together with a refinement substitution θ, which records refinements obtained from earlier branches. When we encounter an index variable, we look up what it is mapped to in θ and return it. Given a hole context Θ and a index variable context Δ, we elaborate an index term c against a given type U. The result is two fold: a context Θ' of holes is related to the original hole context Θ via the hole instantiation ρ. We use the second judgment to elaborate index objects embedded into target expressions.

3.4.2 Elaborating Declarations

We begin our discussion of elaborating source programs in a top-down manner starting with declarations, the entry point of the algorithm. Types and kinds in declarations may contain free variables and there are two different tasks: we need to fill in omitted arguments, infer the type of free variables and abstract over the free variables and holes which are left over in the elaborated type / kind. We rely here on the fact that the index language provides a way of inferring the type of free variables.

To abstract over holes in a given type T, we employ a lifting operation: $\Delta \vdash \epsilon : \Theta$ which maps each hole to a fresh index variable.

$$\frac{}{\cdot \vdash \cdot : \cdot} \qquad \frac{\Delta \vdash \epsilon : \Theta}{\Delta, X : U \;\; \vdash \epsilon,\ (\vdash X)/X : \Theta, X : (\vdash U)}$$

We require that holes are closed (written as $\vdash U$ and $\vdash X$ respectively where the context associated with a hole is empty); otherwise lifting fails. [2] In other words, holes are not allowed to depend on some local meta-variables.

[2]In practice this seems to be not an important restriction, for instance none of Beluga's examples need to reconstruct open holes.

We use double brackets (i.e. $[\![\epsilon]\!]M$) to represent the application of the lifting substitutions and hole instantiation substitutions. We use this to distinguish them from regular substitutions such as the refinement substitutions in the target language.

Elaborating declarations requires three judgments. One for constants and one for kinds to be able to reconstruct inductive type declarations, and one for recursive functions. These judgments are:

$$\begin{array}{lll} \vdash & \mathbf{c}:t & \rightsquigarrow T \\ \vdash & \mathbf{a}:k & \rightsquigarrow K \\ \vdash & \texttt{rec}\, f:t=e & \rightsquigarrow \texttt{rec}\, f:T=E \end{array}$$

The elaboration of declarations succeeds when the result does not contain holes.

Figure 3.6 shows the rules for elaborating declarations. To elaborate a constant declaration $\mathbf{c} : t$ we elaborate the type t to a target type T where free index variables are listed in Δ and the remaining holes in T are described in Θ. We then lift all the holes in Θ to proper declarations in Δ_i via the lifting substitution ϵ. The final elaborated type of the constant $\mathbf{c}$ is: $\Pi^i(\Delta_i, [\![\epsilon]\!]\Delta). [\![\epsilon]\!]T$. Note that both the free variables in the type t and the lifted holes described in Δ_i form the implicit arguments and are marked with Π^i. For example in the certifying evaluator from Section 3.2.2.3, the type of the constructor `Ex` is reconstructed to:

```
ΠⁱT:[⊢ tp],M:[⊢ term T].ΠᵉN:[⊢ value T]. [⊢ big-step T M N]
    → Cert [⊢ T][⊢ M]
```

The elaboration of kinds follows the same principle. Section 3.4.3 explains the details for the elaboration of types and kinds.

To elaborate recursive function declarations, we first elaborate the type t abstracting over all the free variables and lifting the remaining holes to obtain $\Pi^i(\Delta_i, [\![\epsilon]\!]\Delta). [\![\epsilon]\!]T$. Second, we assume f of this type and elaborate the body e checking it against $\Pi^i(\Delta_i, [\![\epsilon]\!]\Delta). [\![\epsilon]\!]T$. We note

that we always elaborate a source expression e together with a possible refinement substitution θ. In the beginning, θ will be empty. We describe elaboration of source expressions in Section 3.4.4.

$\boxed{\Theta; \Delta_f \mid \Delta \vdash k \rightsquigarrow K / \Theta'; \Delta'_f; \rho'}$ Elaborate kind k to target kind K

$$\frac{}{\Theta; \Delta_f \mid \Delta \vdash \mathbf{ctype} \rightsquigarrow \mathbf{ctype} / \Theta; \Delta_f; \mathtt{id}(\Theta)}\ \texttt{el-k-ctype}$$

$$\frac{\Theta; \Delta_f \mid \Delta \vdash u \rightsquigarrow U / \Theta'; \Delta'_f; \rho' \quad \Theta'; \Delta'_f \mid \Delta, X{:}U \vdash k \rightsquigarrow K / \Theta''; \Delta''_f; \rho''}{\Theta; \Delta_f \mid \Delta \vdash \{X{:}u\}\, k \rightsquigarrow \Pi^e X{:}(\llbracket \rho' \rrbracket U).\, K / \Theta''; \Delta''_f; \rho'' \circ \rho'}\ \texttt{el-k-pi}$$

$\boxed{\Theta; \Delta_f \mid \Delta \vdash t \rightsquigarrow T / \Theta'; \Delta'_f; \rho'}$ Elaborate type t to target type T

$$\frac{\Theta; \Delta_f \mid \Delta \vdash t_1 \rightsquigarrow T_1 / \Theta'; \Delta'_f; \rho' \quad \Theta'; \Delta'_f \mid \Delta \vdash t_2 \rightsquigarrow T_2 / \Theta''; \Delta''_f; \rho''}{\Theta; \Delta_f \mid \Delta \vdash t_1 \to t_2 \rightsquigarrow (\llbracket \rho'' \rrbracket T_1) \to T_2 / \Theta''; \Delta''_f; \rho'' \circ \rho'}\ \texttt{el-t-arr}$$

$$\frac{\Theta; \Delta_f \mid \Delta \vdash u \rightsquigarrow U / \Theta'; \Delta'_f; \rho'}{\Theta; \Delta_f \mid \Delta \vdash [u] \rightsquigarrow [U] / \Theta'; \Delta'_f; \rho'}\ \texttt{el-t-idx}$$

$$\frac{\Theta; \Delta_f \mid \Delta \vdash u \rightsquigarrow U / \Theta'; \Delta'_f; \rho' \quad \Theta'; \Delta'_f \mid \Delta, X{:}U \vdash t \rightsquigarrow T / \Theta''; \Delta''_f; \rho''}{\Theta; \Delta_f \mid \Delta \vdash \{X{:}u\}\, t \rightsquigarrow \Pi^e X{:}(\llbracket \rho' \rrbracket U).\, T / \Theta''; \Delta''_f; \rho'' \circ \rho'}\ \texttt{el-t-pi}$$

$$\frac{\Sigma(\mathbf{a}) = K \quad \Theta; \Delta_f \mid \Delta \vdash \overrightarrow{[c]} : K \rightsquigarrow \overrightarrow{[C]} / \Theta'; \Delta'_f; \rho'}{\Theta; \Delta_f \mid \Delta \vdash \mathbf{a}\, \overrightarrow{[c]} \rightsquigarrow \mathbf{a}\, \vec{C} / \Theta'; \Delta'_f; \rho'}\ \texttt{el-t-con}$$

$\boxed{\Theta; \Delta_f \mid \Delta \vdash \overrightarrow{[c]} : K \rightsquigarrow \overrightarrow{[C]} / \Theta'; \Delta'_f; \rho'}$ Elaborate spine $\overrightarrow{[c]}$ checking against kind K to spine $\overrightarrow{[C]}$

$$\frac{\Theta; \Delta_f \mid \Delta \vdash c : U \rightsquigarrow C / \Theta'; \Delta'_f; \rho' \quad \Theta'; \Delta'_f \mid \Delta \vdash \overrightarrow{[c]} : [C/X]K \rightsquigarrow \overrightarrow{[C]} / \Theta''; \Delta''_f; \rho''}{\Theta; \Delta_f \mid \Delta \vdash [c]\, \overrightarrow{[c]} : \Pi^e X{:}U.\, K \rightsquigarrow (\llbracket \rho' \rrbracket [C])\, \overrightarrow{[C]} / \Theta''; \Delta''_f; \rho'' \circ \rho'}\ \texttt{el-t-sp-explicit}$$

$$\frac{\mathsf{genHole}\ (?Y{:}(\Delta_f, \Delta).U) = C \quad \Theta; \Delta_f \mid \Delta \vdash \vec{c} : [C/X]K \rightsquigarrow \vec{C} / \Theta'; \Delta'_f; \rho'}{\Theta; \Delta_f \mid \Delta \vdash \vec{c} : \Pi^i X{:}U.\, K \rightsquigarrow (\llbracket \rho' \rrbracket C)\, \vec{C} / \Theta'; \Delta'_f; \rho'}\ \texttt{el-t-sp-implicit}$$

$$\frac{}{\Theta; \Delta_f \mid \Delta \vdash \cdot : \mathbf{ctype} \rightsquigarrow \cdot / \Theta; \Delta_f; \mathtt{id}(\Theta)}\ \texttt{el-t-sp-empty}$$

Figure 3.7: Elaborating Kinds and Types in Declarations

3.4.3 Elaborating Kinds and Types in Declarations

Recall that programmers may leave index variables free in type and kind decarations. Elaboration must infer the type of the free index variables in addition to reconstructing omitted arguments. We require that the index language provides us with the following judgments:

$$\Theta; \Delta_f \mid \Delta \vdash u \rightsquigarrow U / \Theta'; \Delta'_f; \rho'$$
$$\Theta; \Delta \vdash \wr u \ ; \theta \int \rightsquigarrow U / \Theta'; \rho'$$

Hence, we assume that the index language knows how to infer the type of free variables, for example. In Beluga where the index language is LF, we fall back to the ideas described by Pientka [2013].

The first judgment collects free variables in Δ_f that later in elaboration will become implicit parameters. The context Δ_f is threaded through in addition to the hole context Θ.

The judgments for elaborating computation-level kinds and types are similar:

1. $\Theta; \Delta_f \mid \Delta \vdash k \rightsquigarrow K / \Theta'; \Delta'_f; \rho'$
2. $\Theta; \Delta_f \mid \Delta \vdash t \rightsquigarrow T / \Theta'; \Delta'_f; \rho'$
3. $\Theta; \Delta_f \mid \Delta \vdash \overrightarrow{[c]} : K \rightsquigarrow \overrightarrow{[C]} / \Theta'; \Delta'_f; \rho'$

We again collect free index variables in Δ_f which are threaded through together with the holes context Θ (see Figure 3.6 and Figure 3.7). As notation note, we use Δ_f to store the free index variables, Δ for the bound ones, and Θ for holes.

3.4.4 Elaborating Source Expressions

We elaborate source expressions bidirectionally. Expressions such as non-dependent functions and dependent functions are elaborated by checking the expression against a given type; expressions such as application

$\boxed{\Theta; \Delta; \Gamma \vdash \wr e \; ; \; \theta \int : T \rightsquigarrow E / \Theta'; \rho}$:
Elaborate source $\wr e \; ; \; \theta \int$ to target expression E checking against type T

$$\frac{\Theta; \Delta \vdash \wr c \; ; \; \theta \int : U \rightsquigarrow C / \Theta'; \rho}{\Theta; \Delta; \Gamma \vdash \wr [c] \; ; \; \theta \int : [U] \rightsquigarrow [C] / \Theta'; \rho} \; \texttt{el-box}$$

$$\frac{\Theta; \Delta; \Gamma, x : T_1 \vdash \wr e \; ; \; \theta \int : T_2 \rightsquigarrow E / \Theta'; \rho}{\Theta; \Delta; \Gamma \vdash \wr \texttt{fn}\, x \Rightarrow e \; ; \; \theta \int : T_1 \to T_2 \rightsquigarrow \texttt{fn}\, x \Rightarrow E / \Theta'; \rho} \; \texttt{el-fn}$$

$$\frac{\Theta; \Delta, X : U; \Gamma \vdash \wr e \; ; \; \theta, X/X \int : T \rightsquigarrow E / \Theta'; \rho}{\Theta; \Delta; \Gamma \vdash \wr e \; ; \; \theta \int : \Pi^i X : U. T \rightsquigarrow \texttt{mlam}\, X \Rightarrow E / \Theta'; \rho} \; \texttt{el-mlam-i}$$

$$\frac{\Theta; \Delta, X : U; \Gamma \vdash \wr e \; ; \; \theta, X/X \int : T \rightsquigarrow E / \Theta'; \rho}{\Theta; \Delta; \Gamma \vdash \wr \texttt{mlam}\, X \Rightarrow e \; ; \; \theta \int : \Pi^e X : U. T \rightsquigarrow \texttt{mlam}\, X \Rightarrow E / \Theta'; \rho} \; \texttt{el-mlam}$$

$$\frac{\begin{array}{c}\Theta; \Delta; \Gamma \vdash \wr e \; ; \; \theta \int \rightsquigarrow E : S / \cdot; \rho \\ [\![\rho]\!]\Delta; [\![\rho]\!]\Gamma \vdash \wr \vec{b} \; ; \; [\![\rho]\!]\theta \int : S \to [\![\rho]\!]T \rightsquigarrow \vec{B}\end{array}}{\Theta; \Delta; \Gamma \vdash \wr \texttt{case}\, e\, \texttt{of}\, \vec{b} \; ; \; \theta \int : T \rightsquigarrow \texttt{case}\, E\, \texttt{of}\, \vec{B} / \cdot; \rho} \; \texttt{el-case}$$

$$\frac{\Theta; \Delta; \Gamma \vdash \wr e \; ; \; \theta \int \rightsquigarrow E : T_1 / \Theta_1; \rho \quad \Theta_1; [\![\rho]\!]\Delta \vdash T_1 \doteq [\![\rho]\!]T / \Theta_2; \rho'}{\Theta; \Delta; \Gamma \vdash \wr e \; ; \; \theta \int : T \rightsquigarrow [\![\rho']\!]E / \Theta_2; \rho' \circ \rho} \; \texttt{el-syn}$$

Figure 3.8: Elaboration of Expressions (Checking Mode)

and dependent application are elaborated to a corresponding target expression and at the same time synthesize the corresponding type. This approach seeks to propagate the typing information that we know in the checking rules, and in the synthesis phase, to take advantage of types that we can infer.

Synthesizing:	$\Theta; \Delta; \Gamma \vdash \wr e \; ; \; \theta \int \rightsquigarrow E : T / \Theta'; \rho$
Checking:	$\Theta; \Delta; \Gamma \vdash \wr e \; ; \; \theta \int : T \rightsquigarrow E / \Theta'; \rho$

We first explain the judgment for elaborating a source expression e by

checking it against T given holes in Θ, index variables Δ, and program variables Γ. Because of pattern matching, index variables in Δ may get refined to concrete index terms. Abusing slightly notation, we write θ for the map of free variables occurring in e to their refinements and consider a source expression e together with the refinement map θ, written as $\wr e \; ; \; \theta \int$. The result of elaborating $\wr e \; ; \; \theta \int$ is a target expression E, a new context of holes Θ', and a hole instantiation ρ which instantiates holes in Θ, i.e. $\Theta' \vdash \rho : \Theta$. The result E has type $[\![\rho]\!]T$. It is important to notice here, that ρ contains instances for holes, while θ contains refinements for meta-variables (i.e. instances for meta-variables refined by dependent pattern matching).

The result of elaboration in synthesis mode is similar; we return the target expression E together with its type T, a new context of holes Θ' and a hole instantiation ρ, s.t. $\Theta' \vdash \rho : \Theta$. The result is well-typed, i.e. E has type T.

We give the rules for elaborating source expressions in checking mode in Fig. 3.8 and in synthesis mode in Fig. 3.9. To elaborate a function (see rule `el-fn`) we simply elaborate the body extending the context Γ. There are two cases when we elaborate an expression of dependent function type. In the rule `el-mlam`, we elaborate a dependent function `mlam` $X \Rightarrow e$ against $\Pi^e X{:}U.T$ by elaborating the body e extending the context $\Delta$with the declaration $X{:}U$. In the rule `el-mlam-i`, we elaborate an expression e against $\Pi^i X{:}U.T$ by elaborating e against T extending the context Δ with the declaration $X{:}U$. The result of elaborating e is then wrapped in a dependent function.

When switching to synthesis mode, we elaborate $\wr e \; ; \; \theta \int$ and obtain the corresponding target expression E and type T' together with an instantiation ρ for holes in Θ. We then unify the synthesized type T' and the expected type $[\![\rho]\!]T$ obtaining an instantiation ρ' and return the composition of the instantiation ρ and ρ'. When elaborating an index object $[c]$ (see rule `el-box`), we resort to elaborating c in our indexed language which we assume.

One of the key cases is the one for case-expressions. In the rule `el-case`, we elaborate the scrutinee synthesizing a type S; we then elaborate the branches. Note that we verify that S is a closed type, i.e. it is not allowed to refer to holes. To put it differently, the type of the scrutinee must be fully known. This is done to keep a type refinement in the branches from influencing the type of the scrutinee. The practical impact of this restriction is difficult to quantify, however this seems to be the case for the programs we want to write. A justification is that it is not a problem in any of the examples of the Beluga implementation. For a similar reason, we enforce that the type T, the overall type of the case-expression, is closed; were we to allow holes in T, we would need to reconcile the different instantiations found in different branches.

We omit the special case of pattern matching on index objects to save space and because it is a refinement on the `el-case` rule where we keep the scrutinee when we elaborate a branch. We then unify the scrutinee with the pattern in addition to unifying the type of the scrutinee with the type of the pattern. In the implementation of Beluga, case-expressions on computation-level expressions (which do not need to track the scrutinee and are described in this chapter) and case-expressions on index objects (which do keep the scrutinee when elaborating branches) are handled separately.

When elaborating a constant, we look up its type T_c in the signature Σ and then insert holes for the arguments marked implicit in its type (see

Fig. 3.9). Recall that all implicit arguments are quantified at the outside, i.e. $T_c = \Pi^i X_n : U_n. \ldots \Pi^i X_1 : U_1. S$ where S does not contain any implicit dependent types Π^i. We generate for each implicit declaration $X_k : U_k$ a new hole which can depend on the currently available index variables Δ. When elaborating a variable, we look up its type in Γ and because the variable can correspond to a recursive function with implicit parameters we insert holes for the arguments marked as implicit as in the constant case.

Elaboration of applications in the synthesis mode threads through the hole context and its instantiation, but is otherwise straightforward. In each of the application rules, we elaborate the first argument of the application obtaining a new hole context Θ_1 together with a hole instantiation ρ_1. We then apply the hole instantiation ρ_1 to the context Δ and Γ and to the refinement substitution θ, before elaborating the second part.

$\boxed{\Theta; \Delta \vdash E : T \rightsquigarrow E' : T' / \Theta'}$
Apply E to holes for representing omitted arguments based on T obtaining a term E' of type T'

$$\frac{\textsf{genHole}\ (?Y{:}\Delta \vdash U) = C \quad (\Theta, ?Y : \Delta \vdash U);\ \Delta \vdash E\,[C] : [C/X]T \rightsquigarrow E' : T' \ /\ \Theta'}{\Theta; \Delta \vdash E : \Pi^i X : U.\, T \rightsquigarrow E' : T' \ /\ \Theta'}\ \texttt{el-impl}$$

$$\frac{S \neq \Pi^i X : U.\, T}{\Theta; \Delta \vdash E : S \rightsquigarrow E : S \ /\ \Theta}\ \texttt{el-impl-done}$$

$\boxed{\Theta; \Delta; \Gamma \vdash \wr e \ ;\ \theta \wr \rightsquigarrow E : T / \Theta'; \rho}$ Elaborate source $\wr e\ ;\ \theta \wr$ to target E and synthesize type T

$$\frac{\Gamma(x) = T \quad \Theta; \Delta; \Gamma \vdash x : T \rightsquigarrow E' : T' / \Theta'}{\Theta; \Delta; \Gamma \vdash \wr x \ ;\ \theta \wr \rightsquigarrow E' : T' \ /\ \Theta'; \texttt{id}(\Theta')}\ \texttt{el-var} \qquad \frac{\Sigma(c) = T_c \quad \Theta; \Delta \vdash \mathbf{c} : T_c \rightsquigarrow E : T \ /\ \Theta'}{\Theta; \Delta; \Gamma \vdash \wr \mathbf{c} \ ;\ \theta \wr \rightsquigarrow E : T \ /\ \Theta'; \texttt{id}(\Theta')}\ \texttt{el-const}$$

$$\frac{\Theta; \Delta; \Gamma \vdash \wr e_1 \ ;\ \theta \wr \rightsquigarrow E_1 : S \to T \ /\ \Theta_1; \rho_1 \quad \Theta_1; [\![\rho_1]\!]\Delta; [\![\rho_1]\!]\Gamma \vdash \wr e_2 \ ;\ [\![\rho_1]\!]\theta \wr : [\![\rho_1]\!]S \rightsquigarrow E_2 \ /\ \Theta_2; \rho_2}{\Theta; \Delta; \Gamma \vdash \wr e_1\, e_2 \ ;\ \theta \wr \rightsquigarrow E_1\, E_2 : [\![\rho_2]\!]T \ /\ \Theta_2; \rho_2 \circ \rho_1}\ \texttt{el-app}$$

$$\frac{\Theta; \Delta; \Gamma \vdash \wr e \ ;\ \theta \wr \rightsquigarrow E_1 : \Pi^e X : U.\, T / \Theta_1; \rho_1 \quad \Theta_1; [\![\rho_1]\!]\Delta \vdash \wr c \ ;\ [\![\rho_1]\!]\theta \wr : U \rightsquigarrow C / \Theta_2; \rho_2}{\Theta; \Delta; \Gamma \vdash \wr e\,[c] \ ;\ \theta \wr \rightsquigarrow E_1\,[C] : [C/X]([\![\rho_2]\!]T) / \Theta_2; \rho_2 \circ \rho_1}\ \texttt{el-mapp}$$

$$\frac{\Theta; \Delta; \Gamma \vdash \wr e \ ;\ \theta \wr \rightsquigarrow E : \Pi^e X : U.\, T / \Theta_1; \rho \quad \textsf{genHole}\ (?Y : ([\![\rho]\!]\Delta).U) = C}{\Theta; \Delta; \Gamma \vdash \wr e\,_ \ ;\ \theta \wr \rightsquigarrow E\,[C] \ :\ [C/X]T \ /\ \Theta_1, ?Y : ([\![\rho_1]\!]\Delta).U \ ;\ \rho}\ \texttt{el-mapp-underscore}$$

$$\frac{\Theta; \Delta \vdash \wr t \ ;\ \theta \wr \rightsquigarrow T / \Theta_1; \rho_1 \quad \Theta_1; [\![\rho_1]\!]\Delta; [\![\rho_1]\!]\Delta \vdash \wr e \ ;\ [\![\rho_1]\!]\theta \wr : T \rightsquigarrow E / \Theta_2; \rho_2}{\Theta; \Delta; \Gamma \vdash \wr e : t \ ;\ \theta \wr \rightsquigarrow (E : T) : T / \Theta_2; \rho_2 \circ \rho_1}\ \texttt{el-annotated}$$

Where $\texttt{id}(\Theta)$ returns the identity substitution for context Θ such as:$\Theta \vdash \texttt{id}(\Theta) : \Theta$

Figure 3.9: Elaboration of Expressions (Synthesizing Mode)

3.4.4.1 Elaborating Branches

We give the rules for elaborating branches in Fig. 3.10. Recall that a branch $pat \mapsto e$ consists of the pattern pat and the body e. We elaborate a branch under the refinement θ, because the body e may contain index variables declared earlier and which might have been refined in earlier branches.

Intuitively, to elaborate a branch, we need to elaborate the pattern and synthesize the type of index and pattern variables bound inside of it. In the dependently typed setting, pattern elaboration needs to do however a bit more work: we need to infer implicit arguments which were omitted by the programmer (e.g: the constructor `Ex` takes the type of the expression, and the source of evaluation as implicit parameter `Ex [T] [M]...`) and we need to establish how the synthesized type of the pattern refines the type of the scrutinee.

Moreover, there is a mismatch between the variables the body e may refer to (see rule `wf-branch` in Fig. 3.1) and the context in which the elaborated body E is meaningful (see rule `t-branch` in Fig. 3.5). While our source expression e possibly can refer to index variables declared prior, the elaborated body E is not allowed to refer to any index variables which were declared at the outside; those index variables are replaced by their corresponding refinements. To account for these additional refinements, we not only return an elaborated pattern $\Pi\Delta_r; \Gamma_r.Pat : \theta_r$ when elaborating a pattern pat (see rule `el-subst` in Fig. 3.10), but in addition return a map θ_e between source variables declared explicitly outside and their refinements.

Elaborating a pattern is done in three steps (see rule `el-subst`):

1. First, given pat we elaborate it to a target pattern Pat together with its type S_p synthesizing the type of index variables Δ_p and the type of pattern variables Γ_p together with holes (Θ_p) which denote omitted arguments. This is accomplished by the first premise of the rule

$\boxed{\Delta; \Gamma \vdash \wr b \; ; \; \theta \wr : S \to T \rightsquigarrow B}$ Elaborate source branch $\wr b \; ; \; \theta \wr$ to branch B

$$\frac{\begin{array}{c}\Delta \vdash pat : S \rightsquigarrow \Pi\Delta_r; \Gamma_r.Pat : \theta_r \mid \theta_e \\ \cdot; \Delta_r; [\theta_r]\Gamma, \Gamma_r \vdash \wr e \; ; \; \theta_r \circ \theta, \; \theta_e \wr : [\theta_r]T \rightsquigarrow E / \cdot; \cdot\end{array}}{\Delta; \Gamma \vdash \wr pat \mapsto e \; ; \; \theta \wr : S \to T \rightsquigarrow \Pi\Delta_r; \Gamma_r.Pat : \theta_r \mapsto E} \; \texttt{el-branch}$$

$\boxed{\Delta \vdash pat : T \rightsquigarrow \Pi\Delta_r; \Gamma_r.Pat : \theta_r \mid \theta_e}$

$$\frac{\begin{array}{c}\cdot; \cdot \vdash pat \rightsquigarrow \Pi\Delta_p; \Gamma_p.Pat : S_p / \Theta_p; \cdot \\ \Delta'_p \vdash \epsilon : \Theta_p \\ [\![\epsilon]\!] S_p \doteq S / \Delta_r; \theta_R\end{array}}{\Delta \vdash pat : S \rightsquigarrow \Pi\Delta_r; [\theta_p][\![\epsilon]\!]\Gamma_p.[\theta_p][\![\epsilon]\!]Pat : \theta_r \mid \theta_e} \; \texttt{el-subst}$$

where $\theta_R = \theta_r, \theta_p$ s.t. $\Delta_r \vdash \theta_p : (\Delta'_p, [\![\epsilon]\!]\Delta_p)$ and $\theta_p = \theta_i, \theta_e$ s.t. $\Delta_r \vdash \theta_i : \Delta'_p$

Figure 3.10: Branches and Patterns

`el-subst`:

$$\cdot; \cdot \vdash pat \rightsquigarrow \Pi\Delta_p; \Gamma_p.Pat : S_1 / \Theta_p; \cdot$$

Our pattern elaboration judgment (Figure 3.11) threads through the hole context and the context of index variables, both of which are empty in the beginning. Because program variables occur linearly, we do not thread them through but simply combine program variable contexts when needed. The result of elaborating *pat* is a pattern *Pat* in our target language where Δ_p describes all index variables in *Pat*, Γ_p contains all program variables and Θ_p contains all holes, i.e. most general instantiations of omitted arguments. We describe pattern elaboration in detail in Section 3.4.4.2.

2. Second, we abstract over the hole variables in Θ_p by lifting all holes to fresh index variables from Δ'_p. This is accomplished by

the second premise of the rule `el-subst`using the lifting substitution $\Delta'_p \vdash \epsilon : \Theta_p$.

3. Finally, we compute the refinement substitution θ_r which ensures that the type of the pattern $[\![\epsilon]\!]S_p$ is compatible with the type S of the scrutinee. We note that the type of the scrutinee could also force a refinement of holes in the pattern. This is accomplished by the judgment:

$$\Delta, (\Delta'_p, [\![\epsilon]\!]\Delta_p) \vdash [\![\epsilon]\!]S_1 \doteq T_1/\Delta_r; \theta_R \qquad \theta_R = \theta_r, \theta_p$$

   We note because θ_R maps index variables from $\Delta, (\Delta'_p, [\![\epsilon]\!]\Delta_p)$ to Δ_r, it can be split in two parts: θ_r that provides refinements for variables Δ in the type of the scrutinee; θ_p provides possible refinements of the pattern forced by the scrutinee. This can happen, if the scrutinee's type is more specific than the type of the pattern.

3.4.4.2 Elaborating Patterns

Pattern elaboration is bidirectional. The judgments for elaborating patterns by checking them against a given type and synthesizing their type are:

Synthesizing:	$\Theta; \Delta \vdash pat \quad \rightsquigarrow \Pi\Delta'; \Gamma.Pat : T \; / \; \Theta'; \rho$
Checking:	$\Theta; \Delta \vdash pat : T \rightsquigarrow \Pi\Delta'; \Gamma.Pat \quad / \; \Theta'; \rho$

As mentioned earlier, we thread through a hole context Θ together with the hole substitution ρ that relates: $\Theta' \vdash \rho : \Theta$. Recall that as our examples show index-level variables in patterns need not to be linear and hence we accumulate index variables and thread them through as well. Program variables on the other hand must occur linearly, and we can simply combine them. The elaboration rules are presented in Figure 3.11. In synthesis mode, elaboration returns a reconstructed pattern Pat, a type T where Δ' describes the index variables in Pat and Γ' contains all program

variables occurring in *Pat*. The hole context Θ' describes the most general instantiations for omitted arguments which have been inserted into *Pat*. In checking mode, we elaborate *pat* given a type T to the target expression *Pat* and index variable context Δ', pattern variable context Γ' and the hole context Θ'.

Pattern elaboration starts in synthesis mode, i.e. either elaborating an annotated pattern $(e : t)$ (see rule `el-pann`) or a pattern $\mathbf{c}\ \overrightarrow{pat}$ (see rule `el-pcon`). To reconstruct patterns that start with a constructor we first look-up the constructor in the signature Σ to get its fully elaborated type T_c and then elaborate the arguments $\overrightarrow{pat}$ against T_c. Elaborating the spine of arguments is guided by the type T_c. If $T_c = \Pi^i X{:}U.\, T$, then we generate a new hole for the omitted argument of type U. If $T_c = T_1 \to T_2$, then we elaborate the first argument in the spine $pat\ \overrightarrow{pat}$ against T_1 and the remaining arguments $\overrightarrow{pat}$ against T_2. If $T_c = \Pi^e X{:}U.\, T$, then we elaborate the first argument in the spine $[c]\ \overrightarrow{pat}$ against U and the remaining arguments $\overrightarrow{pat}$ against $[C/X]T$. When the spine is empty, denoted by $\cdot$, we simply return the final type and check that the constructor was fully applied by ensuring that the type S we reconstruct against is either of index level type, i.e. $[U]$, or a recursive type, i.e. $\mathbf{a}[\overrightarrow{C}]$.

For synthesizing the patterns with a type annotation, first we elaborate the type t in an empty context using a judgment that returns the reconstructed type T, its holes and index variables (contexts Θ' and Δ'). Once we have the type we elaborate the pattern checking against the type T.

To be able to synthesize the type of pattern variables and return it, we check variables against a given type T during elaboration (see rule `el-pvar`). For index level objects, rule `el-pindex` we defer to the index level elaboration that the index domain provides[3]. Finally, when elabo-

[3]Both, elaboration of pattern variables and of index objects can be generalized by for example generating a type skeleton in the rule `el-subst` given the scrutinee's type. This is in fact what is done in the implementation of Beluga.

Pattern (synthesis mode) $\boxed{\Theta; \Delta \vdash pat \rightsquigarrow \Pi\Delta'; \Gamma.Pat : T \;/\; \Theta' \;;\; \rho}$

$$\frac{\Sigma(c) = T \quad \Theta; \Delta \vdash \overrightarrow{pat} : T \rightsquigarrow \Pi\Delta'; \Gamma.\overrightarrow{Pat} \rangle S \;/\; \Theta'; \rho}{\Theta; \Delta \vdash \mathbf{c}\,\overrightarrow{pat} \rightsquigarrow \Pi\Delta'; \Gamma.\mathbf{c}\,\overrightarrow{Pat} : S \;/\; \Theta'; \rho}\ \texttt{el-pcon}$$

$$\frac{\cdot; \cdot \vdash \wr t \;;\; \cdot \int \rightsquigarrow T / \Theta'; \Delta'; \cdot \quad (\Theta, \Theta') \;;\; (\Delta, \Delta') \vdash pat : T \rightsquigarrow \Pi\Delta''; \Gamma.Pat \;/\; \Theta''; \rho'}{\Theta; \Delta \vdash (pat : t) \rightsquigarrow \Pi\Delta''; \Gamma.Pat : [\![\rho']\!]T \;/\; \Theta''; \rho'}\ \texttt{el-pann}$$

Pattern (checking mode) $\boxed{\Theta; \Delta \vdash pat : T \rightsquigarrow \Pi\Delta'; \Gamma.Pat \;/\; \Theta'; \rho}$

$$\frac{}{\Theta; \Delta \vdash x : T \rightsquigarrow \Pi\Delta \;;\; x{:}T.x \;/\; \Theta; \mathtt{id}(\Theta)}\ \texttt{el-pvar} \qquad \frac{\Theta; \Delta \vdash c : U \rightsquigarrow C / \Theta'; \Delta'; \rho}{\Theta; \Delta \vdash [c] : [U] \rightsquigarrow \Pi\Delta'; \cdot\,.\,[C] / \Theta' \;;\; \rho}\ \texttt{el-pindex}$$

$$\frac{\Theta; \Delta \vdash pat \rightsquigarrow \Pi\Delta'; \Gamma.Pat : S \;/\; \Theta'; \rho \qquad \Theta'; \Delta' \vdash S \doteq [\![\rho]\!]T \;/\; \Theta''; \rho'}{\Theta; \Delta \vdash pat : T \rightsquigarrow \Pi[\![\rho']\!]\Delta'; [\![\rho']\!]\Gamma\,.\,[\![\rho]\!]Pat \;/\; \Theta'' \;;\; \rho' \circ \rho}\ \texttt{el-psyn}$$

Pattern Spines $\boxed{\Theta; \Delta \vdash \overrightarrow{pat} : T \rightsquigarrow \Pi\Delta'; \Gamma.\overrightarrow{Pat} \rangle S \;/\; \Theta'; \rho}$

$$\frac{\text{either } T = [U] \text{ or } T = \mathbf{a}\,\overrightarrow{[C]}}{\Theta; \Delta \vdash \cdot : T \rightsquigarrow \Pi\Delta; \cdot\,.\,\cdot \rangle T \;/\; \Theta; \mathtt{id}(\Theta)}\ \texttt{el-sp-empty}$$

$$\frac{\Theta; \Delta \vdash pat : T_1 \rightsquigarrow \Pi\Delta'; \Gamma.Pat / \Theta'; \rho \qquad \Theta'; \Delta' \vdash \overrightarrow{pat} : [\![\rho]\!]T_2 \rightsquigarrow \Pi\Delta''; \Gamma'.\overrightarrow{Pat} \rangle S \;/\; \Theta''; \rho'}{\Theta; \Delta \vdash pat\ \ \overrightarrow{pat} : T_1 \to T_2 \rightsquigarrow \Pi\Delta''; (\Gamma, \Gamma')\,.\,([\![\rho']\!]Pat)\ \ \overrightarrow{Pat} \rangle S \;/\; \Theta''; \rho' \circ \rho}\ \texttt{el-sp-cmp}$$

$$\frac{\Theta; \Delta \vdash c : U \rightsquigarrow C / \Theta'; \Delta'; \rho \quad \Theta'; \Delta' \vdash \overrightarrow{pat} : [C/X][\![\rho]\!]T \rightsquigarrow \Pi\Delta''; \Gamma.\overrightarrow{Pat} \rangle S \;/\; \Theta''; \rho'}{\Theta; \Delta \vdash [c]\,\overrightarrow{pat} : \Pi^e X{:}U.\,T \rightsquigarrow \Pi\Delta''; \Gamma\,.\,([\![\rho']\!][C])\,\overrightarrow{Pat} \rangle S \;/\; \Theta''; \rho' \circ \rho}\ \texttt{el-sp-explicit}$$

$$\frac{\mathsf{genHole}\ (?Y{:}\Delta.U) = C \qquad \Theta, ?Y{:}\Delta.U; \Delta \vdash \overrightarrow{pat} : [C/X]T \rightsquigarrow \Pi\Delta'; \Gamma.\overrightarrow{Pat} \rangle S \;/\; \Theta'; \rho}{\Theta; \Delta \vdash \overrightarrow{pat} : \Pi^i X{:}U.\,T \rightsquigarrow \Pi\Delta'; \Gamma.([\![\rho]\!]C)\ \ \overrightarrow{Pat} \rangle S \;/\; \Theta'; \rho}\ \texttt{el-sp-implicit}$$

Figure 3.11: Elaboration of Patterns and Pattern Spines

rating a pattern against a given type it is possible to switch to synthesis mode using rule `el-psyn`, where first we elaborate the pattern synthesizing its type S and then we make sure that S unifies against the type T it should check against.

3.5 Soundness of Elaboration

We establish soundness of our elaboration: if from a well-formed source expression, we obtain a well-typed target expression E which may still contain some holes then E is well-typed for any ground instantiation of these holes. In fact, our final result of elaborating a recursive function and branches must always return a closed expression.

Theorem 3.5.1 (Soundness)**.**

1. If $\Theta; \Delta; \Gamma \vdash \wr e \; ; \; \theta \int \; : \; T \rightsquigarrow E / \Theta_1; \rho_1$ then for any grounding hole instantiation ρ_g s.t. $\cdot \vdash \rho_g : \Theta_1$ and $\rho_0 = \rho_g \circ \rho_1$, we have: $[\![\rho_0]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]E \Leftarrow [\![\rho_0]\!]T$.

2. If $\Theta; \Delta; \Gamma \vdash \wr e \; ; \; \theta \int \rightsquigarrow E : T / \Theta_1; \rho_1$ then for any grounding hole instantiation ρ_g s.t. $\cdot \vdash \rho_g : \Theta_1$ and $\rho_0 = \rho_g \circ \rho_1$, we have: $[\![\rho_0]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]E \Rightarrow [\![\rho_g]\!]T$.

3. If $\Delta; \Gamma \vdash \wr pat \; \mapsto \; e \; ; \; \theta \int \; : \; S \to T \rightsquigarrow \Pi\Delta'; \Gamma'.Pat \; : \; \theta' \; \mapsto \; E$ then $\Delta; \Gamma \vdash \Pi\Delta'; \Gamma'.Pat : \theta' \mapsto E \Leftarrow S \to T$.

To establish the soundness of elaboration of case-expressions and branches, we rely on pattern elaboration which abstracts over the variables in patterns as well as over the holes which derive from the instantiations inferred for omitted arguments. We abstract over these holes using a lifting substitution ϵ. The proofs for this theorem and related lemmas are in Appendix A).

Lemma 1 (Pattern elaboration)**.**

1. If $\Theta; \Delta \vdash pat \rightsquigarrow \Pi\Delta_1; \Gamma_1.Pat : T/\Theta_1; \rho_1$ and ϵ is a ground lifting substitution, such as $\Delta_i \vdash \epsilon : \Theta_1$ then:
   $\Delta_i, [\![\epsilon]\!]\Delta_1; [\![\epsilon]\!]\Gamma_1 \vdash [\![\epsilon]\!]Pat \Leftarrow [\![\epsilon]\!]T$.

2. If $\Theta; \Delta \vdash pat : T \rightsquigarrow \Pi\Delta_1; \Gamma_1.Pat/\Theta_1; \rho_1$ and ϵ is a ground lifting substitution, such as $\Delta_i \vdash \epsilon : \Theta_1$ then:
   $\Delta_i, [\![\epsilon]\!]\Delta_1; [\![\epsilon]\!]\Gamma_1 \vdash [\![\epsilon]\!]Pat \Leftarrow [\![\epsilon]\!][\![\rho_1]\!]T$.

3. If $\Theta; \Delta \vdash \overrightarrow{pat} : T \rightsquigarrow \Pi\Delta_1; \Gamma_1.\overrightarrow{Pat} \rangle S/\Theta_1; \rho_1$ and ϵ is a ground lifting substitution, such as $\Delta_i \vdash \epsilon : \Theta_1$ then:
   $\Delta_i, [\![\epsilon]\!]\Delta_1; [\![\epsilon]\!]\Gamma_1 \vdash [\![\epsilon]\!]\overrightarrow{Pat} \Leftarrow [\![\epsilon]\!][\![\rho_1]\!]T \rangle [\![\epsilon]\!]S$.

3.6 Related Work

Our language contains indexed families of types that are related to Zenger's work [Zenger, 1997] and the Dependent ML (DML) [Xi, 2007] and Applied Type System (ATS) [Xi, 2004, Chen and Xi, 2005]. The objective in these systems is: a program that is typable in the extended indexed type system is already typable in ML. By essentially erasing all the type annotations necessary for verifying the given program is dependently typed, we obtain a simply typed ML-like program. In contrast, our language supports pattern matching on index objects. Our elaboration, in contrast to the one given in Xi [2007], inserts omitted arguments producing programs in a fully explicit dependently typed core language. This is different from DML-like systems which treat *all* index arguments as implicit and do not provide a way for programmers to manipulate and pattern match directly on index objects. Allowing users to explicitly access and match on index arguments changes the game substantially.

Elaboration from implicit to explicit syntax for dependently typed systems has first been mentioned by Pollack [1990] although no concrete

algorithm to reconstruct omitted arguments was given. Luther [2001] refined these ideas as part of the TYPELab project. He describes an elaboration and reconstruction for the calculus of constructions without treating recursive functions and pattern matching. There is in fact little work on elaborating dependently-typed source language supporting recursion and pattern matching. For example, the Agda bi-directional type inference algorithm described in Norell [2007] concentrates on a core dependently typed calculus enriched with dependent pairs, but omits the rules for its extension with recursion and pattern matching[4]. Idris, a dependently typed language developed by Brady [2013] uses a different technique. Idris starts by adding holes for all the implicit variables and it tries to instantiate these holes using unification. However, the language uses internally a tactic based elaborator that is exposed to the user who can interactively fill the holes using tactics. He does not prove soundness of the elaboration, but conjectures that given a type correct program its elaboration followed by a reverse elaboration produces a matching source level program.

A notable example, is the work by Asperti et al. [2012] on describing a bi-directional elaboration algorithm for the Calculus of (Co)Inductive Constructions (CCIC) implemented in Matita. Their setting is very different from ours: CCIC is more powerful than our language since the language of recursive programs can occur in types and there is no distinction between the index language and the programming language itself. Moreover in Matita, we are only allowed to write total programs and all types must be positive. For these reasons their source and target language is more verbose than ours and refinement, i.e. the translation of the source to the target, is much more complex than our elaboration. The difference between our language and Matita particularly comes to light when writing case-expressions. In Matita as in Coq, the programmer

[4]Norell [2007] contains extensive discussions on pattern matching and recursion, but the chapter on elaboration does not discuss them.

might need to supply an invariant for the scrutinee and the overall type of the case expression as a type annotation. Each branch then is checked against the type given in the invariant. Sometimes, these annotations can be inferred by using higher-order unification to find the invariant. In contrast, our case-expressions require no type annotations and we refine each branch according to refinement imposed by the pattern in each branch. The refinement is computed with help from higher-order unification. This makes our source and target language more light-weight and closer to a standard simply typed functional language.

Finally, refinement in Matita may leave some holes in the final program which then can be refined further by the user using for example tactics. We support no such interaction; in fact, we fail, if holes are left-over and the programmer is asked to provide more information.

Agda, Matita and Coq require users to abstract over all variables occurring in a type and the user statically labels arguments the user can freely omit. To ease the requirement of declaring all variables occurring in type, many of these systems such as Agda supports simply listing the variables occurring in a declaration without the type. This however can be brittle since it requires that the user chose the right order. Moreover, the user has the possibility to locally override the implicit arguments mechanism and provide instantiations for implicit arguments explicitly. This is in contrast to our approach where we guide elaboration using type annotations and omit arguments based on the free variables occurring in the declared type, similarly to Idris which abstracts and makes implicit all the free variables in types.

This work is also related to type inference for Generalized Algebraic Data Types (i.e: GADTs) such as [Schrijvers et al., 2009]. Here the authors describe an algorithm where they try to infer the types of programs with GADTs when the principal type can be inferred and requiring type annotations for the cases that lack a principal type or it can not be inferred.

This is in contrast to our system which always requires a type annotation at the function level. On the other hand our system supports a richer variety of index languages (some index languages can be themselves dependently typed as with Contextual LF in Beluga). Moreover we support pattern matching on index terms, a feature that is critical to enable reasoning about objects from the index level. Having said that, the approach to GADTs from [Schrijvers et al., 2009] offers interesting ideas for future work, first making the type annotations optional for cases when they can be inferred, and providing a declarative type systems that helps the programmer understand when will the elaboration succeed to infer the types.

3.7 Conclusion

In this chapter we describe a surface language for writing dependently typed programs where we separate the language of types and index objects from the language of programs. Total programs in our language correspond to first-order inductive proofs over a specific index domain where we mediate between the logical aspects and the domain-specific parts using a box modality. Our programming language supports indexed data-types, dependent pattern matching and recursion. Programmers can leave index variables free when declaring the type of a constructor or recursive program as a way of stating that arguments for these free variables should be inferred by the type-directed elaboration. This offers a lightweight mechanism for writing compact programs which resemble their ML counterparts and information pertaining to index arguments can be omitted. In particular, our handling of case-expressions does not require programmers to specify the type invariants the patterns and their bodies must satisfy and case expressions can be nested due to the refinement substitutions that mediate between the context inside and outside

a branch. Moreover, we seamlessly support nested pattern matching inside functions in our surface and core languages (as opposed to languages such as Agda or Idris where the former supports pattern matching lambdas that are elaborated as top-level functions and the latter only supports simply typed nested pattern matching).

The proposed case-expression can be nested and does not require annotating its return type. Notably, it refines all the variables in the context, this allows the programmer to write pattern matching on dependent types using variables from the context that have been refined by the current branch. We think this is a powerful feature for the users of our language, nonetheless it is at the expense of some complexity in the theory of the language, namely the elaboration of terms together with a substitution that contains the current refinements.

To guide elaboration and type inference, we allow type annotations which indirectly refine the type of sub-expressions; type annotations in patterns are also convenient to name index variables which do not occur explicitly in a pattern.

We prove our elaboration sound, in the sense that if elaboration produces a fully explicit term, this term will be well-typed. Finally, our elaboration is implemented in Beluga, where we use as the index domain contextual LF, and has been shown practical (see for example the implementation of a type-preserving compiler [Belanger et al., 2013]). We believe our language presents an interesting point in the design space for dependently typed languages in general and sheds light into how to design and implement a dependently typed language where we have a separate index language, but still want to support pattern matching on these indices.

4 Contextual Types and Programming Languages

4.1 Introduction

Writing programs that manipulate other programs is a common activity for a computer scientist, either when implementing interpreters, writing compilers, or analyzing phases for static analysis. This is so common that we have programming languages that specialize in writing these kinds of programs. In particular, ML-like languages are well-suited for this task thanks to recursive data types and pattern matching. However, when we define syntax trees for realistic input languages, there are more things on our wish list: we would like support for representing and manipulating variables and tracking their scope; we want to compare terms up-to α-equivalence (i.e. the renaming of bound variables); we would like to avoid implementing capture avoiding substitutions, which is tedious and error-prone. ML languages typically offer no high-level abstractions or support for manipulating variables and the associated operations on abstract syntax trees.

Over the past decade, there have been several proposals to add support for defining and manipulating syntax trees into existing programming environments. For example: FreshML [Shinwell et al., 2003], the related system Romeo [Stansifer and Wand, 2014], and Cαml [Pottier, 2006] use Nominal Logic [Pitts, 2003] as a basis and the Hobbits library for Haskell [Westbrook et al., 2011] uses a name based formalism. In this chapter, we show how to extend an existing (functional) programming language to define abstract syntax trees with variable binders based on higher-order abstract syntax (HOAS) (sometimes also called λ-trees [Miller and Palamidessi, 1999]). Specifically, we allow programmers

to define object languages in the simply-typed λ-calculus where programmers use the intentional function space of the simply typed λ-calculus to define binders (as opposed to the extensional function space of ML). Hence, HOAS representations inherit α-renaming from the simply-typed λ-calculus and we can model object-level substitution for HOAS trees using β-reduction in the underlying simply-typed λ-calculus. We further allow programmers to express whether a given sub-tree in the HOAS tree is closed by using the necessity modality of S4 [Davies and Pfenning, 2001]. This additional expressiveness is convenient to describe closed abstract syntax trees.

This work follows the work of HOAS representations in the logical framework LF [Harper et al., 1993]. On the one hand we restrict it to the simply-typed setting to integrate it smoothly into existing simply-typed functional programming languages such as OCaml, and on the other hand we extend its expressiveness by allowing programmers to distinguish between closed and open parts of their syntax trees. As we analyze HOAS trees, we go under binders and our sub-trees may not remain closed. To model the scope of binders in sub-trees we pair a HOAS tree together with its surrounding context of variables following ideas from Beluga [Pientka, 2008, Nanevski et al., 2008]. In addition, we allow programmers to pattern match on such contextual objects, i.e. an HOAS tree together with its surrounding context. In essence, this work shows that the fundamental ideas underlying Beluga [Pientka and Dunfield, 2010b, Pientka and Cave, 2015] can be smoothly integrated into existing programming environments.

In this chapter the contribution is two-fold: First, we present a general methodology for adding support for HOAS tree definitions and first-class contexts to an existing (simply-typed) programming language. In particular, programmers can define simply-typed HOAS definitions in the syntactic framework (SF) based on modal S4 following [Nanevski et al.,

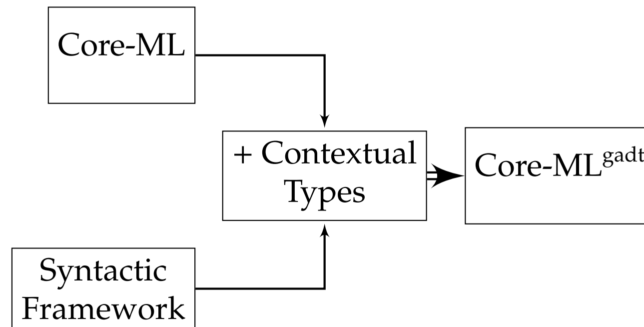

Figure 4.1: Adding Contextual Types to ML

2008, Davies and Pfenning, 2001]. In addition, programmers can manipulate and pattern match on well-scoped HOAS trees by embedding HOAS objects together with their surrounding context into the programming language using contextual types [Pientka, 2008]. The result is a programming language that can express computations over open HOAS objects. We describe our technique abstractly and generically using a language that we call Core-ML. In particular, we show how Core-ML with first-class support for HOAS definitions and contexts can be translated in into a language Core-ML^{gadt} that supports Generalized Abstract Data Types (GADTs) using a deep (first-order) embedding of SF and first-class contexts (see Fig. 4.1 for an overview). We further show that our translation preserves types.

Second, we show how this methodology can be realized in OCaml by describing our prototype Babybel[1]. In our implementation of Babybel we take advantage of the sophisticated type system, in particular GADTs, that OCaml provides to ensure our translation is type-preserving. By translating HOAS objects together with their context to a first-order representation in OCaml with GADTs we can also reuse OCaml's first-order pattern matching compilation allowing for a straightforward compilation. Programmers can also exploit OCaml's impure features such as

[1]available at `www.github.com/fferreira/babybel/`

exceptions or references when implementing programs that manipulate HOAS syntax trees. The Babybel prototype includes implementations of a type-checker, an evaluator, closure conversion (shown in Section 4.2.1.3 together with a variable counting example and a syntax desugaring examples) and a continuation passing style translation. These examples demonstrate that this approach allows programmers to write programs that operate over abstract syntax trees in a manner that is safe and effective.

Finally, notice that Babybel's language extensions do not have to be OCaml specific. The same approach could be implemented in Haskell, and other typed functional programming languages.

4.2 Main Ideas

In this section, we show some examples that illustrate the use of Babybel, our proof of concept implementation where we embed the syntactic framework SF inside OCaml. To smoothly integrate SF into OCaml, Babybel defines a PPX filter (a mechanism for small syntax extensions for OCaml). In particular, we use attributes and quoted strings to implement our syntax extension.

4.2.1 Example Programs

4.2.1.1 Removing Syntactic Sugar

In this example, we describe the compact and elegant implementation of a compiler phase that de-sugars programs functional programs with let-expressions by translating them into function applications. We first specify the syntax of a simple functional language that we will transform. To do this we embed the syntax specification using this tag:

```
[@@@signature {def| ... |def}]
```

Inside the **@@@signature** block we will embed our SF specifications.

Our source language is defined using the type `tm`. It consists of constants (written as `cst`), pairs (written as `pair`), functions (built using `lam`), applications (built using `app`), and let-expressions.

```
[@@@signature {def|
tm : type.
cst     : tm.
pair    : tm → tm → tm.
lam     : (tm → tm) → tm.
fst     : tm → tm.
snd     : tm → tm.
letpair : tm → (tm → tm → tm) → tm.
letv    :  tm → (tm → tm) → tm.
app     : tm → tm → tm.
|def}]
```

Our definition of the source language exploits HOAS using the function space of our syntactic framework SF to represent binders in our object language. For example, the constructor `lam` takes as an argument a term of type `tm → tm`. Similarly, the definition of let-expressions models variable binding by falling back to the function space of our meta-language, in our case the syntactic framework SF. As a consequence, there is no

constructor for variables in our syntactic definition and moreover we can reuse the substitution operation from the syntactic framework SF to model substitution in our object language. This avoids building up our own infrastructure for variables bindings.

We now show how to simplify programs written in our source language by replacing uses of `letpair` in terms with projections, and uses of `letv` by β reduction.

$$\texttt{letv}\, M\ (\lambda \texttt{x}.N) \equiv \texttt{N}[M/\texttt{x}]$$
$$\texttt{letpair}\, M\ (\lambda \texttt{x}.\lambda \texttt{y}.\ N) \equiv \texttt{N}[(\texttt{fst}\, M\ )/\texttt{x},(\texttt{snd}\, M)/\texttt{y}]$$

To implement this simplification phase we implement an OCaml program `rewrite`: it analyzes the structure of our terms, calls itself on the sub-terms, and eliminates the use of the let-expressions into simpler constructs. As we traverse terms, our sub-terms may not remain closed. For simplicity, we use the same language as source and target for our transformation. We therefore specify the type of the function `rewrite` using contextual types pairing the type `tm` together with a context γ in which the term is meaningful inside the tag `[@type .... ]`.

```
rewrite[@type γ.[γ ⊢ tm]→[γ ⊢ tm]]
```

The type can be read: for all contexts γ, given a `tm` object in the context γ, we return a `tm` object in the same context. In general, contextual types associate a context and a type in the syntactic framework SF. For example if we want to specify a term in the empty context we would write `[ ⊢ tm]` or for a term that depends on some context with at least one variable and potentially more we would write `[γ,x:tm ⊢ tm]`.

We now implement the function `rewrite` by pattern matching on the structure of a contextual term. In Babybel, contextual terms are written inside boxes (⦇...⦈) and contextual patterns inside ($⦇\ldots⦈_p$).

```
let rec rewrite[@type γ.[γ ⊢ tm]→[γ ⊢ tm]]
= function
| ⦇ cst ⦈p → ⦇cst⦈
| ⦇ pair 'm 'n ⦈p →
    let mm, nn = rewrite m, rewrite n
    in ⦇pair 'mm 'nn⦈
| ⦇ fst 'm ⦈p →
    let mm = rewrite m in ⦇fst 'mm⦈
| ⦇ snd 'm ⦈p →
    let mm = rewrite m in ⦇snd 'mm⦈
| ⦇ app 'm 'n ⦈p →
    let mm,nn = rewrite m, rewrite n
    in ⦇app 'mm 'nn⦈
| ⦇ lam (λx. 'm) ⦈p →
    let mm = rewrite m in ⦇lam (λx. 'mm)⦈
| ⦇ #x ⦈p → ⦇#x⦈
| ⦇ letpair 'm (λf.λs. 'n) ⦈p →
      let mm = rewrite m in
      rewrite ⦇'n [snd 'mm;fst 'mm]⦈
| ⦇ letv 'm (λx. 'n) ⦈p → rewrite ⦇'n['m]⦈
```

Note that we are pattern matching on potentially open terms. Although we do not write the context γ explicitly, in general patterns may mention their context (i.e.: $⦇$`_ ⊢ cst`$⦈_p$[2]. As a guiding principle, we may omit writing contexts, if they do not mention variables explicitly and are irrel-

[2]The underscore means that there might be a context but we do not bind any variable for it because the term does not explicitly mention them and contexts are not available at run-time.

evant at run-time. Inside patterns or terms, we specify incomplete terms using quoted variables (e.g.: `'n`). Quoted variables are an 'unboxing' of a computational expression inside the syntactic framework SF. The quote signals that we are mentioning a computational variable inside SF. Quoted variables can depend on all the variables in scope in scope where they are defined. For example, in the pattern:

$(\!|$ `letpair 'm (λf.λs. 'n)` $|\!)_p$

the variable `'n` may depend on bound variables `f` and `s`.

The computationally interesting cases are the let-expressions. For them, we perform the rewriting according to the two rules given earlier. The syntax of the substitutions puts in square brackets the terms that will be substituted for the variables. We consider contexts and substitutions ordered, this allows for efficient implementations and more lightweight syntax (e.g.: substitutions omit the name of the variables because contexts are ordered). Importantly, the substitution is an operation that is eagerly applied and not part of the representation. Consequently, the representation of the terms remains normal and substitutions cannot be written in patterns. We come back to this design decision later. When pattern matching on open terms, one needs to match against variables, this is accomplished with the pattern for variables from the context that we denote as $(\!|$ `#x` $|\!)_p$.

To translate contextual SF objects and contexts, Babybel takes advantage of OCaml's advanced type system. In particular, we use Generalized Abstract Data Types [Cheney and Hinze, 2003a, Xi et al., 2003] to index types with the contexts in which they are valid. Type indices, in particular contexts, are then erased at run-time. When the contexts are relevant at run-time, we need to provide a term to explicitly represent the contexts. In Section 4.2.1.3 there is an example of this.

4.2.1.2 Finding the Path to a Variable

In this example, we compute the path to a specific variable in an abstract syntax tree describing a lambda-term. This will show how to specify particular context shapes, how to pattern match on variables, how to manage our contexts, and how the Babybel extensions interact seamlessly with OCaml's impure features. For this example, we concentrate on the fragment of terms that consists only of abstractions and application which we repeat here.

```
[@@@signature {def|
tm : type.
app : tm → tm → tm.
lam : (tm → tm) → tm.
|def}]
```

To find the first occurrence of a particular variable in the HOAS tree, we use backtracking that we implement using the user-defined OCaml exception `Not_found`. To model the path to a particular variable occurrence in the HOAS tree, we define an OCaml data type `step` that describes the individual steps we take and finally model a `path` as a list of individual steps.

```
exception Not_found
type step
= Here  (*the path ends here*)
| AppL  (*take left on app*)
| AppR  (*take right on app*)
| InLam (*go inside the body of the term*)
type path = step list
```

The main function `path_aux` takes as input a term that lives in a context with at least one variable and returns a path to the occurrence of

the top-most variable or an empty list, if the variable is not used. Its type is:

```
[@type γ. [γ, x : tm ⊢ tm] → path].
```

We again quantify over all contexts γ and require that the input term is meaningful in a context with at least one variable. This specification simply excludes closed terms since there would be no top-most variable. Note also how we mix in the type annotation to this function both contextual types and OCaml data types.

```
let rec path_aux [@type γ.[γ, x:tm ⊢ tm] → path]
= function
| ⦇_, x ⊢ x⦈p→ [Here]
| ⦇_, x ⊢ #y⦈p→ raise Not_found
| ⦇_, x ⊢ lam (λy. 'm)⦈p→
      InLam::(path_aux⦇_,x,y ⊢'m[_;y;x]⦈)
| ⦇_, x ⊢ app 'm 'n⦈p→
      try AppL::(path_aux m)
      with _ → AppR::(path_aux n)
```

All patterns in this example make the context explicit, as we pattern match on the context to identify whether the variable we encounter refers to the top-most variable declaration in the context. The underscore simply indicates that there might be more variables in the context. The first case, matches against the bound variable x. As mentioned before, the second case has a special pattern with the sharp symbol that matches against any variable in the context _, x. Because of the first pattern if it had been x it would have matched the first case. Therefore, it simply raises the exception to backtrack to the last choice we had. The case for abstractions is interesting, since we have to go under its binder and the variable that we are looking for is no longer the top most declaration of the context. Hence we must apply a substitution to swap the two variables at the top

of the context.

The case for lambda expressions is interesting because the recursive call happens in an extended context. Furthermore, in order to keep the variable we are searching for on top, we need to swap the two top-most variables. For that purpose, we apply the `[_ ; y; x]` substitution. In this substitution the underscore stands for the identity on the rest of the context, or more precisely, the appropriate shift in our internal representation that uses de Bruijn indices. Once elaborated, this substitution becomes `[^2 ; y; x]` where the shift by two is because we are swapping variables as opposed to instantiating them with closed terms.

The final case is for applications. We first look on the left side and if that raises an exception we catch it and search again on the right. We again use quoted variables (e.g.: `'m`) to bind and refer to ML variables in patterns and terms of the syntactic framework and more generally be able to describe incomplete terms.

```
let get_path [@type γ.[γ, x:tm ⊢ tm] → path]
= fun t → try path_aux t with _ → []
```

The `get_path` function has the same type as the `path_aux` function. It simply handles the exception and returns an empty path in case that variable `x` is not found in the term.

4.2.1.3 Closure Conversion

In the final example, we describe the implementation of a naive algorithm for closure conversion for untyped λ-terms following [Cave and Pientka, 2012]. We take advantage of the syntactic framework SF to represent source terms (using the type family `tm`) and closure-converted terms (using the type family `ctm`). In particular, we use SF's closed modality box to ensure that all functions in the target language are closed (this is written with curly braces: `{}`). This is impossible when we simply use LF as the specification framework for syntax as in [Cave and Pientka, 2012].

```
[@@@signature {def|
ctm: type.  % closed term
btm: type.  % binder term
env : type. % environment

capp   : ctm → ctm → ctm.
clam   : {btm} → ctm.
clo    : ctm → env → ctm.

embed  : ctm → btm.
bind   : (ctm → btm) → btm.

empty  : env.
dot    : env → ctm → env.
|def}]
```

Figure 4.2: Closure Converted Language

We omit here the definition of lambda-terms, our source language, that was given in the previous section and concentrate on the target language `ctm`.

Concretely, `tm` is the type of our input language. Applications are again represented by the constructor `app` that takes two terms, the first represents the function and the second its parameter.

One option would be to simply use the same representation for the target language that we used for the source language (i.e.: the untyped λ-calculus). A better option is to use a new representation that highlights that functions in the target language do no depend on their environments. To this effect we declare in Figure 4.2 two types, `ctm` to represent terms after the conversion and `btm` to represent the bodies of functions. We take advantage of the expressive power of the specification framework SF to define the closed bodies of the functions `btm` and the converted terms `ctm`.

Applications in the target language are defined using the constructor `capp` and simply take two target terms to form an application. But functions (constructor `clam`) take a `btm` object wrapped in `{}` braces. This means that the object inside the braces is closed. The curly braces denote the internal closed modality of the syntactic framework. As the original functions may depend on variables in the environment, we need closures where we pair a function with an environment that provides the appropriate instances for variables. We define our own environment explicitly, because they are part of the target language and the built-in substitution is an operation on terms that is eagerly computed away. Inside the body of the function, we need to bind all the variables from the environment that the body uses such that later we can instantiate them applying the substitution. This is achieved by defining multiple bindings using constructors `bind` and `embed` inside the term.

When writing a function that translates between representations, the open terms depend on contexts that store assumptions of different representations. Therefore, it is often the case that one needs to relate these contexts. In our example here we define a context relation that keeps the input and output contexts in sync using a GADT data type `rel` in OCaml where we model contexts as types. The relation statically checks correspondence between contexts, but it is also available at run-time (i.e. after type-erasure). It states that for each variable in the source contexts there is a corresponding one in the target context.

```
type (_ , _) rel  =
    Empty : ([.], [.]) rel
  | Both  : ([γ], [δ]) rel →
      ([γ, x:tm], [δ, y:ctm]) rel

exception Error of string

let rec lookup
  [@type γ δ.[γ ⊢ tm]→(γ, δ) rel→[δ ⊢ ctm]] =
fun t → function
| Both r’ → begin match t with
  | ⦇ _,x ⊢ x ⦈p → ⦇_,x ⊢ x⦈
  | ⦇ _,x ⊢ ##v ⦈p → let v1 = lookup ⦇#v⦈ r’
                       in ⦇_, x ⊢ ’v1 [_]⦈
  | _ → raise Error (‘‘Term that is not a variable
   ’’)
| Empty → raise Error (‘‘Term is not a variable’’)
end
```

The function `lookup` searches for related variables in the context relation. If we have a source context γ`,x:tm` and a target context δ`,y:ctm`, then we consider two variable cases: In the first case, we use matching to check that we are indeed looking for the top-most variable `x` and we simply return the corresponding target variable. If we encounter a variable from the context, written as `##v`, then we recurse in the smaller context stripping off the variable declaration `x`. Note that `##v` denotes a variable from the context `_`, that is not `x`, while `#v` describes a variable from the context `_, x`, i.e. it could be also `x`. The recursive call returns the corresponding variable `v1` in the target context that does not include the variable declaration `x`. We hence need to weaken `v1` to ensure it is meaningful in the original context. We therefore associate `’v1` with the

identity substitution for the appropriate context, namely: `[_]`. In this case, it will be elaborated into a one variable shift in the internal syntax (i.e.: $\uparrow^1$). The last case returns an exception whenever we are trying to look up in the context something that is not a variable.

As we cannot express at the moment in the type annotation that the input to the lookup function is indeed only a variable from the context γ and not an arbitrary term, we added another fall-through case for when the context is empty. In this case the input term cannot be a variable, as it would be out of scope.

Finally, we implement the function `conv` which takes an untyped source term in a context γ and a relation of source and target variables, described by `(`γ`, `δ`) rel` and returns the corresponding target term in the target context δ.

```
let rec close [@type γ δ. (γ, δ) rel→[δ ⊢ btm]→[
    btm]]
= fun r m → match r with
| Empty → m
| Both r → close r ⦇bind (λx. 'm)⦈

let rec envr [@type γ δ. (γ, δ)rel→[δ ⊢ env]]
= fun r → match r with
| Empty → ⦇empty⦈
| Both r →
  let s = envr r in ⦇_, x ⊢ dot ('s[_]) x⦈

let rec conv [@type γ δ.(γ, δ)rel→[γ ⊢tm]→[δ ⊢ctm
    ]]
= fun r m → match m with
| ⦇ lam (λx. 'm)  ⦈p →
  let mc = conv (Both r) m in
  let mb = close r ⦇bind(λx. embed 'mc)⦈
  in let s = envr r in ⦇clo (clam {'mb}) 's⦈
| ⦇#x⦈p → lookup ⦇#x⦈ r
| ⦇app 'm 'n⦈p → let mm, nn = conv r m, conv r n in
                   ⦇capp 'mm 'nn⦈
```

The core of the translation is defined in functions `conv`, `envr`, and `close`. The main function is `conv`. It is implemented by recursion on the source term. There are three cases: i) source variables simply get translated by looking them up in the context relation, ii) applications just get recursively translated each term in the application, and iii) lambda expressions are translated recursively by converting the body of the expression in the extended context (notice the recursive call with `Both r`) and then turning the lambda expression into a closure.

In the first step we generate the closed body by the function `close` that adds the multiple binders (constructors `bind` and `embed`) and generates the closed term. Note that the return type `[btm]` of `close` guarantees that the final result is indeed a closed term, because we omit the context. For clarity, we could have written `[ ⊢ btm]`.

Finally, the function `envr` computes the substitution (represented by the type `env`) for the closure.

The implementation of closure conversion shows how to enforce closed terms in the specification, and how to make contexts and their relationships explicit at run-time using OCaml's GADTs. We believe it also illustrates well how HOAS trees can be smoothly manipulated and integrated into OCaml programs that may use effects.

4.3 Core-ML: A Functional Language with Pattern Matching and Data Types

We now introduce Core-ML, a functional language based on ML with pattern matching and data types. In Section 4.5 we will extend this language to also support contextual types and terms in our syntactic framework SF.

We keep the language design of Core-ML minimal in the interest of clarity. However, our prototype implementation which we describe in Section 4.9 supports interaction with all of OCaml's features such as exceptions, references and GADTs.

$$
\begin{array}{llcl}
\text{Types} & \tau & ::= & D \mid \tau_1 \to \tau_2 \\
\text{Expressions} & e & ::= & i \mid \mathtt{fun}\, f(x) = e \mid \\
 & & & \mathtt{let}\, x = i\, \mathtt{in}\, e \mid \mathtt{match}\, i\, \mathtt{with}\, \overrightarrow{b} \\
\text{Neutral Exp.} & i & ::= & i\, e \mid C\, \overrightarrow{e} \mid x \mid e : \tau \\
\text{Patterns} & pat & ::= & C\, \overrightarrow{pat} \mid x \\
\text{Branches} & b & ::= & \mid pat \mapsto e \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau \\
\text{Signature} & \Xi & ::= & \cdot \mid \Xi, D : \mathtt{type} \mid \Xi, C : \overrightarrow{\tau} \to D
\end{array}
$$

In Core-ML, we declare data-types by adding type formers (D) and type constructors (C) to the signature (Ξ). Constructors must be fully-applied. In addition all functions are named and recursive. The language supports pattern matching with nested patterns where patterns consist of just variables and fully applied constructors. We assume that all patterns are linear (i.e. each variable occurs at most once) and that they are covering.

The bi-directional typing rules for Core-ML have access to a signature Ξ and are standard (see Fig. 4.3) and the signature remains unchanged throughout the rules.

There are three things to notice in the rules:

1. The rule `t-constr` ensures that the constructor C is fully applied.

2. When type checking pattern matching in rule `t-match` the branches are checked against the type that the pattern needs to have (i.e.: τ') and the type the body needs to have (i.e.: τ). The use of the arrow in this context is an overloading of syntax to mean the type of the pattern and the type of the body.

$\boxed{\Gamma \vdash e \Leftarrow \tau}$: Expression e checks against type τ in context Γ

$$\frac{\Gamma, f : \tau \to \tau', x : \tau \vdash e \Leftarrow \tau'}{\Gamma \vdash \texttt{fun}\, f(x) = e \Leftarrow \tau \to \tau'}\ \texttt{t-rec} \qquad \frac{\Gamma \vdash i \Rightarrow \tau' \quad \Gamma, x : \tau' \vdash e \Leftarrow \tau}{\Gamma \vdash \texttt{let}\, x = i\, \texttt{in}\, e \Leftarrow \tau}\ \texttt{t-let}$$

$$\frac{\Gamma \vdash i \Rightarrow \tau' \quad \forall b_k \in \overrightarrow{b}\ .\ \Gamma \vdash b_k \Leftarrow \tau' \to \tau}{\Gamma \vdash \texttt{match}\, i\, \texttt{with}\, \overrightarrow{b} \Leftarrow \tau}\ \texttt{t-match}$$

$$\frac{\Gamma \vdash i \Rightarrow \tau' \quad \tau = \tau'}{\Gamma \vdash i \Leftarrow \tau}\ \texttt{t-emb}$$

$\boxed{\Gamma \vdash i \Rightarrow \tau}$: Neutral expr. i synthesizes type τ in context Γ

$$\frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau}\ \texttt{t-ann} \qquad \frac{\Gamma \vdash i \Rightarrow \tau' \to \tau \quad \Gamma \vdash e \Leftarrow \tau'}{\Gamma \vdash i\, e \Rightarrow \tau}\ \texttt{t-app}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \Rightarrow \tau}\ \texttt{t-var}$$

$$\frac{\Xi(C) = \overrightarrow{\tau} \to D \quad \forall \tau_i \in \overrightarrow{\tau}\ .\ \forall e_i \in \overrightarrow{e}\ .\ \Gamma \vdash e_i \Leftarrow \tau_i}{\Gamma \vdash C\, \overrightarrow{e} \Rightarrow D}\ \texttt{t-constr}$$

$\boxed{\Gamma \vdash\mid pat \mapsto e \Leftarrow \tau_1 \to \tau_2}$: Branch checks against τ_1 and τ_2 in Γ

$$\frac{\vdash pat : \tau' \downarrow \Gamma' \quad \Gamma, \Gamma' \vdash e \Leftarrow \tau}{\Gamma \vdash\mid pat \mapsto e \Leftarrow \tau' \to \tau}\ \texttt{t-branch}$$

$\boxed{\vdash pat : \tau \downarrow \Gamma}$: Pattern pat is of type τ and binds variables in context Γ

$$\frac{}{\vdash x : \tau \downarrow x : \tau}\ \texttt{t-pat-var}$$

$$\frac{\Xi(C) = \overrightarrow{\tau} \to D \qquad \forall \tau_i \in \overrightarrow{\tau}\ .\ \forall pat_i \in \overrightarrow{pat}\ . \quad \vdash pat_i : \tau_i \downarrow \Gamma_i}{\vdash C\, \overrightarrow{pat} : D \downarrow \Gamma_1, \ldots, \Gamma_i}\ \texttt{t-pat-con}$$

Figure 4.3: Core-ML Typing Rules

$$\begin{array}{lrcl} \text{Values} & v & ::= & C\,\overrightarrow{v} \mid (\texttt{fun}\,f(x) = e)[\rho] \\ \text{Environments} & \rho & ::= & \cdot \mid \rho, v/x \\ \text{Closures} & L & ::= & e[\rho] \end{array}$$

$\boxed{e[\rho] \Downarrow v}$: Expression e evaluates to value v in environment ρ

$$\frac{i[\rho] \Downarrow (\texttt{fun}\,f(x) = e')[\rho'] \quad e[\rho] \Downarrow v' \quad e'[\rho', v'/x] \Downarrow v}{(i\,e)[\rho] \Downarrow v}\ \texttt{e-app}$$

$$\frac{\rho(x) = v}{x[\rho] \Downarrow v}\ \texttt{e-var} \qquad \frac{i[\rho] \Downarrow v_1 \quad e[\rho, v_1/x] \Downarrow v}{(\texttt{let}\,x = i\ \texttt{in}\,e)[\rho] \Downarrow v}\ \texttt{e-let}$$

$$\frac{i[\rho] \Downarrow v_1 \quad \vdash pat_1 \not\doteq v \quad (\texttt{match}\,i\,\texttt{with}\ \overrightarrow{b})[\rho] \Downarrow v}{(\texttt{match}\,i\,\texttt{with} \mid pat_1 \mapsto e_b :: \overrightarrow{b})[\rho] \Downarrow v}\ \texttt{e-match-fail}$$

$$\frac{i[\rho] \Downarrow v_1 \quad \hat{\Gamma} \vdash pat_1 \doteq v/\rho' \quad e_b[\rho, \rho'] \Downarrow v}{(\texttt{match}\,i\,\texttt{with} \mid pat_1 \mapsto e_b :: \overrightarrow{b})[\rho] \Downarrow v}\ \texttt{e-match-succ}$$

$$\frac{\overrightarrow{e}\,[\rho] \Downarrow \overrightarrow{v}}{(C\,\overrightarrow{e}\,)[\rho] \Downarrow C\,\overrightarrow{v}}\ \texttt{e-constr} \qquad \frac{e[\rho] \Downarrow v}{e : \tau[\rho] \Downarrow v}\ \texttt{e-ann} \qquad \frac{i[\rho] \Downarrow v}{i[\rho] \Downarrow v}\ \texttt{e-emb}$$

$$\frac{}{(\texttt{fun}\,f(x) = e)[\rho] \Downarrow (\texttt{fun}\,f(x) = e)[\rho]}\ \texttt{e-fun}$$

Figure 4.4: Core-ML Big-Step Operational Semantics

3. Finally, in the rule for branches(`t-branch`) we check the body in the context Γ extended with the context from the pattern of the branch (context Γ').

In Figure 4.4 we define the operational semantics using an environment based approach. We define:

- values, that are either constructors applied to other values or recursive functions.

- environments that assign a value to each variable in the context.
- closures that represent pending computations in the environment. We write $e[\rho]$ for a closure that consists of expression e in context ρ.

We use an environment based operational semantics to avoid having to define substitutions at the level of Core-ML and later, substitution of quoted variables. This agrees with Babybel that in its implementation we do not need to define quoted variable substitution as we reuse OCaml's substitution so this style of semantics seems fitting.

The evaluation judgment $e[\rho] \Downarrow v$ means that the expression e in environment ρ evaluates in a big step to value v. The more interesting rules are the ones for pattern matching that use first-order matching as defined in Fig. 4.5. As traditional, in a `match` expression, patterns are matched branch by branch until one matches and then the body is executed in the extended environment that resulted from the matching.

We characterize both matching and failure to match. Successful matching is defined by the judgment $\Gamma \vdash pat \doteq v/\rho$ that means that the pattern *pat* with unification variables in Γ and the value v match producing the substitution ρ. Failure to match is defined by the judgment $\vdash pat \not\doteq v$ that states that the pattern *pat* with unification variables in Γ and the value v cannot be matched. Since v is always ground we do not need unification but matching. We also will not address the details of pattern matching compilation but merely state that it is possible to implement it in an efficient manner using decision trees [Augustsson, 1985].

4.4 A Syntactic Framework

In this section we describe the Syntactic Framework (SF) based on the modal logic S4 [Davies and Pfenning, 2001]. Our framework characterizes

$\boxed{\hat{\Gamma} \vdash pat \doteq v/\rho}$: Value v matches pat producing substitution ρ

$$\frac{}{x \vdash x \doteq v/(\cdot, v/x)} \texttt{m-v}$$

$$\frac{n = |\overrightarrow{pat}| \quad \hat{\Gamma} = \hat{\Gamma}_0, \ldots, \hat{\Gamma}_{n-1} \quad \forall i < n. \hat{\Gamma}_i \vdash \; pat_i \doteq v_i/\rho_i}{\hat{\Gamma} \vdash C\,\overrightarrow{pat} \doteq C\,\overrightarrow{v}\,/\rho_0, \ldots \rho_{n-1}} \texttt{m-c}$$

$\boxed{\vdash pat \not\doteq v}$: Value v does not match pattern pat

$$\frac{k \neq m}{\vdash C\,\overrightarrow{pat} \not\doteq C\,\overrightarrow{pat}} \texttt{f-c} \qquad \frac{}{\vdash pat \not\doteq \texttt{fun}\, f(x) = e} \texttt{f-f}$$

$$\frac{n = |\overrightarrow{pat}| \quad \exists i < n.\ pat_i \not\doteq v_i}{\vdash C\,\overrightarrow{pat} \not\doteq C\,\overrightarrow{v}} \texttt{f-r}$$

Figure 4.5: First-order Matching

only normal forms. All computation is delegated to the ML layer, that will perform pattern matching and substitutions on terms.

4.4.1 The definition of SF

The Syntactic Framework (SF) is a simply typed λ-calculus based on S4 where the type system forces all variables to be of base type, and all constants declared in a signature Σ to be fully applied. This simplifies substitution as variables of base type cannot be applied to other terms, and in consequence, there is no need for hereditary substitution in the specification language. Finally, the syntactic framework supports the box type to describe closed terms [Pfenning and Davies, 2001]. It can also be viewed as a restricted version of the contextual modality in [Nanevski et al., 2008] which could be an interesting extension to our work.

Having closed objects enforced at the specification level is not strictly

necessary. However, being able to state that some objects are closed in the specification has two distinct advantages: first, the user can specify some objects as closed so their contexts are always empty. This removes the need for some unnecessary substitutions. Second, it allows us to encode more fine-grained invariants and is hence an important specification tool (i.e. when implementing closure conversion in Section 4.2.1.3).

$$
\begin{array}{llcl}
\text{Types} & A, B & ::= & \mathbf{a} \mid A \rightarrow B \mid \Box A \\
\text{Terms} & M, N & ::= & \mathbf{c}\,\overrightarrow{M} \mid \lambda\, x.M \mid \{M\} \mid x \\
\text{Contexts} & \Psi, \Phi & ::= & \cdot \mid \Psi, x : \mathbf{a} \\
\text{Signature} & \Sigma & ::= & \cdot \mid \Sigma, \mathbf{a} : K \mid \Sigma,\ \mathbf{c} : A
\end{array}
$$

Fig. 4.6 shows the typing rules for the syntactic framework. Note that constructors always are fully applied (as per rule `t-con`), and that all variables are of base type as enforced by rules `t-var` and `t-lam`.

The specification framework's terms are manipulated by the computational language, in the resulting system any concrete use of terms in the syntactic framework will be done by pattern matching and the application of substitutions in the computational language therefore we will not have any elimination forms in the syntactic framework.

4.4.2 Contextual Types

We use contextual types [Nanevski et al., 2008] to embed possibly open SF objects in Core-ML and ensure that they are well-scoped. We use contextual types where we pair the type A of an SF object together with its surrounding context Ψ in which it makes sense. This follows the design of Beluga [Pientka, 2008, Cave and Pientka, 2012].

$$
\begin{array}{lccl}
\text{Contextual Types} & U & ::= & [\Psi \vdash A] \\
\text{Type Erased Contexts} & \hat{\Psi} & ::= & \cdot \mid \hat{\Psi}, x \\
\text{Contextual Objects} & C & ::= & [\hat{\Psi} \vdash M]
\end{array}
$$

$\boxed{\Psi \vdash M : A}$: M has type A in context Ψ

$$\frac{\Psi, x : \mathbf{a} \vdash M : A}{\Psi \vdash \lambda x.M : \mathbf{a} \to A}\ \texttt{t-lam} \qquad \frac{\cdot \vdash M : A}{\Psi \vdash \{M\} : \Box A}\ \texttt{t-box} \qquad \frac{\Psi(x) = \mathbf{a}}{\Psi \vdash x : \mathbf{a}}\ \texttt{t-var}$$

$$\frac{\Sigma(\mathbf{c}) = A \quad \Psi \vdash \overrightarrow{M} : A/\mathbf{a}}{\Psi \vdash \mathbf{c}\,\overrightarrow{M} : \mathbf{a}}\ \texttt{t-con}$$

$\boxed{\Psi \vdash \overrightarrow{M} : A/B}$: spine $\overrightarrow{M}$ checks against type A and has target type B

$$\frac{}{\Psi \vdash \cdot : \mathbf{a}/\mathbf{a}}\ \texttt{t-sp-em} \qquad \frac{\Psi \vdash N : A \quad \Psi \vdash \overrightarrow{M} : B/\mathbf{a}}{\Psi \vdash N\overrightarrow{M} : A \to B/\mathbf{a}}\ \texttt{t-sp}$$

Figure 4.6: Syntactic Framework Typing

Contextual objects, written as $[\hat{\Psi} \vdash M]$ pair the term M with the variable name context $\hat{\Psi}$ to allow for α-renaming of variables occurring in M. Note how the $\hat{\Psi}$ context just corresponds to the context with the typing assumptions erased.

When we embed contextual objects in a programming language we want to refer to variables and expressions from the ambient language, in order to support incomplete terms. Following [Nanevski et al., 2008, Pientka, 2008], we extend our syntactic framework SF with two ideas: first, we have incomplete terms with meta-variables to describe holes in terms. As in Beluga, there are two different kinds: *quoted variables* `'u` represent a hole in the term that may be filled by an arbitrary term. In contrast, *parameter variables* represent a hole in a term that may be filled only with some bound variable from the context. Concretely, a parameter variable may be `#x` and describe any concrete variable from a context Ψ. We may also want to restrict what bound variables a parameter variable describes. For example, if we have two sharp signs (i.e. `##x`) the top-most variable declaration is excluded. Intuitively, the number of sharp signs,

after the first, in front of x correspond to a weakening (or in de Bruijn lingo the number of shifts). Second, substitution operations allow us to move terms from one context to another.

We hence extend the syntactic framework SF with quoted variables, parameter variables and closures, written as $M[\sigma]$. We annotate the substitution with its domain and range to simplify the typing rule, however our prototype omits these typing annotations and lets type inference infer them.

Parameter Variables	v	::=	$\#\mathsf{x} \mid \#\#\mathsf{x}$
Terms	M	::=	$\cdots \mid \mathsf{'u} \mid \mathsf{v} \mid M[\sigma]$
Substitutions	σ	::=	$\cdot \mid \sigma, M/x$
Ambient Ctx.	Γ	::=	$\cdots \mid \Gamma, \mathsf{u} : [\Psi \vdash \mathbf{a}]$

In addition, we extend the context Γ of the ambient language Core-ML to keep track of assumptions that have a contextual type.

Finally, we extend the typing rules of the syntactic framework SF to include quoted variables, parameter variables, closures, and substitutions. We keep all the previous typing rules for SF from Section 4.4 where we thread through the ambient Γ, but the rules remain unchanged otherwise.

$\boxed{\Gamma; \Psi \vdash_v \mathtt{v} : \mathbf{a}}$: Parameter Variable $\mathtt{v}$ has type $\mathbf{a}$ in contexts Ψ and Γ

$$\frac{\Gamma(x) = [\Psi \vdash \mathbf{a}]}{\Gamma; \Psi \vdash_v \#\mathtt{x} : \mathbf{a}}\ \texttt{t-pvar-v} \qquad \frac{\Gamma; \Psi \vdash_v \mathtt{v} : \mathbf{a}}{\Gamma; \Psi, y : _ \vdash_v \#\mathtt{v} : \mathbf{a}}\ \texttt{t-pvar-\#}$$

$\boxed{\Gamma; \Psi \vdash M : A}$: Term M has type A in contexts Ψ and Γ

$$\frac{\Gamma(\mathtt{u}) = [\Psi \vdash \mathbf{a}]}{\Gamma; \Psi \vdash \mathtt{'u} : \mathbf{a}}\ \texttt{t-qvar} \qquad \frac{\Gamma; \Psi \vdash_v \mathtt{v} : \mathbf{a}}{\Gamma; \Psi \vdash \mathtt{v} : \mathbf{a}}\ \texttt{t-pvar}$$

$$\frac{\Gamma; \Psi \vdash \sigma : \Phi \quad \Gamma; \Phi \vdash M : A}{\Gamma; \Psi \vdash M[\sigma] : A}\ \texttt{t-sub}$$

$\boxed{\Gamma; \Psi \vdash \sigma : \Psi'}$: Substitution σ from Ψ' toΨ in the amb. ctx. Γ

$$\frac{}{\Gamma; \Psi \vdash \cdot : \cdot}\ \texttt{t-empty-sub} \qquad \frac{\Gamma; \Psi \vdash \sigma : \Psi' \quad \Gamma; \Psi \vdash M : \mathbf{a}}{\Gamma; \Psi \vdash \sigma, M/x : (\Psi', x : \mathbf{a})}\ \texttt{t-dot-sub}$$

The rules for quoted and parameter variables (`t-qvar` and `t-pvar` respectively) might seem very restrictive as we can only use a meta-variable of type $\Psi \vdash \mathbf{a}$ in the same context Ψ. As a consequence meta-variables often occur as a closure paired with a substitution (i.e.: $\mathtt{'u}\,[\sigma]$). This leads to the following admissible rule:

$$\frac{\Gamma(\mathtt{u}) = [\Phi \vdash \mathbf{a}] \quad \Delta; \Psi \vdash \sigma : \Phi}{\Gamma; \Psi \vdash \mathtt{'u}\,[\sigma] : \mathbf{a}}\ \texttt{t-qvar-adm}$$

We stress that $M[\sigma]$ it is an operation that is applied and not part of the syntax of our terms.

Note that when we compile SF objects, this substitution will be eagerly applied in order to keep terms in the syntactic framework in normal form.

Substitution is straightforward to define. We write it here as a prefix, to stress that it is an operation that is applied and not part of the term.

Substitutions ([_]_) is defined as:

$$
\begin{array}{rcl}
[\sigma](\lambda\, x.M) & = & \lambda\, x.(M[\sigma, x/x]) \\
[\sigma]\,\{M\} & = & \{M\} \\
[\sigma]\,\texttt{'u} & = & \texttt{'u} \\
[\sigma]\,\texttt{\#x} & = & \texttt{\#x} \\
[\sigma]\,x & = & \texttt{lookup}\; x\; \sigma \\
[\sigma](M[\phi]) & = & [\sigma \circ \phi]\, M \\
[\sigma](\mathbf{c}\,\vec{M}) & = & \mathbf{c}\,\vec{N} \quad \text{where } N_i = ([\sigma]M_i) \\
[\sigma]\{M\} & = & \{M\}
\end{array}
$$

We distinguish the actual operation of applying a substitution from the explicit substitution in a term, by writing the substitution in a prefix position.

Composing substitutions is defined as:

$$
\begin{array}{cccl}
\sigma & \circ & \phi, M/x & = (\sigma \circ \phi), ([\sigma]\, M)/x \\
\sigma & \circ & \cdot & = \sigma
\end{array}
$$

The lookup of variables in substitutions is defined as:

$$
\begin{array}{lllcl}
\texttt{lookup} & x & \sigma, M/x & = & M \\
\texttt{lookup} & x & \sigma, M/y & = & \texttt{lookup}\; x\; \sigma
\end{array}
$$

Note that looking up in an empty substitution is not defined as it is ill-typed.

The next step is to define the embedding of this framework in a programming language that will provide the computational power to analyze and manipulate contextual objects.

4.5 Core-ML with Contextual Types

To embed contextual SF objects into Core-ML, we extend the syntax of Core-ML as follows:

Types	τ	::=	$\cdots \mid [\Psi \vdash \mathbf{a}]$
Expressions	e	::=	$\cdots \mid [\hat{\Psi} \vdash M] \mid \mathtt{cmatch}\, e\, \mathtt{with}\, \overrightarrow{c}$
Patterns	pat	::=	$\cdots \mid [\hat{\Psi} \vdash R]$
Contextual Branches	c	::=	$\cdots \parallel [\Psi \vdash R] \mapsto e$

In particular, we allow programmers to use the expression:

$$\mathtt{cmatch}\, e\, \mathtt{with}\, \overrightarrow{c}$$

to directly pattern match on the syntactic structures they define in SF.

4.5.1 SF Objects as SF Patterns

We allow programmers to analyze SF objects directly via pattern matching. The grammar of SF patterns follows the grammar of SF objects.

SF Parameter Pattern	$\mathtt{w}$	::=	$\mathtt{\#p} \mid \mathtt{\#\#w}$
SF Patterns	R	::=	$\lambda\, x.R \mid \{R\} \mid x \mid \mathbf{c}\, \overrightarrow{R} \mid \mathtt{'u} \mid \mathtt{w}$

However, there is an important restriction: closures are not allowed in SF patterns. Intuitively this means that all quoted variables are associated with the identity substitution and hence depend on the entire context in which they occur. Parameter variables may be associated with weakening substitutions. This allows us to easily infer the type of quoted variables and parameter variables as we type check a pattern. This is described by the judgment:

$$\boxed{\Psi \vdash R : A \downarrow \Gamma} : \text{Pattern } R \text{ has type } A \text{ in } \Psi \text{ and binds } \Gamma$$

$\boxed{\Psi \vdash_v \texttt{w} : \mathbf{a} \downarrow \Gamma}$: Parameter Pattern w has type **a** in Ψ and binds Γ

$$\frac{}{\Psi \vdash_v \#\texttt{p} : \mathbf{a} \downarrow p : [\Psi \vdash \mathbf{a}]}\ \texttt{tp-pvar} \qquad \frac{\Psi \vdash_v \texttt{w} : \mathbf{a} \downarrow \Gamma}{\Psi, y : _ \vdash_v \#\texttt{w} : \mathbf{a} \downarrow \Gamma}\ \texttt{tp-pvar-\#}$$

$\boxed{\Psi \vdash R : A \downarrow \Gamma}$: Pattern R has type A in Ψ and binds Γ

$$\frac{\Psi, x : \mathbf{a} \vdash R : A \downarrow \Gamma}{\Psi \vdash \lambda x.R : \mathbf{a} \to A \downarrow \Gamma}\ \texttt{tp-lam} \qquad \frac{\cdot \vdash R : A \downarrow \Gamma}{\Psi \vdash \{R\} : \Box A \downarrow \Gamma}\ \texttt{tp-box}$$

$$\frac{}{\Psi \vdash \texttt{'u} : \mathbf{a} \downarrow u : [\Psi \vdash \mathbf{a}]}\ \texttt{tp-mvar}$$

$$\frac{\Sigma(\mathbf{c}) = A \quad \Psi \vdash \overrightarrow{R} : A/\mathbf{a} \downarrow \Gamma}{\Psi \vdash \mathbf{c}\, \overrightarrow{R} : \mathbf{a} \downarrow \Gamma}\ \texttt{tp-constr}$$

$$\frac{\Psi \vdash_v \texttt{w} : \mathbf{a} \downarrow \Gamma}{\Psi \vdash \texttt{w} : \mathbf{a} \downarrow \Gamma}\ \texttt{tp-pvar} \qquad \frac{\Psi(x) = \mathbf{a}}{\Psi \vdash x : \mathbf{a} \downarrow \cdot}\ \texttt{tp-var}$$

$\boxed{\Psi \vdash \overrightarrow{M} : A/B \downarrow \Gamma}$: Pat. Spine $\overrightarrow{R}$ has type A and target B and binds Γ

$$\frac{}{\Psi \vdash \cdot : \mathbf{a}/\mathbf{a} \downarrow \cdot}\ \texttt{tp-sp-em} \qquad \frac{\Psi \vdash N : A \downarrow \Gamma \quad \Psi \vdash \overrightarrow{M} : B/\mathbf{a} \downarrow \Gamma'}{\Psi \vdash N\, \overrightarrow{M} : A \to B/\mathbf{a} \downarrow \Gamma, \Gamma'}\ \texttt{tp-sp}$$

Figure 4.7: Typing Rules for SF Patterns

Figure 4.7 shows the typing rules for SF patterns. They closely follow the typing of SF terms. The more interesting ones are the parameter patterns as they illustrate the built-in weakening.

Further, the matching algorithm for SF patterns degenerates to simple first-order matching [Pientka and Pfenning, 2003] and can be defined straightforwardly. However, it is worth considering the matching rules for parameter patterns. As matching will only consider well-typed terms, we know that in the rules `m-pv` and `m-pv-#` the variable x is well-typed

in the context $\hat{\Psi}$.

$\boxed{\Gamma; \hat{\Psi} \vdash_v w \doteq x/\rho}$: Param. Patt. w matches var. x from $\hat{\Psi}$ producing ρ.

$$\frac{}{\mathsf{p} : [\Psi \vdash A]; \hat{\Psi} \vdash_v \#\mathsf{p} \doteq x/\cdot, [\hat{\Psi} \vdash x]/\mathsf{p}}\ \texttt{m-pv}$$

$$\frac{x \neq y \quad \Gamma; \hat{\Psi} \vdash_v w \doteq x/\rho}{\Gamma; \hat{\Psi}, y \vdash_v \#\mathsf{w} \doteq x/\rho}\ \texttt{m-pv-\#}$$

$\boxed{\Gamma; \hat{\Psi} \vdash R \doteq M/\rho}$: M matches pattern R with vars. in $\hat{\Psi}$ producing ρ.

$$\frac{\Gamma; \hat{\Psi}, x \vdash R \doteq M/\rho}{\Gamma; \hat{\Psi} \vdash \lambda x.R \doteq \lambda x.M/\rho}\ \texttt{m-}\lambda \qquad \frac{}{\cdot; \hat{\Psi} \vdash x \doteq x/\cdot}\ \texttt{m-bv}$$

$$\frac{\Gamma; \cdot \vdash R \doteq M/\rho}{\Gamma; \hat{\Psi} \vdash \{R\} \doteq \{M\}/\rho}\ \texttt{m-box}$$

$$\frac{\text{for all } R_i \in \overrightarrow{R} \text{ such as } \Gamma; \hat{\Psi} \vdash R_i \doteq M_i/\rho_i}{\Gamma; \hat{\Psi} \vdash \mathbf{c}\,\overrightarrow{R} \doteq \mathbf{c}\,\overrightarrow{M}/\rho_0, \ldots, \rho_n}\ \texttt{m-cc}$$

$$\frac{}{\mathsf{u} : [\Psi \vdash A]; \hat{\Psi} \vdash \text{'}\mathsf{u} \doteq M/\cdot, [\hat{\Psi} \vdash M]/\mathsf{u}}\ \texttt{m-cv} \qquad \frac{\Gamma; \hat{\Psi} \vdash_v \mathsf{w} \doteq x/\rho}{\Gamma; \hat{\Psi} \vdash \mathsf{w} \doteq x/\rho}\ \texttt{m-pv}$$

Finally, it has another important consequence: closures only appear in the branches of case-expressions. As Core-ML has a call-by-value semantics, we know the instantiations of quoted variables and parameter variables when they appear in the body of a case-expression and all closures are eliminated by applying the substitution eagerly. Given these conditions the matching operation remains first-order. An alternative way of explaining this is that in fact we have a degenerate case of the pattern fragment [Miller, 1991] of higher-order unification where all unification variables are applied to different variables and all the variables in the context. In this situation we only need first order matching [Pientka and

Pfenning, 2003], which enables efficient pattern matching compilation.

4.5.2 Typing Rules for Core-ML with Contextual Types

We now add the following typing rules for contextual objects and pattern matching to the typing rules of Core-ML:

$$\frac{\Gamma; \Psi \vdash M : \mathbf{a}}{\Gamma \vdash [\hat{\Psi} \vdash M] \Leftarrow [\Psi \vdash \mathbf{a}]}\ \texttt{t-ctx-obj}$$

$$\frac{\Gamma \vdash i \Rightarrow [\Psi \vdash \mathbf{a}] \quad \forall b \in \overrightarrow{b}\ .\ \Gamma \vdash b \Leftarrow [\Psi \vdash \mathbf{a}] \to \tau}{\Gamma \vdash \texttt{cmatch}\, i\, \texttt{with}\ \overrightarrow{b} \Leftarrow \tau}\ \texttt{t-cm}$$

$$\frac{\Psi \vdash R : \mathbf{a} \downarrow \Gamma' \quad \Gamma, \Gamma' \vdash e \Leftarrow \tau}{\Gamma \vdash [\Psi \vdash R] \mapsto e \Leftarrow [\Psi \vdash \mathbf{a}] \to \tau}\ \texttt{t-cbranch}$$

The typing rule for contextual objects (rule `t-ctx-obj`) simply invokes the typing judgment for contextual objects. Notice, that we need the context Γ when checking contextual objects, as they may contain quoted variables from Γ.

Extending the operational semantics to handle contextual SF objects is also straightforward.

Additionally, we need rules to evaluate the matching of contextual types that largely follow the rules for pattern matching `e-match-succ` and `e-match-fail` but use the matching operation defined for contextual objects.

Finally, the operational semantics needs to be extended with rules to support the new constructs. The rules, in Figure 4.8 for contextual terms simply take the terms apart and re-build them, except for quoted variables that are looked up in the environment(rule `ec-qvar`) and for terms with substitutions where the substitution operation is immediately applied(rule `ec-sub`). The rules for the new pattern matching construct are analogous to the ones from Core-ML.

$\boxed{e[\rho] \Downarrow v}$: Expression e evaluates to value v in environment ρ

$$\frac{[\hat{\Psi}, x \vdash M][\rho] \Downarrow [\hat{\Psi}, x \vdash N]}{[\hat{\Psi} \vdash \lambda\, x.M][\rho] \Downarrow [\hat{\Psi} \vdash \lambda\, x.N]}\ \texttt{ec-lam}$$

$$\frac{[\cdot \vdash \{M\}][\rho] \Downarrow [\cdot \vdash \{N\}]}{[\hat{\Psi} \vdash \{M\}][\rho] \Downarrow [\hat{\Psi} \vdash \{N\}]}\ \texttt{ec-box}$$

$$\frac{}{[\hat{\Psi} \vdash x][\rho] \Downarrow [\hat{\Psi} \vdash x]}\ \texttt{ec-var}$$

$$\frac{\forall M \in \overrightarrow{M}.\ M[\rho] \Downarrow N}{[\hat{\Psi} \vdash \mathbf{c}\,\overrightarrow{M}][\rho] \Downarrow [\hat{\Psi} \vdash \mathbf{c}\,\overrightarrow{N}]}\ \texttt{ec-constr}$$

$$\frac{\rho(\mathsf{u}) = M}{\texttt{'}\mathsf{u}[\rho] \Downarrow M}\ \texttt{ec-qvar} \qquad \frac{\rho(\mathsf{x}) = y}{\texttt{\#}\mathsf{x}[\rho] \Downarrow y}\ \texttt{ec-pvar} \qquad \frac{\rho(\mathsf{x}) = y}{\texttt{\#\#}\mathsf{x}[\rho, M/z] \Downarrow y}\ \texttt{ec-ppvar}$$

$$\frac{([\sigma]M)[\rho] \Downarrow N}{M[\sigma][\rho] \Downarrow N}\ \texttt{ec-sub}$$

$$\frac{e[\rho] \Downarrow [\hat{\Psi} \vdash M] \quad \Psi \vdash R \not\doteq M \quad (\texttt{cmatch}\, e\, \texttt{with}\, \overrightarrow{c})[\rho] \Downarrow v}{(\texttt{cmatch}\, e\, \texttt{with} \mid [\Psi \vdash R] \mapsto e_b\, \overrightarrow{c})[\rho] \Downarrow v}\ \texttt{ec-mf}$$

$$\frac{e[\rho] \Downarrow [\hat{\Psi} \vdash M] \quad \Gamma'; \Psi \vdash R \doteq M/\rho' \quad e_b[\rho, \rho'] \Downarrow v}{(\texttt{cmatch}\, e\, \texttt{with} \mid [\Psi \vdash R] \mapsto e_b\, \overrightarrow{c})[\rho] \Downarrow v}\ \texttt{ec-ms}$$

Figure 4.8: Extended Operational Semantics

4.6 Core-ML with GADTs

So far, in this chapter, we reviewed how to support contextual types and contextual objects in a standard functional programming language. This allows us to define syntactic structures with binders and manipulate them with the guarantee that variables will not escape their scopes. This brings some of the benefits of the Beluga system to mainstream languages focusing on writing programs instead of proofs. A naive implementation of this language extension requires augmenting the type checker and operational semantics of the host language. This is a rather significant task – especially if it includes implementing a compiler for the extended language. However, we can take advantage of the powerful type-system in modern functional languages to make the implementation more straight forward. Concretely, we use Generalized Abstract Data Types (GADTs) as a target to our translation. GADTs are a generalization of algebraic data-types that allow types to be indexed by other types. This mild form type dependency is enough to implement the SF. In the literature, GADTs have been introduced several times under different names: phantom types [Cheney and Hinze, 2003b], guarded recursive types [Xi et al., 2003] and equality types [Sheard and Pasalic, 2008]. In this section, we describe how to embed Core-ML with contextual types in a functional language with GADTs, called Core-ML$^{\text{gadt}}$, based on $\lambda_{2,G\mu}$ by Xi et al. [2003]. The choice of this target language is motivated by the fact that it is close to what realistic typed languages already offer (e.g.: OCaml and Haskell) and it directly lends itself to an implementation.

Signatures	Σ	::=	$\cdot \mid \Sigma, D : (*, \ldots, *) \to * \mid C : \forall \overrightarrow{\alpha} \,.\, \tau \to D[\overrightarrow{\tau}]$
Types	τ	::=	$D[\overrightarrow{\tau}] \mid \forall \alpha \,.\, \tau \mid \tau_1 \to \tau_2 \mid \alpha \mid \tau_1 \times \tau_2$
Expressions	e	::=	$x \mid C[\overrightarrow{\tau}]\, e \mid \mathsf{fix}\, f : \tau = e \mid e_1\, e_2 \mid (e_1, e_2) \mid \lambda x \,.\, e$
			$\mid \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \mid \mathtt{match}\, e\, \mathtt{with}\, \overrightarrow{b}$
			$\mid \Lambda \alpha \,.\, e \mid e[\tau] \mid (e_1, e_2)$
Branch	b	::=	$pat \mapsto e$
Pattern	pat	::=	$x \mid C[\overrightarrow{\alpha}]\, pat \mid (pat_1, pat_2)$
Exp. Ctx.	Γ	::=	$\cdot \mid \Gamma, x : \tau$
Type Ctx.	Δ	::=	$\cdot \mid \Delta, \alpha \mid \Delta, \tau_1 \equiv \tau_2$

Core-ML$^{\text{gadt}}$ contains polymorphism and GADTs, which makes it a good ersatz OCaml that is still small and easy to reason about. GADTs are particularly convenient, since they allow us to track invariants about our objects in a similar fashion to dependent types. Compared to Core-ML, Core-ML$^{\text{gadt}}$'s signatures now store type constants and constructors that are parametrized by other types. We show the typing judgments for the language in Fig. 4.9. The term language is similar to Core-ML, with the addition of the usual terms for supporting abstraction over types (i.e. $\Lambda\alpha \,.\, e$) and type applications (i.e. $e[\tau]$).

The language is type-checked with the two typing judgments, one for terms and one for patterns:

- $\boxed{\Delta; \Gamma \vdash e : \tau}$: expression e is of type τ in typing context Δ and expression context Γ.
- $\boxed{\Delta_o \vdash pat : \tau \downarrow \Delta; \Gamma}$: pat is of type τ and binds type variables in Δ and variables in Γ.

The expressive power of pattern matching is greatly enhanced by the presence of a limited form of dependent types (i.e. types that depend on other types).

$\boxed{\Delta; \Gamma \vdash e : \tau}$: e is of type τ in typing context Δ and expression context Γ.

$$\frac{\Sigma(C) = \forall \overrightarrow{\alpha} \;.\; \tau_1 \to D[\overrightarrow{\alpha}] \quad \Delta; \Gamma \vdash e : \tau_1[\overrightarrow{\tau}] \quad \Delta \vdash \overrightarrow{\tau}\ \mathtt{wf}}{\Delta; \Gamma \vdash C[\overrightarrow{\tau}]\, e : D[\overrightarrow{\tau}]}\ \texttt{g-con} \qquad \frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Delta; \Gamma \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash e_1\, e_2 : \tau_2}\ \texttt{g-app}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \quad \Delta; \Gamma \vdash e_1 : \tau_2}{\Delta; \Gamma \vdash (e_1,\, e_2) : \tau_1 \times \tau_2}\ \texttt{g-pair} \qquad \frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau}\ \texttt{g-var} \qquad \frac{\Delta; \Gamma, f : \tau \vdash e : \tau}{\Delta; \Gamma \vdash \mathsf{fix}\, f : \tau = e : \tau}\ \texttt{g-fix}$$

$$\frac{\Delta; \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x \;.\; e : \tau_1 \to \tau_2}\ \texttt{g-lam} \qquad \frac{\Delta; \Gamma \vdash e : \forall \overrightarrow{\alpha} \;.\; \tau \quad \Delta; \Gamma \vdash \overrightarrow{\tau}_1\, \mathtt{wf}}{\Delta; \Gamma \vdash e[\overrightarrow{\tau}_1] : \tau[\overrightarrow{\tau}_1]}\ \texttt{g-tapp} \qquad \frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha \;.\; e : \forall \alpha \;.\; \tau}\ \texttt{g-Lam}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \quad \Delta; \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Delta; \Gamma \vdash \mathtt{let}\, x = e_1\ \mathtt{in}\ e_2 : \tau}\ \texttt{g-let} \qquad \frac{\Delta; \Gamma \vdash e : \tau_1 \quad \text{for all } i. \Delta; \Gamma \vdash b_i : \tau_1 \to \tau}{\Delta; \Gamma \vdash \mathtt{match}\, e\, \mathtt{with}\ \overrightarrow{b} : \tau}\ \texttt{g-match}$$

$$\frac{\Delta \vdash pat : \tau \downarrow_1 \Delta'; \Gamma' \quad \Delta, \Delta'; \Gamma, \Gamma' \vdash e : \tau_2}{\Delta; \Gamma \vdash pat \mapsto e : \tau_1 \to \tau_2}\ \texttt{g-branch}$$

$\boxed{\Delta_o \vdash pat : \tau \downarrow \Delta; \Gamma}$: pat is of type τ and binds variables in Δ and Γ

$$\frac{}{\Delta_0 \vdash x : \tau \downarrow \cdot; x : \tau}\ \texttt{gp-var}$$

$$\frac{\Delta_0 \vdash pat_1 : \tau_1 \downarrow \Delta_1; \Gamma_1 \quad \Delta_0 \vdash pat_2 : \tau_2 \downarrow \Delta_2; \Gamma_2}{\Delta_0 \vdash (pat_1,\, pat_2) : \tau_1 \times \tau_2 \downarrow \Delta_1, \Delta_2; \Gamma_1, \Gamma_2}\ \texttt{gp-pair}$$

$$\frac{\Sigma(C) = \forall \overrightarrow{\alpha} \;.\; \tau \to D[\overrightarrow{\tau_1}] \quad \Delta_0, \overrightarrow{\alpha}, \overrightarrow{\tau_1} \equiv \overrightarrow{\tau_2} \vdash pat : \tau \downarrow \Delta; \Gamma}{\Delta_o \vdash C[\overrightarrow{\alpha}]\, pat : D[\overrightarrow{\tau_2}] \downarrow \overrightarrow{\alpha}, \overrightarrow{\tau_1} \equiv \overrightarrow{\tau_2}, \Delta; \Gamma}\ \texttt{gp-con}$$

Figure 4.9: The Typing of Core-ML$^{\text{gadt}}$

Of particular interest are the type equalities introduced in the type context in the gp-con rule. Let's consider an example to understand this (a complete discussion is fully developed by Xi et al. [2003]). As an example, we define the type of vectors as lists indexed by their length (i.e. the classical dependent types example).

$$\begin{aligned}\Sigma \;=\;& \mathtt{z}{:}*, \mathtt{s}{:}* \to *, \mathtt{vec}{:}(*,*) \to *,\\ & \mathtt{Nil}{:}\forall\beta\,.\,\mathtt{vec}[\mathtt{z},\beta],\\ & \mathtt{Cons}{:}\forall\alpha,\beta\,.\,(\beta \times \mathtt{vec}[\alpha,\beta]) \to \mathtt{vec}[\mathtt{s}[\alpha],\beta]\end{aligned}$$

In the signature we need to define the type level encoding of natural numbers: types z and s, and the type of vectors vec that are indexed by their length and the type of their elements. Finally, we define the constructor Nil of empty vectors of length zero, thus the first index is the type z. And finally the constructor Cons that puts together an element of type β and a vector of length α to form a vector of length $\alpha + 1$, that is $\mathtt{s}[\alpha]$ in Core-ML^{gadt}.

Vectors are interesting because they make many functions type safe by relying on the more expressive type system. As an example, in Figure 4.10 shows the function zip that joins two vectors of the same length and produces a vector of the same lengths containing pairs of values taken from the original vectors:

In this case, type refinement happens for example in the second branch, where in the body we have the choice of referring to the length as α_1 or α_2, knowing they have to be the same. In fact, when typing the pattern in rule gp-con context Δ is extended with $\alpha \equiv \alpha_1$ and $\alpha \equiv \alpha_2$ from which is easy to conclude that $\alpha_1 \equiv \alpha_2$. Additionally, the type constraint that the vectors are of the same length makes it obvious that the diagonal cases (those when one vector has more elements than the other) are impossible.

Since Core-ML^{gadt} has strong type separation the operational semantics, in Figure 4.11, is similar to the semantics for Core-ML, after all, type information is irrelevant at run-time. The interested reader can find the

$$
\begin{array}{l}
\mathsf{fix}\,\mathtt{zip} : \forall \alpha, \beta_1, \beta_2\,. \\
\quad (\mathtt{vec}[\alpha, \beta_1] \times \mathtt{vec}[\alpha, \beta_2]) \rightarrow \mathtt{vec}[\alpha, (\beta_1,\ \beta_2)] = \\
\Lambda\alpha, \beta_1, \beta_2\,.\ \lambda v\,.\ \mathtt{match}\, v\, \mathtt{with} \\
\mid (\mathtt{Nil}[\beta_1], \mathtt{Nil}[\beta_2]) \mapsto \mathtt{Nil}[\beta_1 \times \beta_2] \\
\mid (\mathtt{Cons}[\alpha_1, \beta_1](x,\ xs), \mathtt{Cons}[\alpha_2, \beta_2](y,\ ys) \mapsto \\
\quad \mathtt{Cons}[\alpha_1, \beta_1 \times \beta_2]((x,\ y),\ \mathtt{zip}[\alpha_1, \beta_1, \beta_2](xs,\ ys)\,)
\end{array}
$$

Figure 4.10: Zip in Core-ML$^{\text{gadt}}$

meta-theory in [Xi et al., 2003]. We define values and environments as:

$$
\begin{array}{llll}
\text{Values} & v & ::= & C[\overrightarrow{\tau}]\,v \,|(v_1,\ v_2) \mid (\lambda x\,.\ e)[\theta; \rho] \mid (\Lambda\alpha\,.\ e)[\theta; \rho] \\
\text{Environments} & \theta; \rho & ::= & \cdot ; \cdot \mid \theta; (\rho, v/x) \mid (\theta, \tau/\alpha); \rho
\end{array}
$$

And with them the semantics are implemented with two judgments, one to evaluate expressions and the other to compute matching, the matching operation simply augments the environment inside of the branches in case expressions:

- $\boxed{e[\theta; \rho] \Downarrow v}$: Expression e evaluates to value v in environment ρ.
- $\boxed{pat \doteq v \backslash \theta; \rho}$: Value v matches pattern pat and produces env. $\theta; \rho$.

4.7 Deep Embedding of SF into Core-ML$^{\text{gadt}}$

We now show how to translate objects and types defined in the syntactic framework SF into Core-ML$^{\text{gadt}}$ using a deep embedding. We take advantage of the advanced features of Core-ML$^{\text{gadt}}$'s type system to fully type-check the result. Our representation of SF objects and types is inspired by [Benton et al., 2012b] but uses GADTs instead of full dependent types. We add the idea of typed context shifts instead of renamings to

$\boxed{e[\theta;\rho] \Downarrow v}$: Expression e evaluates to value v in envirnoment ρ

$$\frac{e_1[\theta;\rho] \Downarrow (\lambda x\,.\,e)[\theta';\rho'] \quad e_2[\theta;\rho] \Downarrow v' \quad e[\theta';\rho',v'/x] \Downarrow v}{(e_1\,e_2)[\theta;\rho] \Downarrow v}\ \texttt{ge-app}$$

$$\frac{e[\theta;\rho] \Downarrow \Lambda\alpha\,.\,e' \quad e'[\theta,\tau/\alpha;\rho] \Downarrow v}{e[\tau][\theta;\rho] \Downarrow v}\ \texttt{ge-tapp}$$

$$\frac{\rho(x) = v}{x[\theta;\rho] \Downarrow v}\ \texttt{ge-var} \qquad \frac{e_1[\theta;\rho] \Downarrow v_1 \quad e_2[\theta;\rho,v_1/x] \Downarrow v}{(\texttt{let}\,x = e_1\ \texttt{in}\,e_2)[\theta;\rho] \Downarrow v}\ \texttt{ge-let}$$

$$\frac{e[\theta;\rho] \Downarrow v_1 \quad pat_1 \doteq v\backslash\theta';\rho' \quad e_b[\theta,\theta';\rho,\rho'] \Downarrow v}{(\texttt{match}\,e\,\texttt{with}\,|\,pat_1 \mapsto e_b :: \overrightarrow{b})[\theta;\rho] \Downarrow v}\ \texttt{ge-match}$$

$$\frac{e[\theta;\rho] \Downarrow v}{\mathsf{C}[\overrightarrow{\tau}]\,e[\theta;\rho] \Downarrow \mathsf{C}[\overrightarrow{\tau}]\,v}\ \texttt{ge-constr} \qquad \frac{}{(\lambda x\,.\,e)[\theta;\rho] \Downarrow (\lambda x\,.\,e)[\theta;\rho]}\ \texttt{ge-lam}$$

$$\frac{}{(\Lambda\alpha\,.\,e)[\theta;\rho] \Downarrow (\Lambda\alpha\,.\,e)[\theta;\rho]}\ \texttt{ge-Lam}$$

$$\frac{e_1[\theta;\rho] \Downarrow v_1 \quad e_2[\theta;\rho] \Downarrow v_2}{(e_1,\, e_2)[\theta;\rho] \Downarrow (v_1,\, v_2)}\ \texttt{ge-pair} \qquad \frac{e[\theta;(\rho,\mathsf{fix}\,f : t = e/f)] \Downarrow v}{\mathsf{fix}\,f : t = e[\theta;\rho] \Downarrow v}\ \texttt{ge-fix}$$

$\boxed{pat \doteq v\backslash\theta;\rho}$: Value v matches pattern pat and produces env. $\theta;\rho$.

$$\frac{}{x \doteq v\backslash\cdot;(v/x)}\ \texttt{gm-v} \qquad \frac{pat_1 \doteq v_1\backslash\theta_1;\rho_1 \quad pat_2 \doteq v_2\backslash\theta_2;\rho_2}{((pat_1,\, pat_2) \doteq (v_1,\, v_2)\backslash\theta_1,\theta_2;\rho_1,\rho_2}\ \texttt{gm-p}$$

$$\frac{v \doteq pat/\theta;\rho}{\mathsf{C}[\overrightarrow{\alpha}]\,pat \doteq \mathsf{C}[\overrightarrow{\tau}]\,v\backslash(\overrightarrow{\tau}/\overrightarrow{\alpha},\theta);\rho}\ \texttt{gm-c}$$

Figure 4.11: Core-ML$^{\text{gadt}}$ Big-Step Operational Semantics

represent weakening. This is necessary to be able to completely erase types at run-time.

To ensure SF terms are well-scoped and well-typed, we define SF types in Core-ML$^{\text{gadt}}$ and index their representations by their type and context. The following types are only used as indices for GADTs. Because of that, they do not have any term constructors.

$$\begin{aligned}\Sigma \;=\;& \texttt{base}{:}* \to *, \texttt{arr}{:}(*,*) \to *, \texttt{boxed}{:}* \to *,\\ & \texttt{prod}{:}(*,*) \to *, \texttt{unit}{:}*\end{aligned}$$

We define three type families, one for each of SF's type constructors. It is important to note the number of type parameters they require. Base types take one parameter: a type from the signature. Function types simply have an input and output type. Finally, boxes contain just one type.

Terms are also indexed by the contexts in which they are valid. To this effect, we define two types to statically represent contexts. Analogously to the representation of types, these two types are only used during type-checking and there will be no instances at run-time. The type `nil` represents an empty context and thus has no parameters. And the constructor `cons` has two parameters the first one is the rest of the context and the second one is the type of the top-most variable so far.

$$\Sigma \;=\; \ldots, \texttt{nil}{:}*, \texttt{cons}{:}(*,*) \to *$$

We show the the encoding well-typed SF objects and types in Fig. 4.12. Every declaration is parametrized with the type of constructors that the user defined inside of the **`@@@signature`** blocks.

The specification takes the form of the type `con`$:(*,*) \to *$, where `con` is the name of a constructor from the signature indexed by the type of its parameters and the base type they produce. So, all the SF definitions the user makes add constructors, in our closure conversion example some constructors could be for instance: `capp`, `clam`, `clo`.

$$
\begin{aligned}
\Sigma \;=\; & \ldots, \mathtt{var}:(*,*) \to *, \\
& \mathtt{Top}: \forall \gamma, \alpha \,.\, \mathtt{var}[\mathtt{cons}[\gamma, \alpha], \alpha], \\
& \mathtt{Pop}: \forall \gamma, \alpha, \beta \,.\, \mathtt{var}[\gamma, \alpha] \to \mathtt{var}[\mathtt{cons}[\gamma, \beta], \alpha], \\
& \mathtt{sftm}:(*,*) \to *, \mathtt{sp}:(*,*,*) \to * \\
& \mathtt{Lam}: \forall \gamma, \alpha, \tau \,.\, \mathtt{sftm}[\mathtt{cons}[\gamma, \mathtt{base}[\alpha]], \tau] \to \mathtt{sftm}[\gamma, \mathtt{arr}[\mathtt{base}[\alpha], \tau]], \\
& \mathtt{Var}: \forall \gamma, \alpha \,.\, \mathtt{var}[\gamma, \alpha] \to \mathtt{sftm}[\gamma, \mathtt{base}[\alpha]], \\
& \mathtt{Box}: \forall \gamma, \tau \,.\, \mathtt{sftm}[\cdot, \tau] \to \mathtt{sftm}[\gamma, \tau], \\
& \mathtt{C}: \forall \gamma, \tau, \alpha \,.\, \mathtt{con}[\tau, \alpha] \times \mathtt{sp}[\gamma, \tau] \to \mathtt{sftm}[\gamma, \mathtt{base}[\alpha]], \\
& \mathtt{Empty}: \forall \gamma, \tau \,.\, \mathtt{sp}[\gamma, \tau, \tau], \\
& \mathtt{Cons}: \forall \gamma, \tau_1, \tau_2, \tau_3 \,.\, \mathtt{sftm}[\gamma, \tau_1] \times \mathtt{sp}[\gamma, \tau_2, \tau_3] \to \mathtt{sp}[\gamma, \mathtt{arr}[\tau_1, \tau 2], \tau_3], \\
& \mathtt{shift}:(*,*) \to *, \\
& \mathtt{Id}: \forall \gamma \,.\, \mathtt{shift}[\gamma, \gamma], \\
& \mathtt{Suc}: \forall \gamma, \delta, \alpha \,.\, \mathtt{shift}[\gamma, \delta] \to \mathtt{shift}[\mathtt{cons}[\gamma, \mathtt{base}[\alpha]], \delta], \\
& \mathtt{sub}:(*,*) \to *, \\
& \mathtt{Shift}: \forall \gamma, \delta \,.\, \mathtt{shift}[\gamma, \delta] \to \mathtt{sub}[\gamma, \delta], \\
& \mathtt{Dot}: \forall \gamma, \delta, \tau \,.\, \mathtt{sub}[\gamma, \delta] \times \mathtt{sftm}[\gamma, \tau] \to \mathtt{sub}[\gamma, \mathtt{cons}[\delta, \tau]]
\end{aligned}
$$

Figure 4.12: Syntactic Framework Definition

Variables and terms are indexed by two types, the first parameter is always their context and the second is their type. The type `var` represents variables with two constructors: `Top` represents the variable that was introduced last in the context and if `Top` corresponds to the de Bruijn index 0 then the constructor `Pop` represents the successor of the variable that it takes as parameter. It is interesting to consider the parameters of these constructors. `Top` is simply indexed by its context and type (variables γ and α respectively). On the other hand, `Pop` requires three type parameter: the first γ represents a context, α the resulting type of the variable, and β the type of the extension of the context. These parameter make it so that if we apply the constructor `Pop` to a variable of type α in context γ we obtain a variable of type α in the context γ extended with type β. `Top` and `Pop` contexts cannot be empty because they refer to existing variables. The constructor `Top` is indexed by a context with at

least one variable, and its type is the same as the top variable. On the other hand, the constructor `Pop` is also indexed by a non-empty context but its type corresponds to the type of the variable that it takes as a parameter.

As mentioned, terms described by the type family `sftm` are indexed by their context and their type. It is interesting to check in some detail how the indices of the term constructors follow the typing rules from Fig. 4.6. The constructor for lambda terms (`Lam`), extends the context γ with base type α and then it produces a term in γ of function type from the base type α to the type of the body τ. The constructor for boxes simply forces its body to be closed by using the context type `nil`. The constructor `Var` simply embeds variables as terms. Finally the `C` constructor has two parameters, one is the name of the constructor from the user's definitions that constrains the type of the second parameter, the other is the term of the appropriated type.

The definition of substitution is a modified presentation of the substitution for well-scoped de Bruijn indices, as for example presented in [Benton et al., 2012b]. We define two types, `sub` and `shift` indexed by two contexts, the domain and the range of the substitutions. Substitutions are either a shift (constructor `Shift`) or the combination of a term for the top-most variable and the rest of the substitution (constructor `Dot`).

Our implementation differs from Benton et al. [2012b] in the representation of renamings. Benton et.al define substitutions and renamings, the latter as a way of representing shifts. However to compute a shift, they need the context that they use to index the data-types. Hence, contexts are not erasable during run-time. As we do want contexts to be erasable at run-time, we cannot use renamings. Instead, we replace renamings with typed shifts (defined in type `shift`), that encode how many variables we are shifting over. This is encoded in the indices of shifts.

Variables that use de Bruijn indices and shifts ultimately correspond to natural numbers, as they encode how many binders we need to traverse,

or how many variables a shift adds. However, we use specialized data types because we explicitly carry the typing information in the indices. Finally, we omit the function implementing the substituti0on as it is standard. We will simply mention that we implement a function `apply_sub` of type:

$$\forall \gamma, \delta, \tau \;.\; \texttt{sftm}[\gamma, \tau] \rightarrow \texttt{sub}[\gamma, \delta] \rightarrow \texttt{sftm}[\delta, \tau]$$

That applies a substitution moving a term from context γ to context δ.

The finding the path to a variable example from Section 4.2.1.2, contains a signature that defines the untyped λ-calculus. This would result in a new type, and two new constructors for the existing type `con`. We will prefix the definitions with `def_` to create unique names:

- A new type: `def_tm:*`
- A constructor for applications:
  `def_app:con[base[def_tm], base[def_tm] × base[def_tm]]`
- A constructor for abstractions:
  `def_lam:con[base[def_tm], arr[base[def_tm], base[def_tm]]]`

To implement the signature we declare a new type for the terms of the defined language (`def_tm`), and we extend the type of SF constructors, with as many constructors as our signature had (`def_app` and `def_lam`).

4.8 From Core-ML with Contextual Types to Core-ML$^{\text{gadt}}$

In this section, we translate Core-ML with contextual types into the lower level Core-ML$^{\text{gadt}}$. Because our embedding of the syntactic framework SF in Core-ML$^{\text{gadt}}$ is intrinsically typed, there is no need to extend the type-checker to accommodate contextual objects. Further, recall that we restricted quoted variables and parameter variables such that the matching operation remains first order. In addition, as our deep embedding uses a representation with canonical names (namely de Bruijn indices), we are able to translate pattern matching into Core-ML$^{\text{gadt}}$'s pattern matching; thus there is no need to extend the operational semantics of the language.

The translation we describe in this section provides the footprint of an implementation of it to directly generate OCaml code, as Core-ML$^{\text{gadt}}$ is

essentially a subset of OCaml. It therefore shows how to extend a functional programming language such as OCaml with the syntactic framework with minimal impact on OCaml's compiler.

Our main translation of Core-ML to Core-ML$^{\text{gadt}}$ uses the following main operations:

$$\begin{array}{rl}
\ulcorner \tau \urcorner, \ulcorner \Xi \urcorner, \ulcorner \Gamma \urcorner : & \text{Translate types, signatures and contexts.} \\
\ulcorner e \urcorner_{\Gamma \vdash \tau} : & \text{Type directed translation of expressions.} \\
\ulcorner pat \urcorner^{\Gamma'}_{\Gamma \vdash \tau} : & \text{Translates patterns and outputs } \Gamma' \text{ the} \\
& \text{context of the bound variables.}
\end{array}$$

We begin by translating SF types and contexts into Core-ML$^{\text{gadt}}$ types. These types are used to index terms in the implementation of SF:

$$\begin{array}{lrcl}
\text{SF Types:} & \ulcorner \mathbf{a} \to A \urcorner & = & \mathtt{arr}[\mathbf{a}, \ulcorner A \urcorner] \\
& \ulcorner \Box A \urcorner & = & \mathtt{boxed}[\ulcorner A \urcorner] \\
& \ulcorner \mathbf{a} \urcorner & = & \mathbf{a} \\
\text{SF Contexts:} & \ulcorner \cdot \urcorner & = & \mathtt{nil}[] \\
& \ulcorner \Psi, x : \mathbf{a} \urcorner & = & \mathtt{cons}[\ulcorner \Psi \urcorner, \mathbf{a}]
\end{array}$$

The translation of SF terms is directed by their contextual type $\Psi \vdash A$, because it needs the context to perform the translation of names to

de Bruijn indices and the types to appropriately index the terms.

$$
\begin{array}{llcl}
\text{SF Terms:} & \ulcorner \lambda\, x.M \urcorner_{\Psi \vdash \mathbf{a} \to A} & = & \mathtt{Lam[cons[}\ulcorner\Psi\urcorner, \mathbf{a}], \ulcorner A \urcorner]\ulcorner M \urcorner_{\Psi,\mathbf{a} \vdash A} \\
& \ulcorner \{M\} \urcorner_{\Psi \vdash \Box A} & = & \mathtt{Box[}\ulcorner\Psi\urcorner, \ulcorner A \urcorner]\ulcorner M \urcorner_{\cdot \vdash A} \\
& \ulcorner x \urcorner_{\Psi \vdash \mathbf{a}} & = & \mathtt{Var[}\ulcorner\Psi\urcorner, \mathbf{a}]\ulcorner x \urcorner^{v}_{\Psi} \\
& \ulcorner \mathbf{c}\, \overrightarrow{M} \urcorner_{\Psi \vdash \mathbf{a}} & = & \mathtt{C[}\ulcorner\Psi\urcorner, \ulcorner A \urcorner, \mathbf{a}](\mathbf{c}, \ulcorner \overrightarrow{M} \urcorner_{\Psi \vdash A \downarrow \mathbf{a}}) \\
& & & \text{with } \Sigma(\mathbf{c}) = A \\
& \ulcorner M[\sigma] \urcorner_{\Psi \vdash A} & = & \mathtt{apply_sub}\ulcorner M \urcorner_{\Phi \vdash A}\ulcorner \sigma \urcorner_{\Psi \vdash \Phi} \\
& \ulcorner \mathtt{'u} \urcorner_{\Psi \vdash A} & = & u \\
& \ulcorner \mathtt{\#v} \urcorner_{\Psi \vdash \mathbf{a}} & = & \mathtt{Var[}\ulcorner\Psi\urcorner, \mathbf{a}]\ulcorner \mathtt{v} \urcorner_{\Psi \vdash \mathbf{a}} \\
& \ulcorner N\, \overrightarrow{M} \urcorner_{\Psi \vdash A \to B \downarrow \mathbf{a}} & = & \mathtt{Cons[}\ulcorner\Psi\urcorner, \mathtt{arr[}\ulcorner A \urcorner, \ulcorner B \urcorner], \mathbf{a}] \\
& & & \quad (\ulcorner N \urcorner_{\Psi \vdash A}, \ulcorner \overrightarrow{M} \urcorner_{\Psi \vdash B \downarrow \mathbf{a}}) \\
& \ulcorner \cdot \urcorner_{\Psi \vdash \mathbf{a} \downarrow \mathbf{a}} & = & \mathtt{Empty[}\ulcorner\Psi\urcorner, \mathbf{a}, \mathbf{a}] \\
\text{SF Vars.:} & \ulcorner x \urcorner^{v}_{\Psi, x : \mathbf{a} \vdash \mathbf{a}} & = & \mathtt{Top[}\ulcorner\Psi\urcorner, \mathbf{a}] \\
& \ulcorner y \urcorner^{v}_{\Psi, x : \mathbf{b} \vdash \mathbf{a}} & = & \mathtt{Pop[}\ulcorner\Psi\urcorner, \mathbf{a}, \mathbf{b}]\ulcorner y \urcorner^{v}_{\Psi \vdash \mathbf{a}} \\
\text{Param. Vars:} & \ulcorner \mathtt{x} \urcorner_{\Psi \vdash \mathbf{a}} & = & x \\
& \ulcorner \mathtt{\#v} \urcorner_{\Psi, y : \mathbf{a}' \vdash \mathbf{a}} & = & \mathtt{Pop[}\ulcorner\Psi\urcorner, \mathbf{a}, \mathbf{a}']\ulcorner \mathtt{v} \urcorner_{\Psi \vdash \mathbf{a}}
\end{array}
$$

There are are three kinds of variables in the syntactic framework SF: bound variables, quoted variables and parameter variables. Each kind requires a different translation strategy. Bound variables are translated into de Bruijn indices where the numbers are encoded using the constructors `Top` and `Pop`. Quoted variables are simply translated into the Core-ML^{gadt} variables they quote. And finally the parameter variables are translated into a `Var` constructor to indicate that the resulting expression is an SF variable, and the shifts (indicated by extra '#') are translated to applications of the constructor `Pop`.

Notice how substitutions are not part of the representation. They are translated to the eager application of `apply_sub`, an OCaml function that performs the substitution. Before we call `apply_sub` we translate

the substitution. This amounts to generating the right shift for empty substitutions and otherwise recursively translating the terms and the substitution.

We also need to translate SF patterns into Core-ML$^{\text{gadt}}$ expressions with the right structure. The special cases are:

- Variables are translated to de Bruijn indexes.
- Quoted variables simply translate to Core-ML$^{\text{gadt}}$ variables.
- Parameter variables translate to a pattern that matches only variables by specifying the `Var` constructor.

The translation of patterns follows the same line as the translation of terms, however, we do not use the indices of type variables in Core-ML$^{\text{gadt}}$ patterns. This is indicated by writing an underscore.

Additionally, the translation of substitutions amounts to generating the right shift for empty substitutions and otherwise recursively translating the terms and the substitution:

$$\begin{array}{rcl}
\ulcorner \cdot \urcorner^{\Psi}_{\Psi} & = & \mathtt{Shift}[\ulcorner\Psi\urcorner, \ulcorner\Psi\urcorner](\mathtt{Id}[\ulcorner\Psi\urcorner, \ulcorner\Psi\urcorner]) \\
\ulcorner \cdot \urcorner^{\Phi}_{\Psi, x:\mathbf{a}} & = & \mathtt{Shift}[\ulcorner\Psi, x:\mathbf{a}\urcorner, \ulcorner\Phi\urcorner] \\
 & & \quad (\mathtt{Suc}[\ulcorner\Psi, x:\mathbf{a}\urcorner, \ulcorner\Phi\urcorner]\ulcorner\cdot\urcorner^{\Phi}_{\Psi}) \\
\ulcorner \sigma, x/M \urcorner^{\Phi, x:A}_{\Psi} & = & \mathtt{Dot}[\ulcorner\Psi\urcorner, \ulcorner\Phi\urcorner, \mathbf{a}](\ulcorner\sigma\urcorner^{\Phi}_{\Psi}, \ulcorner M\urcorner_{\Phi\vdash\mathbf{a}})
\end{array}$$

Our Babybel prototype (see also our examples from the beginning) exploits the power GADTs to model context relations and relies on OCaml's type reconstruction engine [Garrigue and Rémy, 2013] to infer the omitted indices.

$$
\begin{array}{llcl}
\text{SF Patterns:} & \ulcorner \lambda\, x.R \urcorner^{\Gamma}_{\Psi \vdash \mathbf{a} \to A} & = & \texttt{Lam[_,_,_]}\ulcorner R \urcorner^{\Gamma}_{\Psi,\mathbf{a} \vdash A} \\
& \ulcorner \{R\} \urcorner^{\Gamma}_{\Psi \vdash \Box A} & = & \texttt{Box[_,_]}\ulcorner R \urcorner^{\Gamma}_{\cdot \vdash A} \\
& \ulcorner x \urcorner^{\cdot}_{\Psi \vdash \mathbf{a}} & = & \texttt{Var[_,_]}\ulcorner x \urcorner^{p}_{\Psi} \\
& \ulcorner \mathbf{c}\, \overrightarrow{R} \urcorner^{\Gamma}_{\Psi \vdash \mathbf{a}} & = & \texttt{C[_,_,_]}\ulcorner \overrightarrow{R} \urcorner^{\Gamma}_{\Psi \vdash A \downarrow \mathbf{a}} \\
& & & \text{with } \Sigma(\mathbf{c}) = A \\
& \ulcorner \texttt{'u} \urcorner^{u : \ulcorner \Psi \vdash A \urcorner}_{\Psi \vdash A} & = & u \\
& \ulcorner \texttt{\#x} \urcorner^{x : \ulcorner \Psi \vdash A \urcorner}_{\Psi \vdash \mathbf{a}} & = & \texttt{Var[_,_]}x \\
& \ulcorner \texttt{\#\#x} \urcorner^{x : \ulcorner \Psi, y : _ \vdash A \urcorner}_{\Psi, y : _ \vdash \mathbf{a}} & = & \texttt{Var[_,_]}(\texttt{Pop[_,_,_]}x) \\
& \ulcorner R\, \overrightarrow{R'} \urcorner^{\Gamma,\Gamma'}_{\Psi \vdash A \to B \downarrow \mathbf{a}} & = & \texttt{Cons[_,_,_,_]}(\ulcorner R \urcorner^{\Gamma}_{\Psi \vdash A}, \ulcorner \overrightarrow{R'} \urcorner^{\Gamma'}_{\Psi \vdash B \downarrow \mathbf{a}}) \\
& \ulcorner . \urcorner^{\cdot}_{\Psi \vdash \mathbf{a} \downarrow \mathbf{a}} & = & \texttt{Empty[_,_]} \\
\text{SF Variables:} & \ulcorner x \urcorner^{p}_{\Psi, x : \mathbf{a}} & = & \texttt{Top[_,_]} \\
& \ulcorner y \urcorner^{p}_{\Psi, x : \mathbf{b}} & = & \texttt{Pop[_,_,_]}\ulcorner y \urcorner^{p}_{\Psi}
\end{array}
$$

Figure 4.13 shows the translation of types, signatures and contexts for the computational language and the index language.

The translation of Core-ML expressions into Core-ML$^{\text{gadt}}$ directly follows the structure of programs in Core-ML and is type directed to fill in the required types for the Core-ML$^{\text{gadt}}$ representation. Figure 4.14 contains the translation.

Finally we show that the translation from Core-ML with contextual types into Core-ML$^{\text{gadt}}$ preserves types.

Theorem 4.8.1 (Main)**.**

1. If $\Gamma \vdash e \Leftarrow \tau$ then $\cdot; \ulcorner \Gamma \urcorner \vdash \ulcorner e \urcorner_{\Gamma \vdash \tau} : \ulcorner \tau \urcorner$.
2. If $\Gamma \vdash i \Rightarrow \tau$ then $\cdot; \ulcorner \Gamma \urcorner \vdash \ulcorner i \urcorner_{\Gamma \vdash \tau} : \ulcorner \tau \urcorner$.

Our result follows form mutual induction on the typing derivations and relies on several lemmas that deal with the other judgments and context lookups. Appendix B contains the proofs.

Translating Types:

$$
\begin{array}{rcl}
\ulcorner D \urcorner & = & D[] \\
\ulcorner \tau_1 \to \tau_2 \urcorner & = & \ulcorner \tau_1 \urcorner \to \ulcorner \tau_2 \urcorner \\
\ulcorner [\Psi \vdash A] \urcorner & = & \ulcorner A \urcorner \\
\ulcorner \overrightarrow{\tau} \urcorner & = & \ulcorner \tau_1 \urcorner, \ldots, \ulcorner \tau_n \urcorner
\end{array}
$$

Translating Signatures:

$$
\begin{array}{rcl}
\ulcorner \cdot \urcorner & = & \cdot \\
\ulcorner \Xi, D \urcorner & = & \ulcorner \Xi \urcorner, D : * \\
\ulcorner \Xi, C : \overrightarrow{\tau} \to D \urcorner & = & \ulcorner \Xi \urcorner, C : \ulcorner \overrightarrow{\tau} \urcorner \to D[]
\end{array}
$$

Translating Contexts:

$$
\begin{array}{rcl}
\ulcorner \cdot \urcorner & = & \cdot \\
\ulcorner \Gamma, x : \tau \urcorner & = & \ulcorner \Gamma \urcorner, x : \ulcorner \tau \urcorner
\end{array}
$$

Translating SF Signatures:

$$
\begin{array}{rcl}
\ulcorner \cdot \urcorner & = & \cdot \\
\ulcorner \Sigma, \mathbf{a} : \mathtt{type} \urcorner & = & \ulcorner \Sigma \urcorner, \mathbf{a} : * \\
\ulcorner \Sigma, \mathbf{c} : A \to \mathbf{a} \urcorner & = & \ulcorner \Sigma \urcorner, \mathbf{c} : \ulcorner A \urcorner \to \mathbf{a}
\end{array}
$$

Translating SF Contexts:

$$
\begin{array}{rcl}
\ulcorner \cdot \urcorner & = & \mathtt{nil}[] \\
\ulcorner \Psi, x : \mathbf{a} \urcorner & = & \mathtt{cons}[\ulcorner \Psi \urcorner, \mathbf{a}]
\end{array}
$$

Figure 4.13: Translating Types, Signatures, and Contexts

Translating expressions:

$$
\begin{array}{rcl}
\ulcorner x \urcorner_{\Gamma \vdash \tau} & = & x \\
\ulcorner C\,\overrightarrow{e} \urcorner_{\Gamma \vdash D} & = & C[\,]\,\ulcorner \overrightarrow{e} \urcorner_{\Gamma \vdash \overrightarrow{\tau}} \text{ with } \Xi(C) = \overrightarrow{\tau} \to D \\
\ulcorner \mathtt{fun}\, f(x) = e \urcorner_{\Gamma \vdash \tau_1 \to \tau_2} & = & \mathsf{fix}\, f : \ulcorner \tau_1 \to \tau_2 \urcorner = \lambda x\,.\, \ulcorner e \urcorner_{\Gamma, x : \tau_1 \vdash \tau_2} \\
\ulcorner i\, e \urcorner_{\Gamma \vdash \tau} & = & \ulcorner i \urcorner_{\Gamma \vdash \tau_1 \to \tau}\, \ulcorner e \urcorner_{\Gamma \vdash \tau_1} \\
 & & \text{with } \Gamma \vdash i \Rightarrow \tau_1 \to \tau \\
\ulcorner \mathtt{let}\, x = i \,\mathtt{in}\, e \urcorner_{\Gamma \vdash \tau} & = & \mathtt{let}\, x = \ulcorner i \urcorner_{\Gamma \vdash \tau_1} \,\mathtt{in}\, \ulcorner e \urcorner_{\Gamma, x : \tau_1 \vdash \tau} \\
 & & \text{with } \Gamma \vdash i \Rightarrow \tau_1 \\
\ulcorner \mathtt{match}\, i \,\mathtt{with}\, \overrightarrow{b} \urcorner_{\Gamma \vdash \tau} & = & \mathtt{match}\, \ulcorner i \urcorner_{\Gamma \vdash \tau_1} \,\mathtt{with}\, \ulcorner \overrightarrow{b} \urcorner_{\Gamma \vdash \tau_1 \to \tau} \\
 & & \text{with } \Gamma \vdash i \Rightarrow \tau_1 \\
\ulcorner e_1, \ldots, e_n \urcorner_{\Gamma \vdash \overrightarrow{\tau}} & = & \ulcorner e_1 \urcorner_{\Gamma \vdash \tau_1}, \ldots, \ulcorner e_n \urcorner_{\Gamma \vdash \tau_n} \\
\ulcorner [\hat{\Psi} \vdash M] \urcorner_{\Gamma \vdash [\Psi \vdash A]} & = & \ulcorner M \urcorner_{\Psi \vdash A} \\
\ulcorner \mathtt{cmatch}\, i \,\mathtt{with}\, \overrightarrow{c} \urcorner_{\Gamma \vdash \tau} & = & \mathtt{match}\, \ulcorner i \urcorner_{\Gamma \vdash [\Psi \vdash A]} \,\mathtt{with}\, \ulcorner \overrightarrow{c} \urcorner_{\Gamma \vdash [\Psi \vdash A] \to \tau} \\
 & & \text{with } \Gamma \vdash i \Rightarrow [\Psi \vdash A]
\end{array}
$$

Translating branches and patterns:

$$
\begin{array}{rcl}
\ulcorner pat \mapsto e \urcorner_{\Gamma \vdash \tau_1 \to \tau_2} & = & \ulcorner pat \urcorner^{\Gamma'}_{\Gamma \vdash \tau_1} \mapsto \ulcorner e \urcorner_{\Gamma, \Gamma' \vdash \tau_2} \\
\ulcorner x \urcorner^{x : \tau}_{\Gamma \vdash \tau} & = & x \\
\ulcorner C\,\overrightarrow{pat} \urcorner^{\Gamma'}_{\Gamma \vdash D} & = & C[]\,\ulcorner \overrightarrow{pat} \urcorner^{\Gamma'}_{\Gamma \vdash \overrightarrow{\tau}} \\
 & & \quad \text{with } \Xi(C) = \overrightarrow{\tau} \to D
\end{array}
$$

Translating branches:

$$
\ulcorner [\Psi \vdash R] \mapsto e \urcorner_{\Gamma \vdash [\Psi \vdash A] \to \tau} = \ulcorner R \urcorner^{\Gamma'}_{\Psi \vdash A} \mapsto \ulcorner e \urcorner_{\Gamma, \Gamma' \vdash \tau}
$$

Figure 4.14: Translating Computational Expressions

Lemma 2 (Ambient Context)**.** If $\Gamma(\mathsf{u}) = [\Psi \vdash \mathbf{a}]$ then $\ulcorner\Gamma\urcorner(u) = \mathtt{sftm}[\ulcorner\Psi\urcorner, \mathbf{a}]$.

Lemma 3 (Terms)**.**

1. If $\Gamma; \Psi \vdash M : A$ then $\cdot; \ulcorner\Gamma\urcorner \vdash \ulcorner M\urcorner_{\Psi\vdash A} : \ulcorner\Psi \vdash A\urcorner$.

2. If $\Gamma; \Psi \vdash \sigma : \Phi$ then $\cdot; \ulcorner\Gamma\urcorner \vdash \ulcorner\sigma\urcorner_{\Psi\vdash\Phi} : \ulcorner\Psi \vdash \Phi\urcorner$

Lemma 4 (Pat.)**.** If $\vdash pat : \tau \downarrow \Gamma$ then $\cdot \vdash \ulcorner pat\urcorner^{\Gamma}_{\Psi\vdash A} : \ulcorner\tau\urcorner \downarrow \Gamma$.

Lemma 5 (Ctx. Pat.)**.** If $\Psi \vdash R : A \downarrow \Gamma$ then $\cdot \vdash \ulcorner R\urcorner^{\Gamma}_{\Psi\vdash A} : \ulcorner\Psi \vdash A\urcorner \downarrow \Gamma$.

Given our set-up, the proofs for the are straightforward by induction on the structure of Γ and the typing derivation for Lemma 4, 5, 6.

4.9 A Proof of Concept Implementation

In this section, we describe the implementation[3] of Babybel which uses the ideas described in this chapter. One major difference is that Babybel translates OCaml programs that use syntax extensions for contextual SF types and terms and translates them into pure OCaml with GADTs. In fact, even our input OCaml programs may use GADTs to for example describe context relations on SF contexts (see also our examples from Sec.4.2).

The presence of GADTs in our source language also means that we can specify precise types for functions where we can quantify over contexts. Let's revisit some of the types of the programs that we wrote earlier in Sec.4.2:

- `rewrite:` γ`.` `[`γ `⊢` `tm]→[`γ `⊢` `tm]`: we implicitly quantify over all contexts γ and then we take a potentially open term and

[3]Available at `www.github.com/fferreira/babybel/`

return another term in the same context. These constraints imposed in the types are due to being able to index types with types thanks to GADTs.

- `get_path:` γ`.` `[`γ`,x:tm ⊢ tm]→path`: In this case we quantify over all contexts, but the input of the function is some term in a non-empty context. On the left of the turn-style our type mandates that there is at least one assumption of type `tm`. Notice how the return type is not inside a box(i.e.: square braces) because it is a regular OCaml recursive type and not a contextual type. We can mix and match as necessary.

- `conv:` γ δ`.(`γ`,` δ`) rel→[`γ` ⊢ tm]→[`δ` ⊢ ctm]`:
This final example shows that we can also use the contexts to index regular OCaml GADTs. In this function we are translating between terms in different representations. Naturally, their contexts contain assumptions of different type. To be able to translate between these different context representations, it is necessary to establish a relation between these contexts. So we need to define a special OCaml type (i.e.:`rel`) that relates variable to variable in each contexts.

By embedding the SF in OCaml using contextual types, we can combine and use the impure features of OCaml. Our example, in Section 4.2.1.2 takes advantage of them in our implementation of backtracking with exceptions. Additionally, performing I/O or using references works seamlessly in the prototype.

The presence of GADTs in our target language also makes the actual implementation of Babybel simpler than the theoretical description, as we take advantage of OCaml's built-in type reconstruction. In addition to GADTs, our implementation depends on several OCaml extensions. We use Attributes from Section 7.18 of the reference manual [Leroy et al., 2016a] and strings to embed the specification of our signature. We use

quoted strings from Section 7.20 to implement the boxes for terms ($\llparenthesis \ldots \rrparenthesis$) and patterns ($\llparenthesis \ldots \rrparenthesis_p$). All these appear as annotations in the internal Abstract Syntax Tree in the compiler implementation. To perform the translation (based on Section 4.8) we define a PPX rewriter as discussed in Section 23.1 of the OCaml manual. In our rewriter, we implement a parser for the framework SF and translate all the annotations using our embedding.

The Babybel prototype has been used to implement several case studies that expand from the examples at the beginning of this chapter. The Babybel distribution contains the following case studies besides some smaller programs used as a test suite.

- The example that removes syntactic sugar from Section 4.2.1.1.
- The example that finds the path to a variable from Section 4.2.1.2.
- The closure conversion example from Section 4.2.1.3.
- Inferring types for MiniML.
- An evaluator for an untyped MiniML.
- Comparing λ-terms up to alpha renaming.
- Translating the λ-calculus to a CPS form.

Currently, implementing these programs was straightforward and not having to think about the representation of binders was liberating. Along with the examples there are a couple of compiler phases, it would be interesting to implement a small compiler combining them and generating some code. Much remains to be done, but the prototype is already usable[4].

[4]Given that type-checking is done by OCaml on the translation, the error messages are far from perfect, but otherwise it is easy to use.

4.10 Related Work

Babybel and the syntactic framework SF are derived from ideas that originated in proof checkers based on the logical framework LF such as the Twelf system [Pfenning and Schürmann, 1999]. In the same category are the proof and programming languages Delphin [Poswolsky and Schürmann, 2009] and Beluga [Pientka and Cave, 2015] that offer a computational language on top of the LF. In many ways, this chapter and Babybel are a distillation of Beluga's ideas applied to a mainstream programming language. As a consequence, we have shown that we can get some of the benefits from Beluga at a much lower cost, since we do not have to build a stand-alone system or extend the compiler of an existing language to support contexts, contextual types and objects.

Our approach of embedding an LF specification language into a host language is in spirit related to the systems Abella [Gacek et al., 2012] and Hybrid [Felty and Momigliano, 2012] that use a two-level approach. In these systems we embed the specification language (typically hereditary harrop formulas) in first-order logic (or a variant of it). While our approach is similar in spirit, we focus on embedding SF specifications into a programming language instead of embedding it into a proof theory. Moreover, our embedding is type preserving by construction.

There are also many approaches and tools that specifically add support for writing programs that manipulate syntax with binders – even if they do not necessarily use HOAS. FreshML [Shinwell et al., 2003] and Cαml [Pottier, 2006] extend OCaml's data types with the ideas of names and binders from nominal logic [Pitts, 2003]. In these system, name generation is an effect, and if the user is not careful variables may extrude their scopes. Purity can be enforced by adding a proof system with a decision procedure that statically guarantees that no variable escapes its scope [Pottier, 2007]. This adds some complexity to the system. We feel that Babybel's contextual types offer a simpler formalism to deal with

bound variables. On the other hand, Babybel's approach does not try to model variables that do not have lexical scope, like top-level definitions. Another related language is Romeo [Stansifer and Wand, 2014] that uses ideas from Pure FreshML to represent binders. Where our system statically catches variables escaping their scope, the Romeo system either throws a run time exception or uses an SMT solver to prove the absence of scoping issues. The Hobbits for Haskell [Westbrook et al., 2011] system is implemented in a similar way to ours, using quasi-quoting but they use a different formalism based on the concepts of names and freshness. In general, parametric HOAS (PHOAS) [Chlipala, 2008] reuses the extensional (as in black box functions) function space of the language, while our approach introduces a distinction between an intensional and extensional function space. A particular approach is described in [Washburn and Weirich, 2008] where the authors propose a library that uses catamorphisms to compute over a parametric HOAS representation. This is a powerful approach but requires a different way of implementing recursive functions.

The separate intensional function space from SF allows us to model binding and supports pattern matching. The extentional function space allows us to write recursive functions. Following essentially work started by Despeyroux, Schuermann, and Pfenning [Despeyroux et al., 1997] and then later in the work on Beluga, we use a modality to embed syntactic objects characterized using the intensional function space into our programs. This is done while preserving the host language's (ML) extensional function space that allows for computation.

4.11 Conclusion

In this chapter, we describe the syntactic framework SF (a simply typed variant of the logical framework LF with the necessity modality from

S4) and explain the embedding of SF into a functional programming language using contextual types. This gives programmers the ability to write programs that manipulate abstract syntax trees with binders while knowing at type checking time that no variables extrude their scope. We also show how to translate the extended language back into a first order representation. For this, we use de Bruijn indices and GADTs to implement the SF in Core-ML^{gadt}. Important characteristics of the embedding are that it preserves the phase separation, making types (and thus contexts) erasable at run-time. This allows pattern matching to remain first-order and thus it is possible to compile with the traditional algorithms.

Finally, we describe Babybel an implementation of these ideas that embeds SF in OCaml using Contextual Types. The embedding is flexible enough that we can take advantage of the more powerful type system in OCaml to make the extension more powerful. We use GADTs in our examples to express more powerful invariants (e.g. that the translation preserves the context).

5 Contextual Types and Type Theory

5.1 Introduction

This is the last chapter[1], but in a sense it is not the end but the beginning of what we hope will be a fruitful research path. This chapter presents the first steps on answering an important question in this field, namely how to integrate contextual types and the logical framework LF with a reasoning language that provides full dependent types. Here, we develop a system that extends the ideas of Chapter 4 to a system that combines the logical framework LF [Harper et al., 1993] with an extended Martin-Löf style type theory [Martin-Löf, 1984] (MLTT). As in Babybel, the embedding uses contextual types [Nanevski et al., 2008] to mediate between both layers and allow the system to effortlessly represent open objects.

Furthermore, this system can be seen also as an extension of the theory of Beluga [Pientka and Cave, 2015] into full dependent types. The resulting system supports the definition of abstract syntax with binders and the use of substitutions. This simplifies proofs about systems with binders by providing for free the substitution lemma and lemmas about the equational theory of substitutions. As in Beluga, we mediate between specifications and computations via contextual types. However, unlike Beluga, we can embed and use computations directly in contextual objects and types, hence we allow the arbitrary mixing of specifications and computations. Moreover, dependent types allow for reasoning about proofs in addition to reasoning about LF specifications. This resulted in the Orca language, with its prototype implementation available at `https://github.com/orca-lang/orca`.

From an expressivity perspective, Orca and Beluga differ as when using

[1] This is the last chapter before the concluding remarks and conclusion.

the former the user can reason about specifications **and** computational functions. While the latter cannot reason about computations and instead needs to specify the computations in a relational style in LF.

The focus of this chapter is to describe the theory that is implemented by the Orca language (where the prototype already implements some extensions to the theory presented here) and to describe some ideas for an appropriate surface language for this theory. In particular, we will discuss how the prototype tries to alleviate the mediation between the two main syntactic categories (the specifications in LF and the reasoning language) by performing what we call a box inference pass on the surface language. Much remains to be done. As we discuss in the future work section a particularly important next step is to work on the meta-theory of this language. This will settle the answer to the question how to combine LF specifications with dependent types, while this chapter just hints at the answer.

5.2 Example: Translating Boolean Types

Let's consider a simple example of translating between two languages with a different representation of boolean types. Starting from a simply typed λ-calculus with boolean type:

$$
\begin{array}{rrcl}
\text{Types} & t & ::= & \texttt{bool} \mid t \rightarrow t' \\
\text{Expressions} & e & ::= & \texttt{tt} \mid \texttt{ff} \mid \texttt{if}\, e\, \texttt{then}\, e'\, \texttt{else}\, e'' \mid \lambda x^t.e \mid e\, e' \mid x \\
\text{Context} & \Gamma & ::= & \cdot \mid \Gamma, x : t
\end{array}
$$

We want to translate into a language that lacks booleans, but instead it

$\boxed{\Gamma \vdash e : t}$: e is of type t in context Γ

$$\frac{}{\Gamma \vdash \texttt{tt} : \texttt{bool}}\ \texttt{t-tt} \qquad \frac{}{\Gamma \vdash \texttt{ff} : \texttt{bool}}\ \texttt{t-ff}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e' : t \quad \Gamma \vdash e'' : t}{\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e'\, \texttt{else}\, e'' : t}\ \texttt{t-if}$$

$$\frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \lambda x^t.e : t \to t'}\ \texttt{t-lam} \qquad \frac{\Gamma \vdash e : t' \to t \quad \Gamma \vdash e' : t'}{\Gamma \vdash e\, e' : t}\ \texttt{t-app}$$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t}\ \texttt{t-var}$$

Figure 5.1: The Typing of the Source Language

has unit and sums:

$$\begin{array}{rrcl} \text{Types} & T & ::= & \texttt{1} \mid T + T' \mid T \to T' \\ \text{Expressions} & E & ::= & \texttt{()} \mid \texttt{inl}^T\, E \mid \texttt{inr}^T\, E \\ & & \mid & (\texttt{case}\, E\, \texttt{inl}\, x \mapsto E' \mid \texttt{inr}\, x \mapsto E'') \\ & & \mid & \lambda x^T.E \mid E\, E' \mid x \\ \text{Context} & \Delta & ::= & \cdot \mid \Delta, x : T \end{array}$$

The typing for both languages is as one would expect. For completeness, Figure 5.1 contains the typing rules for the source language and Figure 5.2 contains the typing for the target language.

The idea for the translation is that type `bool` from the source language can be represented by type $\texttt{1} + \texttt{1}$ in the target language, and that there is a mechanical way to translate expressions in a way that preserves types.

The translation of types is a function from types in the source language to types in the target language and it is defined as follows:

$$\begin{array}{rcl} [\![\texttt{bool}]\!] & = & \texttt{1} + \texttt{1} \\ [\![t \to t']\!] & = & [\![t]\!] \to [\![t']\!] \end{array}$$

$\boxed{\Delta \vdash E : T}$: E is of type T in context Δ

$$\frac{}{\Delta \vdash \texttt{()} : \texttt{1}}\ \texttt{T-unit} \qquad \frac{x : T \in \Delta}{\Delta \vdash x : T}\ \texttt{T-var}$$

$$\frac{\Delta \vdash E : T'}{\Delta \vdash \texttt{inl}^T\, E : T' + T}\ \texttt{T-inl} \qquad \frac{\Delta \vdash E : T'}{\Delta \vdash \texttt{inr}^T\, E : T + T'}\ \texttt{T-inr}$$

$$\frac{\Delta \vdash E : T' + T'' \quad \Delta, x : T' \vdash E' : T \quad \Delta, x : T'' \vdash E'' : T}{\Delta \vdash \texttt{case}\, E\, \texttt{inl}\, x \mapsto E' \mid \texttt{inr}\, x \mapsto E'' : T}\ \texttt{T-case}$$

$$\frac{\Delta, x : T \vdash E : T'}{\Delta \vdash \lambda x^T . E : T \to T'}\ \texttt{T-lam} \qquad \frac{\Delta \vdash E : T' \to T \quad \Delta \vdash E' : T'}{\Delta \vdash E\, E' : T}\ \texttt{T-app}$$

Figure 5.2: The Typing of the Target Language

Finally, the translation of expressions depends on the translation of types and it is as follows:

$$\begin{array}{rcl}
[\![\texttt{tt}]\!] & = & \texttt{inl}^{\texttt{unit}}\,\texttt{()} \\
[\![\texttt{ff}]\!] & = & \texttt{inr}^{\texttt{unit}}\,\texttt{()} \\
[\![\texttt{if}\, e\, \texttt{then}\, e'\, \texttt{else}\, e']\!] & = & \texttt{case}\,[\![e]\!]\,\texttt{inl}\ _ \mapsto [\![e']\!] \mid \texttt{inr}\ _ \mapsto [\![e'']\!] \\
[\![\lambda x^t . e]\!] & = & \lambda x^{[\![t]\!]} . [\![e]\!] \\
[\![e\, e']\!] & = & [\![e]\!]\,[\![e']\!] \\
[\![x]\!] & = & x
\end{array}$$

This translation can be implemented as a function that computes the expression in the target language. Moreover, it would be interesting to establish some properties of the translation. One such property is the fact that the translation is type preserving. Notice, that because both languages have different types, we say that the idea of type preservation is *up-to* the notion of translated types.

Let's consider how to formalize this in Orca in the most direct way. This is as opposed to formalizing an existing proof. The idea is to minimize

the lemmas and complexity of the formalization. To this end, let's specify the source and the target languages in an intrinsically typed way. That is, using dependent types in the logical framework LF to easily restrict the expressions to those that are well-typed.

```
spec s-tp : * where
| bool : s-tp
| arr : s-tp ↠ s-tp ↠ s-tp

spec s-exp : s-tp ↠ * where
| app : (s : s-tp) ↠ (t : s-tp) ↠
        s-exp (arr s t) ↠ s-exp s ↠ s-exp t
| lam : (s : s-tp) ↠ (t : s-tp) ↠
        (s-exp s ↠ s-exp t) ↠ s-exp (arr s t)
| tt : s-exp bool
| ff : s-exp bool
| if : (t : s-tp) ↠
      s-exp bool ↠ s-exp t ↠ s-exp t ↠ s-exp t
```

In this code, we define two new types families, first `s-tp` for types and for expressions `s-exp`. Notice how the expressions form a type family where expressions are indexed by their types (i.e.: the function from `s-tp` to `*` that represents the base kind for LF specifications). The declaration of constructors is straightforward even if the lack of implicit parameters makes it verbose. Specifically, applications take two expressions, the first one a function from `s` to `t` and a parameter of appropriate type to produce an expression of type `t`. The constructor `lam` builds an abstraction from an expression with one bound variable (using HOAS we represent the binder using the LF function space) to produce an expression of function type. Then the two values of boolean type, and finally expressions of boolean type are eliminated with `if` expressions. It is straightforward to see how these definitions follow from the typing rules in Figure 5.1. Notice how, for LF specifications (declared with **`spec`**), we use a special function space `(x : α) ↠ β` to indicate the LF function space. Also, as it is commonly done, for non dependent functions we use `α ↠ β` when α does not occur in β.

```
spec t-tp : ⋆ where
| tunit : t-tp
| tsum : t-tp ↠ t-tp ↠ t-tp
| tarr : t-tp ↠ t-tp ↠ t-tp

spec t-exp : t-tp ↠ ⋆ where
| tapp : (s : t-tp) ↠ (t : t-tp) ↠
         t-exp (tarr s t) ↠ t-exp s ↠ t-exp t
| tlam : (s : t-tp) ↠ (t : t-tp) ↠
         (t-exp s ↠ t-exp t) ↠ t-exp (tarr s t)
| tone : t-exp tunit
| tinl : (s : t-tp) ↠ (t : t-tp) ↠
         t-exp s ↠ t-exp (tsum s t)
| tinr : (s : t-tp) ↠ (t : t-tp) ↠
         t-exp t ↠ t-exp (tsum s t)
| tcase : (s : t-tp) ↠ (t : t-tp) ↠ (r : t-tp) ↠
          t-exp (tsum s t) ↠ (t-exp s ↠ t-exp r) ↠
                             (t-exp t ↠ t-exp r) ↠
          t-exp r
```

The target language is similar to the source language, and it is represented in a analogous way. Notice, how we model the binders in the branches of the case expression with the logical framework's function space.

The translation of types is represented with a function from source types to target types. We use the keyword **def** to introduce a top level function definition with pattern matching. It is implemented as follows:

```
def tran-tp : (⊢ s-tp) → (⊢ t-tp) where
| bool ⇒ tsum tunit tunit
| (arr s t) ⇒ tarr (tran-tp s) (tran-tp t)
```

First notice the type specification after the colon in the first line where the turn-styles specify that these are specification types and that they are closed types (after all these two languages types have no binders in them). Then the implementation of the function goes by pattern matching on the argument and producing the appropriate target type according to the definition. In Orca, definitions are elaborated in a type directed way to allow the user to omit the boxes when possible. This is explained further

when we describe the prototype in Section 5.5 and we show the result of the reconstruction of boxes for the translation function in Figure 5.17.

We proceed similarly to implement the translation of expressions. However, let's consider the type for the translation function first. A first idea could be:

```
def tran : (st : ⊢ s-tp) → (⊢ s-exp st) →
              (⊢ t-exp (tran-tp st)) where
...
```

This type indicates that given a closed source expression of type `st` we can produce a closed target expression whose type will be the translation of type `st`. Notice how this is a place we interleave specifications and computations as the resulting expression's type is the result of actually calling the function `tran-tp` on the source expression. In a system without computation in types like Beluga, we need to implement this using a relation and this requires a couple of lemmas to show that the relation is actually a function. As a reference Appendix C contains a Beluga implementation of this example. However, this type is not strong enough as the recursive calls may go under binders. Thus we need to be able to translate open expressions. We need to generalize this type to arbitrary contexts and add a context relation that expresses that the source context and target context grow in unison. Context relations are required because so far, Orca contexts can contain assumptions of any kind. For this we define a binary relation on contexts as follows:

```
data rel: ctx → ctx → set where
| empty : rel ∅ ∅
| cons : (g h : ctx) (t : ⊢ s-tp) →
         rel g h →
         rel (g, x: s-exp t)
             (h, y: t-exp (tran-tp t))
```

In this relation the `empty` constructor states that empty contexts are related, and the `cons` constructor shows that if we have two related contexts we can extend both contexts with a fresh related assumption.

With all this in place, we can write the translation function:

```
def tran : (g h : ctx) → rel g h →
           (st : ⊢ s-tp) → (g ⊢ s-tm st) →
           (h ⊢ t-tm (tran-tp st))
where
| g h r t (app s[^] .t m n) ⇒
    tapp (tran-tp s) (tran-tp t)
         (tran g h r (arr s t) m) (tran g h r s n)
| g h r (arr s[^] t[^]) (lam .s .t m) ⇒
    tlam (tran-tp s) (tran-tp t)
         (λx. tran (g,x:s-tm s) (h,x:t-tm (tran-tp s))
                   (cons g h s r) t (m x))
| g h r bool tt ⇒
    tinl tunit tunit tone
| g h r bool ff ⇒
    tinr tunit tunit tone
| g h r t (if .t b e1 e2) ⇒
    tcase tunit tunit (tran-tp t) (tran g h r bool b)
                      (λx. tran g h r t e1)
                      (λx. tran g h r t e2)
(* when translating variables it is either the top variable *)
| ._ ._ (cons g h .t r) t (g, x: s-tm t :> x) ⇒
    (h, x: t-tm (tran-tp t) :> x)
(* or a variable deeper in the context (hence the shift) *)
| ._ ._ (cons g h t r) s (v[^1]) ⇒
    tran g h r s v
```

The function `tran` has type:

```
(g h : ctx) → rel g h → (st : ⊢ s-tp) → (g ⊢ s-exp st) →
     (h ⊢ t-exp (tran-tp st))
```

This function transforms expressions of type: `(g ⊢ s-exp st)` into expressions of type `(h ⊢ t-exp (tran-tp st))`, where it computes the type of the source expression by embedding the computation `tran-tp st` in the resulting type. The computation proceeds by pattern matching, in the patterns we write `._` for inaccessible patterns that the implementation fills in automatically. Inaccessible patterns are a concept from Agda that simplifies implementing pattern matching, their value is forced by the other variables, for example in the pattern for lambda expressions `g h r (arr s[^] t[^]) (lam ._ ._ m)` the two inaccessible patterns are forced by

the arrow type to be `s` and `t` respectively. In the same pattern, we want to indicate that the types of expressions are closed, we annotate `s` and `t` with the empty substitution (i.e.:`[^]`) that prevents them from using variables from the context. We build contextual types by combining a term with a context (e.g.: `g ⊢ s-exp st`), for contextual terms we use a different syntax to make it easier to disambiguate between types and terms, contexts and terms are combined in the following way: `g, x: s-tm t :> x`.

This is a simple example, but even in its simplicity it showcases the power of the Orca language. Even if the Orca prototype does not offer many of the features present in Beluga, this example shows a hint to what is coming in the future. In the next section, we peek behind the curtains into Orca's type theory.

5.3 Orca's Core Calculus

Orca's core calculus is composed of two separate languages, a *reasoning* and *computation* language on one side and a *specification* language on the other side. We refer to the former as the computation language when we want to emphasize that we can write programs with it, and as the reasoning language when trying to emphasize that we can write proofs with it.

The syntax for the reasoning language is a fully dependently typed theory, extended with contextual types and terms (in red in the syntax), it is exactly as one would expect a type theory to be. As is the case with Agda [Norell, 2007] and Idris [Brady, 2013], we do not have recursion in the calculus, but we do top-level recursive definitions that allow for defining inductive functions. We say contextual objects are boxed specifications as a reference to the contextual modality in contextual modal type theory [Nanevski et al., 2008]. The type for contexts is a simplification that avoids discussing classifiers for contexts (i.e.: context schemas).

This is a limitation of the current presentation of the theory, and it makes discussing totality and coverage difficult. In short, it would be necessary to add schemas to be able to describe a coverage checking algorithm.

Reasoning Language				
Terms	E, S, T	$::=$	$\texttt{set}_n$	A universe of level n
		$\mid$	$(x : S) \to T$	A function type
		$\mid$	$\lambda x.E$	An abstraction
		$\mid$	$E\, E'$	An application
		$\mid$	x	A variable
		$\mid$	$\mathbf{c}$	A constant
		$\mid$	$[\hat{\Psi} \triangleright M]$	A contextual term
		$\mid$	$[\Psi \vdash \alpha]$	A contextual type
		$\mid$	$\texttt{ctx}$	The type of contexts

Conversely, the specification language is a version of the logical framework LF, but instead of meta-variables, it supports a direct embedding of computations together with a substitution, to unbox the result of computation into a specification object. Of course, the computation needs to produce a result of specification type.

Specification Language

Kinds	K	$::=$	$\star$	The base kind
		$\mid$	$(\widehat{x}:\alpha) \twoheadrightarrow K$	A kind family
Families	α, β	$::=$	$\mathbf{a}$	A base type
		$\mid$	$(\widehat{x}:\alpha) \twoheadrightarrow \beta$	A type family
		$\mid$	$\alpha\, M$	A family instance
Objects	M, N	$::=$	$\widehat{\lambda} x.M$	An abstraction
		$\mid$	$M \,'\, N$	An application
		$\mid$	$\widehat{x}$	A variable
		$\mid$	$\widehat{\mathbf{c}}$	A constant
		$\mid$	$\lfloor E \rfloor_\sigma$	An unboxed computation
Substitutions	σ	$::=$	$\wedge$	An empty substitution
		$\mid$	$\mathtt{id}_n$	An identity substitution, with weakening
		$\mid$	$\sigma; \widehat{x} := M$	A substitution extension
Context	Ψ	$::=$	$\emptyset$	Empty context
		$\mid$	$\lfloor x \rfloor$	An unboxed context variable
		$\mid$	$\Psi, \widehat{x}:\alpha$	A context extension
Erased Context	$\hat{\Psi}$	$::=$	$\emptyset$	An empty context
		$\mid$	$\lfloor x \rfloor$	An unboxed context variable
		$\mid$	$\hat{\Psi}, \widehat{x}$	A variable declaration

Finally, we define contexts for computational variables, that as in Chapter 4 will be the meta-variables for incomplete objects, and a signature to define types and constants for computation terms and for the specification language.

Signature and Computational Context

Context	Γ	$::=$	$\cdot$	Empty context
		$\mid$	$\Gamma, x : T$	An assumption
Signature	Σ	$::=$	$\cdot$	Empty signature
		$\mid$	$\Sigma, \mathbf{c} : T$	A comp. constant
		$\mid$	$\Sigma, \mathbf{a} : K$	A spec. type
		$\mid$	$\sigma, \widehat{\mathbf{c}} : \alpha$	A spec. constant

We present a Martin-Löf style type theory with an infinite hierarchy of universes extended with contextual types $[\Psi \vdash \alpha]$ which represent a specification type α in an open context Ψ. Specification types are classified by a single universe $\star$, together with an intensional function space $(\widehat{x} : \alpha) \twoheadrightarrow \beta$ and constants. Substitutions can be an empty substitution $\wedge$ which weakens closed objects, an identity substitution $\mathtt{id}_n$ that weakens the context with n elements (we write $\mathtt{id} = \mathtt{id}_0$ as syntactic sugar), or a substitution extended with an object for a variable $\sigma; \widehat{x} := M$. Contexts are either empty $\emptyset$ or a context extended with a new assumption $\Psi, \widehat{x} : A$.

Type theory terms and specification objects are typed with two different judgments $\Gamma \vdash E : T$ and $\Gamma; \Psi \vdash M : \beta$, respectively. For instance, we type the type theory functions and specifications functions in the following way:

$$\frac{\Gamma, x : S \vdash E : T}{\Gamma \vdash \lambda x.E : (x : S) \to T}\ \texttt{t-fun} \qquad \frac{\Gamma; \Psi, \widehat{x} : \alpha \vdash M : \beta}{\Gamma; \Psi \vdash \widehat{\lambda} x.M : (\widehat{x} : \alpha) \twoheadrightarrow \beta}\ \texttt{s-lam}$$

The two function spaces differ by the contexts they act on and that the computational function space is extensional (i.e.: we can compute with these functions) while the specification function space is intensional (i.e.: we can inspect these functions with pattern matching) in the sense of Pfenning [2001]. Type theory λ-abstractions introduce variables in the computational context Γ while specification $\widehat{\lambda}$-abstractions use the

Substitution in the reasoning language

$$
\begin{array}{lcl}
\{\theta\}\,\mathtt{set}_n & = & \mathtt{set}_n \\
\{\theta\}\,(x : S) \to T & = & (x : \{\theta\}\,S) \to \{\theta; x := x\}\,T \text{ for } x \text{ free in } \theta \\
\{\theta\}\,\lambda x.E & = & \lambda x.(\{\theta; x := x\}\,E) \text{ for } x \text{ free in } \theta \\
\{\theta\}\,E\,E' & = & (\{\theta\}\,E)\,(\{\theta\}\,E') \\
\{\theta\}\,x & = & E \text{ if } x := E \text{ is in } \theta \\
\{\theta\}\,x & = & x \text{ if } x \text{ is not in } \theta \\
\{\theta\}\,\mathbf{c} & = & \mathbf{c} \\
\{\theta\}\,[\hat{\Psi} \triangleright M] & = & [\{\theta\}\,\hat{\Psi} \triangleright \{\theta\}\,M] \\
\{\theta\}\,[\Psi \vdash \alpha] & = & [\{\theta\}\,\Psi \vdash \{\theta\}\,\alpha]
\end{array}
$$

Figure 5.3: Computation Substitution

specification context Ψ. The β-rule for each λ-abstraction uses its own substitution operation for its corresponding class of variables denoted respectively $\{\theta\}E$ and $[\sigma]M$. The definition for the substitution operations is presented in Figures 5.3, 5.4, 5.5, and 5.6. This is in contrast to $\lfloor E \rfloor_\sigma$ for the closure that embeds a computation of boxed type into a specification and is defined by the following introduction rule:

$$\frac{\Gamma; \Psi \vdash \sigma : \Phi \quad \Gamma \vdash E : [\Phi \vdash \alpha]}{\Gamma; \Psi \vdash \lfloor E \rfloor_\sigma : [\sigma]\alpha}\ \texttt{s-clo}$$

In addition to the description of the term calculus we present a prototype implementation[2] for our type theory which supplements the calculus with recursive functions and Agda-style dependent pattern matching [Norell, 2007] extended to allow matching on specifications including specification-level λ-abstractions and thus abstracting over binders.

In conclusion, we present a theory that allows for embedding contextual LF specifications into a fully dependently typed language that simplifies proofs about structures with syntactic binders (such as pro-

[2] Available at: `http://github.com/orca-lang/orca`

Substitution in objects of the specification language

$$
\begin{array}{lcl}
\{\theta\}\,\widehat{\lambda} x.M & = & \widehat{\lambda} x.(\{\theta\}\,M) \\
\{\theta\}\,M \mathbin{'} N & = & (\{\theta\}\,M) \mathbin{'} (\{\theta\}\,N) \\
\{\theta\}\,\widehat{x} & = & \widehat{x} \\
\{\theta\}\,\widehat{\mathbf{c}} & = & \widehat{\mathbf{c}} \\
\{\theta\}\,\lfloor E \rfloor_{\sigma} & = & \lfloor (\{\theta\}\,E) \rfloor_{(\{\theta\}\,\sigma)}
\end{array}
$$

Substitution in families of the specification language

$$
\begin{array}{lcl}
\{\theta\}\,\mathbf{a} & = & \mathbf{a} \\
\{\theta\}\,(\widehat{x}\,{:}\,\alpha) \twoheadrightarrow \beta & = & (\widehat{x}\,{:}\,(\{\theta\}\,\alpha)) \twoheadrightarrow (\{\theta\}\,\beta) \\
\{\theta\}\,\alpha\,M & = & (\{\theta\}\,\alpha)\,(\{\theta\}\,M)
\end{array}
$$

Figure 5.4: Computation Substitution in Specifications

Substitution in the specification contexts

$$
\begin{array}{lcl}
\{\theta\}\,\emptyset & = & \emptyset \\
\{\theta\}\,\Psi, \widehat{x}\,{:}\,\alpha & = & (\{\theta\}\,\Psi), \widehat{x}\,{:}\,(\{\theta\}\,\alpha)
\end{array}
$$

Substitution in the specification substitutions

$$
\begin{array}{lcl}
\{\theta\}\,\wedge & = & \wedge \\
\{\theta\}\,\mathtt{id}_n & = & \mathtt{id}_n \\
\{\theta\}\,\sigma; \widehat{x} := M & = & (\{\theta\}\,\sigma); \widehat{x} := (\{\theta\}\,M)
\end{array}
$$

Figure 5.5: Computation Substitution in Contexts and Substitutions

LF Substitution in objects

$$
\begin{array}{lcl}
[\sigma]\,\widehat{\lambda}x.M & = & \widehat{\lambda}x.([\sigma;\widehat{x} := \widehat{x}]\,M) \\
[\sigma]\,M \mathbin{'} N & = & ([\sigma]\,M) \mathbin{'} ([\sigma]\,N) \\
[\sigma]\,\widehat{x} & = & M \qquad \text{if } \widehat{x} := M \text{ is in } \sigma \\
[\sigma]\,\widehat{x} & = & \widehat{x} \qquad \text{if } \widehat{x} \text{ is not in } \sigma \\
[\sigma]\,\widehat{\mathbf{c}} & = & \widehat{\mathbf{c}} \\
[\sigma]\,\lfloor E \rfloor_{\sigma'} & = & \lfloor E \rfloor_{([\sigma]\,\sigma')}
\end{array}
$$

LF Substitution in families

$$
\begin{array}{lcl}
[\sigma]\,\mathbf{a} & = & \mathbf{a} \\
[\sigma]\,(\widehat{x}:\alpha) \twoheadrightarrow \beta & = & (\widehat{x}:([\sigma]\,\alpha)) \twoheadrightarrow ([\sigma;\widehat{x} := \widehat{x}]\,\beta) \text{ with } \widehat{x} \text{ fresh in } \sigma \\
[\sigma]\,\alpha\,M & = & ([\sigma]\,\alpha)\,([\sigma]\,M)
\end{array}
$$

LF Substitution in kinds

$$
\begin{array}{lcl}
[\sigma]\,\star & = & \star \\
[\sigma]\,(\widehat{x}:\alpha) \twoheadrightarrow K & = & (\widehat{x}:[\sigma]\,\alpha) \twoheadrightarrow ([\sigma;\widehat{x} := \widehat{x}]\,K) \text{ with } \widehat{x} \text{ fresh in } \sigma
\end{array}
$$

Figure 5.6: Specification Substitution

gramming languages and logics). Moreover, we have a prototype, the Orca system, in which we implement some example proofs.

5.3.1 The Typing of Programs

Type checking depends on these mutually dependent typing judgments:

- $\boxed{\Gamma \vdash E : T}$: E is of type T in context Γ.
- $\boxed{\Gamma \vdash \Psi \; \texttt{is ctx}}$: Ψ is a valid specification context in ctx. Γ.
- $\boxed{\Gamma;\Psi \vdash \alpha \; \texttt{is kind}}$: α is a syntactic type in ctx. Γ and spec. ctx. Ψ.
- $\boxed{\Gamma;\Psi \vdash \alpha : K}$: α is a syntactic type in ctx. Γ and spec. ctx. Ψ.

$\boxed{\Gamma \vdash E : T}$: E is of type T in context Γ

$$\frac{}{\Gamma \vdash \mathtt{set}_n : \mathtt{set}_{(n+1)}}\ \mathtt{t\text{-}set} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \mathtt{t\text{-}var} \qquad \frac{\mathbf{c} : T \in \Sigma}{\Gamma \vdash \mathbf{c} : T}\ \mathtt{t\text{-}con}$$

$$\frac{\Gamma \vdash S : \mathtt{set}_n \quad \Gamma, x : S \vdash T : \mathtt{set}_0}{\Gamma \vdash (x : S) \to T : \mathtt{set}_0}\ \mathtt{t\text{-}pi\text{-}1}$$

$$\frac{\Gamma \vdash S : \mathtt{set}_n \quad \Gamma, x : S \vdash T : \mathtt{set}_{(m+1)}}{\Gamma \vdash (x : S) \to T : \mathtt{set}_{(\max\ n\ (m+1))}}\ \mathtt{t\text{-}pi\text{-}2}$$

$$\frac{\Gamma, x : S \vdash E : T}{\Gamma \vdash \lambda x.E : (x : S) \to T}\ \mathtt{t\text{-}fun} \qquad \frac{\Gamma \vdash E : (x : S) \to T \quad \Gamma \vdash E' : S}{\Gamma \vdash E\, E' : \{x := E'\}T}\ \mathtt{t\text{-}app}$$

$$\frac{\Gamma \vdash E : S \quad \Gamma \vdash S \equiv T}{\Gamma \vdash E : T}\ \mathtt{t\text{-}conv}$$

$$\frac{\Gamma \vdash \Psi\ \mathtt{is\ ctx} \quad \Gamma; \Psi \vdash M : \alpha}{\Gamma \vdash [\hat{\Psi} \triangleright M] : [\Psi \vdash \alpha]}\ \mathtt{t\text{-}box}^*$$

$$\frac{\Gamma \vdash \Psi\ \mathtt{is\ ctx} \quad \Gamma; \Psi \vdash \alpha : \star}{\Gamma \vdash [\Psi \vdash \alpha] : \mathtt{set}_0}\ \mathtt{t\text{-}spec}^*$$

Figure 5.7: Typing for Computations

- $\boxed{\Gamma; \Psi \vdash M : \alpha}$: M is of type α in ctx. Γ and spec. ctx. Ψ.
- $\boxed{\Gamma; \Psi \vdash \sigma : \Phi}$: σ transports types from spec. context Φ to ctx. Ψ.

and these definitional equality judgments:

- $\boxed{\Gamma \vdash E \equiv E' : T}$: E is equal to E' at type T in context Γ.
- $\boxed{\Gamma; \Psi \vdash M \equiv N : \alpha}$: M is equal to N at type α in contexts Γ and Ψ.
- $\boxed{\Gamma; \Psi \vdash K \equiv K'}$: K is equal to K' in contexts Γ and Ψ.
- $\boxed{\Gamma; \Psi \vdash \alpha \equiv \beta : K}$: α is equal to β at kind K in contexts Γ and Ψ.

- $\boxed{\Gamma \vdash \Psi \equiv \Phi : \texttt{ctx}}$: Ψ is equal to Φ in context Γ.
- $\boxed{\Gamma \vdash \hat{\Psi} \equiv \hat{\Phi} : \texttt{ctx}}$: Erased context $\hat{\Psi}$ is equal to $\hat{\Phi}$ in context Γ.

The typing for computations appears in Figure 5.7 and the typing for specifications appears in Figures 5.8 and 5.9. While rules for computational equality appear in Figure 5.10 and for specifications in Figures 5.11, 5.12, and 5.13. Rules for equivalences and congruence rules are omitted as they are as expected.

Most rules are as expected, however the interesting ones appear with a star in their names. These rules, reproduced below deal with the embedding of specifications in computations and vice versa.

$$\frac{\Gamma \vdash \Psi \;\texttt{is ctx} \quad \Gamma; \Psi \vdash M : \alpha}{\Gamma \vdash [\hat{\Psi} \triangleright M] : [\Psi \vdash \alpha]} \;\texttt{t-box}^{*}$$

$$\frac{\Gamma; \Psi \vdash \sigma : \Phi \quad \Gamma \vdash E : [\Phi \vdash \alpha]}{\Gamma; \Psi \vdash \lfloor E \rfloor_{\sigma} : [\sigma]\, \alpha} \;\texttt{s-unbox}^{*}$$

The rule `t-box` embeds an object together with its context in a computation (of contextual type). And the `s-unbox` rule allows specifications to refer to computations, in particular this is essential to represent incomplete terms where the role of meta-variables is taken by computational variables inside an unbox construction.

5.3.2 Equality

Due to the presence of fully dependent types Orca's computational language does not have a distinction between type checking and evaluation (as evaluation may happen at type check time). As usual, the conversion rule (rule `t-conv` in Figure 5.7) is where there may be some computation to compare two types, that may not be yet in normal form. Equality (and computation), as typing, is shown in several judgments, one for the

$\boxed{\Gamma \vdash \Psi \texttt{ is ctx}}$: Ψ is a valid specification context in ctx. Γ.

$$\frac{}{\Gamma \vdash \emptyset \texttt{ is ctx}}\ \texttt{c-empty} \qquad \frac{\Gamma \vdash \Psi \texttt{ is ctx} \quad \Gamma; \Psi \vdash \alpha : \star}{\Gamma \vdash \Psi, \widehat{x} : \alpha \texttt{ is ctx}}\ \texttt{c-hyp}$$

$$\frac{x : \texttt{ctx} \in \Gamma}{\Gamma \vdash \lfloor x \rfloor \texttt{ is ctx}}\ \texttt{c-ctx-var}$$

$\boxed{\Gamma \vdash \hat{\Psi} \texttt{ is erased}}$: $\hat{\Psi}$ is a valid erased context in ctx. Γ.

$$\frac{}{\Gamma \vdash \emptyset \texttt{ is erased}}\ \texttt{cp-empty} \qquad \frac{\Gamma \vdash \Psi \texttt{ is erased}}{\Gamma \vdash \Psi, \widehat{x} \texttt{ is erased}}\ \texttt{cp-hyp}$$

$$\frac{x : \texttt{ctx} \in \Gamma}{\Gamma \vdash \lfloor x \rfloor \texttt{ is erased}}\ \texttt{cp-ctx-var}$$

$\boxed{\Gamma; \Psi \vdash K \texttt{ is kind}}$: α is a syntactic kind in ctx. Γ and spec. ctx. Ψ.

$$\frac{}{\Gamma; \Psi \vdash \star \texttt{ is kind}}\ \texttt{s-kind}$$

$$\frac{\Gamma; \Psi \vdash \alpha : \star \quad \Gamma; \Psi, x : \alpha \vdash K \texttt{ is kind}}{\Gamma; \Psi \vdash (\widehat{x} : \alpha) \twoheadrightarrow K \texttt{ is kind}}\ \texttt{s-pi-k}$$

$\boxed{\Gamma; \Psi \vdash \alpha : K}$: α is a syntactic type in ctx. Γ and spec. ctx. Ψ.

$$\frac{\mathbf{a} : K \in \Sigma}{\Gamma; \Psi \vdash \mathbf{a} : K}\ \texttt{s-base} \qquad \frac{\Gamma; \Psi \vdash \alpha : \star \quad \Gamma; \Psi, \widehat{x} : \alpha \vdash \beta : \star}{\Gamma; \Psi \vdash (\widehat{x} : \alpha) \twoheadrightarrow \beta : \star}\ \texttt{s-fam}$$

$$\frac{\Gamma; \Psi \vdash \alpha : (\widehat{x} : \beta) \twoheadrightarrow K \quad \Gamma; \Psi \vdash M : \beta}{\Gamma; \Psi \vdash \alpha\, M : [x := M]K}\ \texttt{s-ins}$$

$$\frac{\Gamma; \Psi \vdash \alpha : K' \quad \Gamma; \Psi \vdash K \equiv K'}{\Gamma; \Psi \vdash \alpha : K}\ \texttt{s-eqk}$$

Figure 5.8: Typing Rules for Specifications (I)

$\boxed{\Gamma;\Psi \vdash M : \alpha}$: M is of type α in ctx. Γ and spec. ctx. Ψ.

$$\frac{\Gamma;\Psi, \widehat{x} : \alpha \vdash M : \beta}{\Gamma;\Psi \vdash \widehat{\lambda} x.M : (\widehat{x} : \alpha) \twoheadrightarrow \beta}\ \texttt{s-lam}$$

$$\frac{\Gamma;\Psi \vdash M : (\widehat{x} : \alpha) \twoheadrightarrow \beta \quad \Gamma;\Psi \vdash N : \alpha}{\Gamma;\Psi \vdash M\ '\ N : [x := N]\beta}\ \texttt{s-app}$$

$$\frac{\widehat{x} : \alpha \in \Psi}{\Gamma;\Psi \vdash \widehat{x} : \alpha}\ \texttt{s-var} \qquad \frac{\widehat{\mathbf{c}} : \alpha \in \Sigma}{\Gamma;\Psi \vdash \widehat{\mathbf{c}} : \alpha}\ \texttt{s-con}$$

$$\frac{\Gamma;\Psi \vdash \sigma : \Phi \quad \Gamma \vdash E : [\Phi \vdash \alpha]}{\Gamma;\Psi \vdash \lfloor E \rfloor_\sigma : [\sigma]\,\alpha}\ \texttt{s-unbox}^*$$

$$\frac{\Gamma;\Psi \vdash M : \alpha' \quad \Gamma;\Psi \vdash \alpha \equiv \alpha' : \star}{\Gamma;\Psi \vdash M : \alpha}\ \texttt{s-eq}$$

$\boxed{\Gamma;\Psi \vdash \sigma : \Phi}$: σ transports types from spec. context Φ to ctx. Ψ.

$$\frac{}{\Gamma;\Psi \vdash^\wedge\ : \emptyset}\ \texttt{s-empty} \qquad \frac{|\Psi'| = n}{\Gamma;\Psi,\Psi' \vdash \texttt{id}_n : \Psi}\ \texttt{s-id}$$

$$\frac{\Gamma;\Psi \vdash \sigma : \Phi \quad \Gamma;\Psi \vdash M : [\sigma]\alpha}{\Gamma;\Psi \vdash \sigma;\widehat{x} :=\ M : \Phi, \widehat{x} : \alpha}\ \texttt{s-ex}$$

Figure 5.9: Typing Rules for Specifications (II)

$\boxed{\Gamma \vdash E \equiv E' : T}$: E is equal to E' at type T in context Γ.

$$\frac{\Gamma, x : S \vdash E : T \quad \Gamma \vdash E' : S}{\Gamma \vdash (\lambda x.E)\, E' \equiv \{x := E'\}\, E : \{x := E'\}\, T}\ \texttt{e-beta}$$

$$\frac{\Gamma \vdash E : [\Psi \vdash \alpha]}{\Gamma \vdash [\hat{\Psi} \triangleright \lfloor E \rfloor_{\texttt{id}}] \equiv E : [\Psi \vdash \alpha]}\ \texttt{e-box-unbox}$$

$$\frac{\Gamma \vdash \hat{\Psi} \equiv \hat{\Phi} : \texttt{ctx} \quad \Gamma ; \Psi \vdash M \equiv M' : \alpha}{\Gamma \vdash [\hat{\Psi} \triangleright M] \equiv [\hat{\Phi} \triangleright M'] : [\Psi \vdash \alpha]}\ \texttt{e-box-1}$$

$$\frac{\Gamma \vdash \Psi \equiv \Phi : \texttt{ctx} \quad \Gamma ; \Psi \vdash \alpha \equiv \beta : \star}{\Gamma \vdash [\Psi \vdash \alpha] \equiv [\Phi \vdash \beta] : \texttt{set}_0}\ \texttt{e-box-2}$$

Figure 5.10: Equality Rules for Computations

$\boxed{\Gamma ; \Psi \vdash M \equiv N : \alpha}$: M is equal to N at type α in contexts Γ and Ψ.

$$\frac{\Gamma ; \Psi, \widehat{x} : \alpha \vdash M : \beta \quad \Gamma ; \Psi \vdash N : \alpha}{\Gamma ; \Psi \vdash (\widehat{\lambda} x.M)\ '\ N \equiv (\{\hat{x} := N\} M) : (\{\hat{x} := N\} \beta)}\ \texttt{es-beta}$$

$$\frac{\Gamma ; \Psi \vdash M : (\widehat{x} : \alpha) \twoheadrightarrow \beta}{\Gamma ; \Psi \vdash \widehat{\lambda} x.M\, x \equiv M : (\widehat{x} : \alpha) \twoheadrightarrow \beta}\ \texttt{es-eta}$$

$$\frac{\Gamma ; \Psi \vdash \sigma : \Phi \quad \Gamma ; \Phi \vdash M : \alpha}{\Gamma ; \Psi \vdash \lfloor [\hat{\Phi} \triangleright M] \rfloor_{\sigma} \equiv [\sigma]\, M : [\sigma]\, \alpha}\ \texttt{es-unbox}$$

Figure 5.11: Equality Rules for Specification Terms

computational/reasoning language and one for each syntactic category of the specification language.

Again equality is defined in Figure 5.10 for computations and for specifications in Figures 5.11, 5.12, and 5.13.

$\boxed{\Gamma;\Psi \vdash K \equiv K'}$: K is equal to K' in contexts Γ and Ψ.

$$\frac{}{\Gamma;\Psi \vdash \star \equiv \star}\ \texttt{ek-base} \qquad \frac{\Gamma;\Psi \vdash \alpha \equiv \beta : \star \quad \Gamma;\Psi, \widehat{x}:\alpha \vdash K \equiv K'}{\Gamma;\Psi \vdash (\widehat{x}:\alpha) \twoheadrightarrow K \equiv (\widehat{x}:\beta) \twoheadrightarrow K'}$$

$\boxed{\Gamma;\Psi \vdash \alpha \equiv \beta : K}$: α is equal to β at kind K in contexts Γ and Ψ.

$$\frac{}{\Gamma;\Psi \vdash \mathbf{a} \equiv \mathbf{a} : K}\ \texttt{ef-base}$$

$$\frac{\Gamma;\Psi \vdash \alpha \equiv \alpha' : \star \quad \Gamma;\Psi, \widehat{x}:\alpha \vdash \beta \equiv \beta' : \star}{\Gamma;\Psi \vdash (\widehat{x}:\alpha) \twoheadrightarrow \beta \equiv (\widehat{x}:\alpha') \twoheadrightarrow \beta' : \star}\ \texttt{ef-pi}$$

$$\frac{\Gamma;\Psi \vdash \alpha \equiv \beta : (\widehat{x}:\delta) \twoheadrightarrow K \quad \Gamma;\Psi \vdash M \equiv N : \delta}{\Gamma;\Psi \vdash \alpha\, M \equiv \beta\, N : [x := M]K}\ \texttt{ef-ins}$$

Figure 5.12: Equality Rules for Specification Types and Kinds

$\boxed{\Gamma \vdash \Psi \equiv \Phi : \texttt{ctx}}$: Ψ is equal to Φ in context Γ.

$$\frac{}{\Gamma \vdash \emptyset \equiv \emptyset : \texttt{ctx}}\ \texttt{ec-empty} \qquad \frac{}{\Gamma \vdash \lfloor x \rfloor \equiv \lfloor x \rfloor : \texttt{ctx}}\ \texttt{ec-ctx}$$

$$\frac{\Gamma \vdash \Psi \equiv \Phi : \texttt{ctx} \quad \Gamma;\Psi \vdash \alpha \equiv \beta : \star}{\Gamma \vdash \Psi, \widehat{x}:\alpha \equiv \Phi, \widehat{x}:\beta : \texttt{ctx}}\ \texttt{ec-cons}$$

$\boxed{\Gamma \vdash \hat{\Psi} \equiv \hat{\Phi} : \texttt{ctx}}$: Erased context $\hat{\Psi}$ is equal to $\hat{\Phi}$ in context Γ.

$$\frac{}{\Gamma \vdash \emptyset \equiv \emptyset : \texttt{ctx}}\ \texttt{ee-empty} \qquad \frac{}{\Gamma \vdash \lfloor x \rfloor \equiv \lfloor x \rfloor : \texttt{ctx}}\ \texttt{ee-ctx}$$

$$\frac{\Gamma \vdash \Psi \equiv \Phi : \texttt{ctx}}{\Gamma \vdash \Psi, \widehat{x} \equiv \Phi, \widehat{x} : \texttt{ctx}}\ \texttt{ee-cons}$$

Figure 5.13: Equality Rules for Contexts

5.4 Definitions and Pattern Matching

In Section 5.3, we extended a type theory with contextual types and their introduction forms (i.e.: contextual terms). However, to make use of these definitions it is necessary to have an elimination form. To that effect in these section we extend the language with top-level definitions by pattern matching. These definitions take apart values by using simultaneous pattern matching in the style of Agda [Norell, 2007].

Patterns for Definitions

Pattern	P	$::=$	$\mathbf{c}\,\vec{P}$	A fully applied constant
		$\mid$	x	A variable
		$\mid$	$.P$	An inaccessible pattern
		$\mid$	$[\hat{\Psi} \triangleright \mathcal{P}]$	A contextual pattern
Spec. Pattern	$\mathcal{P}$	$::=$	$\widehat{\mathbf{c}}\,\vec{\mathcal{P}}$	A fully applied constant
		$\mid$	$\widehat{\lambda} x.\mathcal{P}$	An abstraction
		$\mid$	$\widehat{x}$	A bound variable
		$\mid$	$.\mathcal{P}$	An inaccessible pattern
		$\mid$	$\lfloor x \rfloor_{\bar{\sigma}}$	An unboxed meta-variable
Pattern Subst.	$\bar{\sigma}$	$::=$	$\wedge$	An empty substitution
		$\mid$	$\mathtt{id}_n$	An identity substitution, with weakening

The syntax for patterns is straightforward, we highlight in red where computations embed specifications and vice-versa. Notice, how in patterns only variables can be unboxed as it is not clear what it would mean to pattern match against a computation and not a value. Furthermore, the substitution is weaker as only empty and identity with weakening substitutions are allowed. Because we use fully applied constructors for patterns we define the patterns $\mathbf{c}\,\vec{P}$ and $\widehat{\mathbf{c}}\,\vec{\mathcal{P}}$ that are applied to a spine

of parameters. The type-checking of pattern spines uses its own typing judgment where we check a spine against a type that produces as a result the final type of the constant applied to the pattern. Figure 5.14 shows the typing of patterns.

Having defined patterns we move on to define top-level pattern matching definitions:

Type annotation	$\mathbf{f} : T$
Equations	$\Gamma_1 \,.\, \mathbf{f}\, \vec{P}_1 = E_1$
	$\vdots$
	$\Gamma_n \,.\, \mathbf{f}\, \vec{P}_n = E_n$

A definition by pattern matching introduces a new constant and its type followed by its pattern matching equations. Each equation contains a context that binds all the variables used in the pattern, a set of pattern and the right hand side of the equation that contains the term that performs the computation when this equation is matched. For each equation, patterns use every variable from the context exactly once. Because non-linear patterns are unavoidable in dependent pattern matching, all non-linear occurrences of a variable must be in inside inaccessible patterns (where the value of the pattern is forced by the type and thus is not really a discrimination pattern). This is not enforced in the typing rules and has to be checked by the implementation. Inaccessible patterns provide a nice way of implementing the computation of pattern matching by using discrimination trees, this is mostly a practical concern at this point. The constant defined in a function by pattern matching may be used inside the right hand side of any equations. As functions may appear in types and be executed during type-checking, they should be total. In this chapter we do not discuss how to check termination and coverage to establish totality, but rather leave that as future work. However the user is required to provide functions where pattern matching is covering and that recursive calls are terminating.

$\boxed{\Gamma \vdash P : T}$: P is of type T in context Γ.

$$\frac{\mathbf{c} : S \in \Sigma \quad \Gamma \vdash \vec{P} : S \rangle T}{\Gamma \vdash \mathbf{c}\,\vec{P} : T}\ \texttt{tp-con} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \texttt{tp-var}$$

$$\frac{\Gamma \vdash P : T}{\Gamma \vdash .P : T}\ \texttt{tp-inac} \qquad \frac{\Gamma; \Psi \vdash \mathcal{P} : \alpha}{\Gamma \vdash [\hat{\Psi} \triangleright \mathcal{P}] : [\Psi \vdash \alpha]}\ \texttt{tp-box}$$

$\boxed{\Gamma \vdash \overrightarrow{P} : S \rangle T}$: Spine $\overrightarrow{P}$ is of type S and produces type T in context Γ.

$$\frac{\Gamma \vdash P : S \quad \Gamma \vdash \vec{P} : \{x := P\}\,S' \rangle T}{\Gamma \vdash P\,\vec{P} : (x : S) \to S' \rangle T}\ \texttt{tp-spine} \qquad \frac{}{\Gamma \vdash \cdot : T \rangle T}\ \texttt{tp-nil}$$

$\boxed{\Gamma; \Psi \vdash \mathcal{P} : \alpha}$: $\mathcal{P}$ is of type α in contexts Γ and Ψ.

$$\frac{\widehat{\mathbf{c}} : \overrightarrow{(\widehat{x} : \alpha)} \twoheadrightarrow \beta \in \Sigma \quad \Gamma; \Psi \vdash \vec{\mathcal{P}} : \vec{\alpha} \quad \sigma = \overrightarrow{x := \mathcal{P}_i}}{\Gamma; \Psi \vdash \widehat{\mathbf{c}}\,\vec{\mathcal{P}} : [\sigma]\,\beta}\ \texttt{sp-const}$$

$$\frac{\Gamma; \Psi, \widehat{x} : \alpha \vdash \mathcal{P} : \beta}{\Gamma; \Psi \vdash \widehat{\lambda} x.\mathcal{P} : (\widehat{x} : \alpha) \twoheadrightarrow \beta}\ \texttt{sp-lam} \qquad \frac{\widehat{x} : \alpha \in \Psi}{\Gamma; \Psi \vdash \widehat{x} : \alpha}\ \texttt{sp-var}$$

$$\frac{\Gamma; \Psi \vdash \mathcal{P} : \alpha}{\Gamma; \Psi \vdash .\mathcal{P} : \alpha}\ \texttt{sp-inac} \qquad \frac{\Gamma; \Phi \vdash x : \alpha \quad \Gamma; \Psi \vdash \sigma : \Phi}{\Gamma; \Psi \vdash \lfloor x \rfloor_\sigma : [\sigma]\,\alpha}\ \texttt{sp-unbox}$$

$\boxed{\Gamma; \Psi \vdash \overrightarrow{\mathcal{P}} : \alpha \rangle \beta}$: Spine $\overrightarrow{\mathcal{P}}$ is of type α and produces type β in Γ and Ψ.

$$\frac{\Gamma; \Psi \vdash \mathcal{P} : \alpha \quad \Gamma \vdash \vec{\mathcal{P}} : [x := \mathcal{P}]\alpha' \rangle \beta}{\Gamma; \Psi \vdash \mathcal{P}\,\vec{\mathcal{P}} : (\widehat{x} : \alpha) \twoheadrightarrow \alpha' \rangle \beta}\ \texttt{sp-spine} \qquad \frac{}{\Gamma; \Psi \vdash \cdot : \beta \rangle \beta}\ \texttt{sp-nil}$$

Figure 5.14: Typing Rules for Patterns

$\boxed{\vdash \Gamma_i \,.\, \mathbf{f}\,\vec{P_i} = E_i\ \texttt{valid}}$: The equation is valid

$$\frac{\Gamma_i \vdash \mathbf{f}\,\vec{P_1} : T \quad \Gamma_i \vdash E_i : T' \quad \Gamma_i \vdash T \equiv T'}{\vdash \Gamma_i \,.\, \mathbf{f}\,\vec{P} = E_1\ \texttt{valid}}\ \texttt{t-pat}$$

Figure 5.15: Typing Equations

A valid pattern matching definition extends the signature and the reduction behaviour of the system where each of the clauses becomes a new reduction rule.

The typing of equations is done using the typing judgment in Figure 5.15 that depends on the typing of patterns presented in Figure 5.14.

Definitions by pattern matching extend the signature Σ with a new constant (i.e.: the name of the function) and possibly multiple pattern matching equations that become new computation rules. We extend the syntax of the system as follows to accommodate for equations:

Extended Signature and Simultaneous Substitutions

Signature	Σ	::=	$\ldots$	
		$\mid$	$\Sigma, \Gamma_i \,.\, \mathbf{f}\,\vec{P_i} = E_i$	Pattern matching equation
Substitution	θ	::=	$\cdot$	An empty substitution
		$\mid$	$\theta; x := E$	A substitution extension

Definitions by pattern matching compute by extending computation using the equations in the definition in the following form:

$$\frac{\mathbf{f} : \overrightarrow{(x : S)} \to T \in \Sigma \quad \Gamma \vdash \vec{E} : \vec{S} \quad \Gamma_1 \,.\, \mathbf{f}\,\vec{P_i} = E_i \in \Sigma \quad \Gamma \vdash \vec{P_i} \doteq \vec{E}/\theta}{\Gamma \vdash \mathbf{f}\,\vec{E} \equiv \{\theta\}\,E_i : \{\theta\}\,T}\ \texttt{e-equ}$$

This requires the computation of higher-order matching (i.e.: operation $\Gamma \vdash P \doteq E/\theta$) between patterns and the terms passed as parameter to

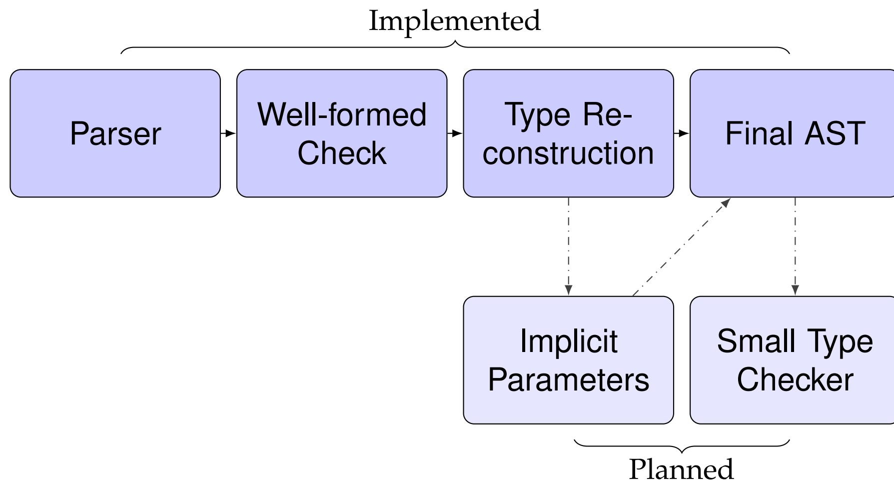

Figure 5.16: The Orca Pipeline

the function. A sensible implementation, (and it is indeed the Orca implementation does), tries the equations in the order they where defined. This is done so as to respect the user's specified order in the case of overlapping patterns.

5.5 The Prototype Implementation

As seen in the example from Section 5.2, the ideas from Section 5.3 have been implemented in a prototype language called Orca (following the marine mammal tradition of languages like Beluga and Delphin [Poswolsky and Schürmann, 2009]). The OCaml implementation is available on Github[3].

Figure 5.16 shows the pipeline for the prototype. The first line shows the existing pipeline, and the phases in the second line show planned features that have not been implemented yet. The first two phases form the front end of the language. First, the parser produces the first version

[3] `https://github.com/orca-lang/orca`

of the AST (Abstract Syntax Tree). Subsequently, the AST generated by the parser is checked for some well formed properties:

- It disambiguates between constructors, variables, and bound variables.
- It checks the scope of variables and definitions.
- It transforms the bound variables of the specification framework to de Bruijn indices.

The idea behind this design is to have a very permissive *parser*, and a second pass that removes ambiguities and can report accurate error messages[4]. The language does not enforce restrictions on naming conventions for user defined identifiers. Constructors, variable names and definitions can be composed of capitals, symbols and characters from any writing system supported by the Unicode standard (with some restriction for keywords and the symbols used in the grammar itself). The *well-formed check* phase checks that variables and definitions respect their scopes and prepares an intermediate AST that is susceptible to type reconstruction and checking. As a side note, this phase is the perfect place to add mix-fix parsing [Danielsson and Norell, 2011] in the style of the Agda language. As a remark, the lack of enforced conventions for names is done for flexibility, but for large developments it would be important to develop such a convention. We expect an ad-hoc convention to emerge once larger Orca examples are implemented.

The third pass, type reconstruction, generates the final version of the AST with all the type checking information included. This pass performs two main tasks:

- It disambiguates the computation and specification terms by inserting boxes and un-boxes appropriately.

[4]Even if the current implementation does not exploit this to have great error messages

```
def tran : (g h : ctx) → rel [g] [h] →
           (st : ⊢ s-tp) → (g ⊢ s-exp ' st) →
           (h ⊢ t-exp ' [(tran-tp st)][^])
where
| g h r t [g :> (app s[^] .t[^] m n)] ⇒
    [h :> tapp ' [(tran-tp s)][^] ' [(tran-tp t)][^]
      ' [(tran g h r [g :> (arr ' [s][^] ' [t][^])] m)]
      ' [(tran g h r s n)]]

| g h r [:> (arr ' s[^] ' t[^])]
      [g :>(lam ' .s[^] ' .t[^] ' m)] ⇒
    [h :> tlam ' [(tran-tp s)][^] ' [(tran-tp t)][^] '
          (λx. [tran (g, x:s-exp s[^])
                      (h, x:t-exp [tran-tp s][^])
                      (cons g h s r)
                      t
                      ([g, x:s-exp s[^] :> m[^1] '  x])
                ][^1;x])]

| g h r [:>  bool] [g :> tt] ⇒
  [h :> tinl ' tunit ' tunit ' tone ]

| g h r [:>  bool] [g :> ff] ⇒
  [h :> tinr ' tunit ' tunit ' tone ]

| g h r t [g :> (if ' .t[^] ' b ' e1 ' e2)] ⇒
    [h :> tcase ' tunit ' tunit ' [(tran-tp t)][^] '
          [(tran g h r bool b)] '
          (λx. [tran g h r t e1][^1]) '
          (λx. [tran g h r t e2][^1])]

| .[g, x:s-exp ' t[^]] .[h, x:t-exp ' [tran-tp t][^]]
  (cons g h .t r) t (g, x: s-exp t[^] :> x) ⇒
      (h, x: t-exp [(tran-tp t)][^] :> x)

| .[g, x:s-exp ' t[^]] .[h, x:t-exp ' [tran-tp t][^]]
  (cons g h t r) s (v[^1]) ⇒
    [tran g h r s v][^1]
```

Figure 5.17: Boolean Translation After Box Inference

- It simultaneously type checks user input.

Following the spirit from Chapter 3, the boxes are reconstructed by a type directed algorithm that by construction produces well typed expressions as a result. The AST produced by this phase, is complete, and can be type-checked without having to infer or complete any type infor-

mation. A small type-checker would make the trusted computing base of the system small. Similarly, having implicit parameter reconstruction would make writing programs easier to write. Both aspects are planned additions to the Orca prototype.

5.6 Related Work

The idea of embedding a specification framework in a computational λ-calculus using a modality was first presented in [Despeyroux et al., 1997] where they embed a simply-typed version of LF in a simply typed lambda calculus extended with a box modality based on that from S4. As mentioned, in Orca the reasoning framework is the logical framework LF [Harper et al., 1993] and the computational framework is a dependently typed system [Martin-Löf, 1982]. This presents a very first step towards answering the long standing question of combining HOAS and dependent types.

Moreover, Orca can be seen as an extension of the Beluga language [Pientka and Cave, 2015]. As such it uses the logical framework LF to represent syntax and judgments using higher-order abstract syntax (HOAS). However, it extends Beluga's first order reasoning language to a dependently typed system where computations can be embedded in specifications. The Twelf system [Pfenning and Schürmann, 1999] also implements LF where they implement computation using logic programming instead of a functional framework. The Delphin system [Poswolsky and Schürmann, 2008] manipulates LF objects using functional programs in a similar way to Beluga but without support for inductive types or contextual types.

Other approaches that use a two-level system with specifications and reasoning are the Abella prover [Baelde et al., 2014] and Hybrid framework [Felty and Momigliano, 2012], these two systems provide a spec-

ification logic on top of a proof-theoretic reasoning logic. Abella is a standalone proof assistant, while Hybrid can be implemented on existing proof assistants, implementations for Coq and Isabelle exist.

Traditionally, implementing HOAS in systems without a specification framework is troublesome because of the positivity restriction for inductive types. To overcome this problem, weaker forms of HOAS use an abstract type to represent variables and obtain a strictly positive type. Examples of this are the work of Chlipala [2008] and Despeyroux and Hirschowitz [1994]. Weaker forms of HOAS are convenient but they do not get substitution from β-reduction in the host language and make dealing with open terms less convenient.

Another option is to implement binders using de Bruijn indices. A convenient way is using well-scoped de Bruijn indices [Altenkirch, 1993] (and later extended to intrinsically typed terms in [Benton et al., 2012a]), where substitution is implemented as an inductive data type and its structural properties need to be proved manually. There are implementations of de Bruijn indices that provide and facilitate proving the required lemmas. One example is Autosubst [Schäfer et al., 2015] that implements a decision procedure to compare terms with explicit substitutions. The Nameless Painless [Pouillard, 2011] approach is based on de Bruijn indices, but it simplifies the arithmetic reasoning by hiding the numbers in an abstract "world" representation. Finally, GMeta [Lee et al., 2012] simplifies implementing first-order binding representations (e.g.: de Bruijn indices) using generic programming to provide proofs for the lemmas about binders.

More recently, Allais et al. [2017] propose a way of encoding syntax (as opposed to syntax and judgments) in Agda using a generic type and scope preserving semantics. This allows them to prove lemmas about the semantics and then reuse them to implement renaming, substitution, normalization by evaluation and CPS transformations.

Finally, a possible approach is to use nominal logic [Pitts, 2003] that uses an infinite set of atoms (abstract names) to replace the position based approach. These ideas have been implemented in Nominal [Urban, 2008] package of Isabelle [Nipkow et al., 2002].

Finally, thanks to its specification framework, Orca is well-positioned to implement meta-programming in dependently typed systems where syntax can be reflected into LF objects where computational functions can manipulate them. There is interest in doing this in existing systems as a way of implementing tactics and elaboration, for example the MTAC [Ziliani et al., 2015] system for Coq, or elaboration reflection [Christiansen and Brady, 2016] for Idris implement meta-programming (The Agda proof assistant also implements Idris-style reflection). What Orca offers to meta-programming is the ability of processing structures with binders in a type safe and convenient way, instead, for example, of the de Bruijn representation that the Idris reflection offers.

5.7 Conclusion

In this chapter we describe Orca, as an idea and a prototype that implements a Martin-Löf style type theory with a specification language based on the logical framework LF. This language can be thought of as a dependently typed Beluga, and thus continues the theme of cetacean inspired names (after Beluga and Delphin). The design of Orca can also be seen as a next step extending the design of the Babybel system from Chapter 4 to total languages and full LF and thus continues that line of work. However, this chapter does not address the full spectrum of possibilities and questions about Orca. This should not be seen as a limitation but as an aspiration. Exploring the meta-theory of the system, its expressivity and the fields where it will be applicable is all exciting future work. Some of this work is already in progress, particularly the meta-theory and improving

and extending the prototype with new ideas. However, the current state of affairs is promising, we show how Orca can interleave computation and specifications, and how this saves some lemmas that would be necessary in systems that need to represent specification computation as relations. Moreover, the Orca distribution contains several examples that hint to the power of reasoning not only about specifications and inductive types like in Beluga, but also about functions. Some of these examples are: a small type preservation proof, the translation we used as an example here, a proof that a particular computation preservers types (illustrated with the simplest computation function, the copy function), a conversion between two styles of operational semantics and several small programs that we use as the beginnings of a test harness. In conclusion, Orca represents a new bold first step in the reasoning about open objects and offers a path to explore extensions to the ideas presented in this chapter.

6 Conclusion

In this thesis, after introducing the problem of reasoning about open terms using a HOAS representation, we show how to simplify writing these programs using inference of omitted arguments, how to write programs by integrating contextual types and HOAS with existing programming languages, and finally we present the Orca system that does the first step towards integrating a Martin-Löf's style type theory with contextual types and the logical framework LF, this allows for the interleaving of specifications and proofs/computations. Therefore, Orca, once the meta-theory is proven and the totality check implemented, allows for proofs about meta-theoretic properties of specifications together with proofs about computations which allows the user to prove properties of their computations over specifications. This provides for the seamless combination of programs and proofs.

6.1 Future Work

6.1.1 Implicit Parameter Reconstruction

Implicit parameter reconstruction makes our lives easier, by allowing us to concentrate in what is important. Reconstruction only fills in parts that are forced by the surrounding program. As future work, one would like to extend the kinds of information that can be inferred. Furthermore, as the current algorithm works for indexed type systems, it would need to be extended for full dependent types to support reconstruction of Orca programs. Having a formally specified type reconstruction for Orca will dramatically improve its ergonomics. Another interesting approach would be to specify reconstruction as form elaboration reflection [Christiansen and Brady, 2016] that would allow for reconstruction to be done in

Orca code itself. Orca's specification framework should make for a great target for this reflection mechanism because the LF framework could be used to represent the program being elaborated.

6.1.2 Contextual Types and Programming Languages

In the future, we plan to implement our approach also in other languages. In particular, it would be natural to implement our approach in Haskell. GHC Haskell offers the required syntax extension mechanism and its more powerful type system offers interesting possibilities. With regards to the implementation, the PPX syntax extension mechanism is a bit limited (e.g.: the syntax of type annotations cannot be extended), it would be interesting to either extend PPX or to replace PPX with CamlP4 that would allow for a more seamless syntax extension. And finally, while the current features of Babybel allow for many interesting use cases, it would be interesting to develop more substantial use cases to both validate the convenience our approach and to possibly justify new features that a larger program might require. For example, implementing the compiler for a small language would neatly showcase the use of the syntactic framework.

6.1.3 Contextual Types and Type Theory

This chapter represents an open path to future research, which is another way to say that there are lots of open questions regarding this subjects. So far, there is no meta-theory for the core calculus we presented, as future work it would be important to show that normalization is preserved when extending a dependently typed calculus with the logical framework LF using contextual types in the way described in this chapter[1]. Also on the side of the theory, certain needed features are missing to make it more

[1]There is ongoing work on draft of meta-theory for this

expressive, like substitution variables [Cave and Pientka, 2013]. Finally, inspired by the Babybel theory where contexts are erased at run-time, in this calculus contexts are run-time irrelevant. This means that properties about contexts need to be established using inductive predicates (e.g.: the `rel` predicate in the example from Section 5.2). Adding Beluga style context schemas would allow for more powerful use of contexts.

On the side of the Orca prototype, the immediate future work is mechanically making sure that the functions are total, that is implementing coverage and termination checking. Implementing coverage would extend ideas from the Beluga system [Pientka and Dunfield, 2010a] and for termination many choices are possible. The author would like to explore the idea of sized types [Hughes et al., 1996] and [Abel, 2006]. And from here the possibilities expand, as Orca can be used as a vehicle for research in this area. Examples of this are: adding generic judgments as in the work by Miller and Tiu [2005] to have Abella [Gacek, 2008] and Delphin style introduction of fresh names, and exploring co-inductive types with co-patterns.

A Proof of Soundness of Reconstruction

We begin with some lemmas that we will use to establish the main result:

Lemma 1 (Implicit parameter instantiation)**.** Let's consider the judgment: $\Theta; \Delta; \Gamma \vdash E : T \rightsquigarrow E_1 : T_1 / \Theta_1$, where Θ_1 is a weakening of Θ.

We want to prove that, if ρ_g is a grounding instantiation such as $\cdot \vdash \rho_g : \Theta_1$ where we split $\rho_g = \rho'_g, \rho''_g$ and $\cdot \vdash \rho'_g : \Theta$ and $\cdot; [\![\rho'_g]\!]\Delta; [\![\rho'_g]\!]\Gamma \vdash [\![\rho'_g]\!]E : [\![\rho'_g]\!]T$ then $\cdot; [\![\rho_g]\!]\Delta; [\![\rho_g]\!]\Gamma \vdash [\![\rho_g]\!]E_1 : [\![\rho_g]\!]T_1$.

Proof. The proof follows by induction on the rules of the judgment where the base case for `el-impl-done` is trivial and the inductive step for `el-impl` has also a very direct proof. □

Lemma 2 (Pattern elaboration)**.**

1. If $\Theta; \Delta \vdash pat \rightsquigarrow \Pi\Delta_1; \Gamma_1.Pat : T / \Theta_1; \rho_1$ and ρ_r is a further refinement substitution, such as $\Theta_2 \vdash \rho_r : \Theta_1$ and ϵ is a ground lifting substitution, such as $\Delta_i \vdash \epsilon : \Theta_1$ then
   $\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash [\![\epsilon]\!][\![\rho_r]\!]Pat \Leftarrow [\![\epsilon]\!][\![\rho_r]\!]T$.

2. If $\Theta; \Delta \vdash pat : T \rightsquigarrow \Pi\Delta_1; \Gamma_1.Pat / \Theta_1; \rho_1$ and ρ_r is a further refinement substitution, such as $\Theta_2 \vdash \rho_r : \Theta_1$ and ϵ is a ground lifting substitution, such as $\Delta_i \vdash \epsilon : \Theta_1$ then
   $\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash [\![\epsilon]\!][\![\rho_r]\!]Pat \Leftarrow [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]T$.

3. If $\Theta; \Delta \vdash \overrightarrow{pat} : T \rightsquigarrow \Pi\Delta_1; \Gamma_1.\overrightarrow{Pat} \rangle S / \Theta_1; \rho_1$ and ρ_r is a further refinement substitution, such as $\Theta_2 \vdash \rho_r : \Theta_1$ and ϵ is a ground lifting substitution, such as $\Delta_i \vdash \epsilon : \Theta_1$ then
   $\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash [\![\epsilon]\!][\![\rho_r]\!]\overrightarrow{Pat} \Leftarrow [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]T \rangle [\![\epsilon]\!][\![\rho_r]\!]S$.

Proof. By simultaneous induction on the first derivation.

For (1):

Case $\mathcal{D} : \Theta; \Delta \vdash \mathbf{c}\,\overrightarrow{pat} \rightsquigarrow \Pi\Delta_1; \Gamma_1.\mathbf{c}\,\overrightarrow{Pat} : S/\Theta_1; \rho_1$

$\Sigma(\mathbf{c}) = T$

$\Theta; \Delta \vdash \overrightarrow{pat} : T \rightsquigarrow \Pi\Delta_1; \Gamma_1.\overrightarrow{Pat} \rangle S/\Theta_1; \rho_1$ by assumption

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash [\![\epsilon]\!][\![\rho_r]\!]\overrightarrow{Pat} \Leftarrow [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]T \rangle [\![\epsilon]\!][\![\rho_r]\!]S$ by i.h. (3)

Note that types in the signature (i.e. Σ) are ground so $[\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]T = T$

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash \mathbf{c}\,([\![\epsilon]\!][\![\rho_r]\!]\overrightarrow{Pat}) \Leftarrow [\![\epsilon]\!][\![\rho_r]\!]S$ by `t-pcon`.

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash [\![\epsilon]\!][\![\rho_r]\!](\mathbf{c}\,\overrightarrow{Pat}) \Leftarrow [\![\epsilon]\!][\![\rho_r]\!]S$

by properties of substitution

which is what we wanted to show.

For (2):

Case $\mathcal{E} : \Theta; \Delta \vdash x : T \rightsquigarrow \Pi\Delta_1 ; \underbrace{x:T}_{\Gamma_1}.x \,/\, \Theta; \mathtt{id}(\Theta)$

$\Gamma_1(x) = T$ by x being the only variable in Γ_1

$[\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 = [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 T$ by applying ϵ and ρ_r to Δ_1, Γ_1 and T

$[\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash x \Leftarrow [\![\epsilon]\!][\![\rho_r]\!]T$ by rule `t-pvar`

which is what we wanted to prove

For (3):

Case

$\mathcal{F} : \Theta; \Delta \vdash pat\ \overrightarrow{pat} : T_1 \to T_2 \rightsquigarrow \Pi\Delta_2; \Gamma_1, \Gamma_2.([\![\rho']\!]Pat)\ \overrightarrow{Pat} \rangle S/\Theta_2; \rho_2 \circ \rho_1$

$\Theta; \Delta \vdash pat : T_1 \rightsquigarrow \Pi\Delta_1; \Gamma_1.Pat/\Theta_1; \rho_1$

$\Theta_1; \Delta_1 \vdash \overrightarrow{pat} : [\![\rho]\!]T_2 \rightsquigarrow \Pi\Delta_2; \Gamma_2.\overrightarrow{Pat} \rangle S/\Theta_2; \rho_2$ by assumption

$\Theta_2 \vdash \rho_2 : \Theta_1$ by invariant of rule

$\Theta_3 \vdash \rho_3 \circ \rho_2 : \Theta_1$ *(further refinement substitution)* by composition
$\Delta_i \vdash \epsilon : \Theta_3$ lifting substitution
$\Delta_i, [\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]\Delta_1; [\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]\Gamma_1 \vdash [\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]Pat \Leftarrow [\![\epsilon]\!][\![\rho_3 \circ \rho_2 \circ \rho_1]\!]T_1$
by i.h. on (1). (†)
$\Delta_i, [\![\epsilon]\!][\![\rho_3]\!]\Delta_2; [\![\epsilon]\!][\![\rho_3]\!]\Gamma_2 \vdash [\![\epsilon]\!][\![\rho_3]\!]\overrightarrow{Pat} \Leftarrow [\![\epsilon]\!][\![\rho_3 \circ \rho_2 \circ \rho_1]\!]T_2 \rangle [\![\epsilon]\!][\![\rho_3]\!]S$
by i.h. on (2)
we note that in pattern elaboration we have:
$\Delta_2 = [\![\rho_2]\!]\Delta_1, \Delta_2'$

Δ_2 is the context Δ_1 with the hole instantiation applied and some extra assumptions (i.e. Δ_2'))).
and $\Gamma_2 = [\![\rho_2]\!]\Gamma_1, \Gamma_2'$

Γ_2 is the context Γ_1 with the hole instantiation applied and some extra assumptions (i.e. Γ_2').
and we can weaken (†) to:
$\Delta_i, [\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]\Delta_1, [\![\epsilon]\!][\![\rho_3]\!]\Delta_2'; [\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]\Gamma_1, [\![\epsilon]\!][\![\rho_3]\!]\Gamma_2' \vdash$
$[\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]Pat \Leftarrow [\![\rho]\!][\![\rho_3 \circ \rho_2 \circ \rho_1]\!]T_1$
$\Delta_i, [\![\epsilon]\!][\![\rho_3]\!]\Delta_2; [\![\epsilon]\!][\![\rho_3]\!]\Gamma_2 \vdash ([\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]Pat)([\![\epsilon]\!][\![\rho_3]\!]\overrightarrow{Pat}) \Leftarrow$
$[\![\epsilon]\!][\![\rho_3 \circ \rho_2 \circ \rho_1]\!]T_1 \to [\![\epsilon]\!][\![\rho_3 \circ \rho_2 \circ \rho_1]\!]T_2 \rangle [\![\epsilon]\!][\![\rho_3]\!]S$
by `t-sarr`.
$\Delta_i, [\![\epsilon]\!][\![\rho_3]\!]\Delta_2; [\![\epsilon]\!][\![\rho_3]\!]\Gamma_2 \vdash [\![\epsilon]\!][\![\rho_3]\!]([\![\rho_2]\!]Pat\,\overrightarrow{Pat}) \Leftarrow [\![\epsilon]\!][\![\rho_3 \circ \rho_2 \circ \rho_1]\!](T_1 \to T_2) \rangle [\![\epsilon]\!][\![\rho_3]\!]S$ by properties of substitution
which is what we wanted to show.

Case
$\mathcal{F} : \Theta; \Delta \vdash [c]\,\overrightarrow{pat} : \Pi^e X : U.\, T \rightsquigarrow \Pi\Delta_2; \Gamma_2.([\![\rho_1]\!][C])\,\overrightarrow{Pat} \rangle S/\Theta_2; \rho_2 \circ \rho_1$

$\Theta; \Delta \vdash c : U \rightsquigarrow C/\Theta_1; \Delta_1; \rho_1$
$\Theta_1; \Delta_1 \vdash \overrightarrow{pat} : [C/X][\![\rho_1]\!]T \rightsquigarrow \Pi\Delta_2; \Gamma_2.\overrightarrow{Pat} \rangle S/\Theta_2; \rho_2$ by assumption
$\Theta_2 \vdash \rho_2 : \Theta_1$ by invariant of rule
$\Theta_3 \vdash \rho_3 \circ \rho_2 : \Theta_1$ *(further refinement substitution)* by composition
$\Delta_i \vdash \epsilon : \Theta_3$ lifting substitution

$\Delta_i, [\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]\Delta_1 \vdash [\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]C \Leftarrow [\![\epsilon]\!][\![\rho_3 \circ \rho_2 \circ \rho_1]\!]U$

by property of the index language(†)

$\Delta_i, [\![\epsilon]\!][\![\rho_3]\!]\Delta_2; [\![\epsilon]\!][\![\rho_3]\!]\Gamma_2 \vdash [\![\epsilon]\!][\![\rho_3]\!]\overrightarrow{Pat} \Leftarrow [\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]([C/X][\![\rho_1]\!]T) \rangle$
$[\![\epsilon]\!][\![\rho_3]\!]S$

by i.h. (3)

as before, we note that:

$\Delta_2 = [\![\rho_2]\!]\Delta_1, \Delta'_2$ Δ_2 is the context Δ_1 with the hole instantiation applied and some extra assumptions (i.e. Δ'_2).

and we can weaken (†)to:

$\Delta_i, [\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]\Delta_1, [\![\epsilon]\!][\![\rho_3]\!]\Delta'_2 \vdash [\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]C \Leftarrow [\![\epsilon]\!][\![\rho_3 \circ \rho_2 \circ \rho_1]\!]U$

Note that

$[\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]([C/X][\![\rho_1]\!]T) = [([\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]C)/X]([\![\epsilon]\!][\![\rho_3 \circ \rho_2 \circ \rho_1]\!]T)$

by properties of substitution

$\Delta_i, [\![\epsilon]\!][\![\rho_3]\!]\Delta_2; [\![\epsilon]\!][\![\rho_3]\!]\Gamma_2 \vdash [[\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]C]\ ([\![\epsilon]\!][\![\rho_3]\!]\overrightarrow{Pat}) \Leftarrow$
$\Pi^e X : ([\![\epsilon]\!][\![\rho_3 \circ \rho_2 \circ \rho_1]\!]U). ([\![\epsilon]\!][\![\rho_3 \circ \rho_2 \circ \rho_1]\!]T) \rangle [\![\epsilon]\!][\![\rho_3]\!]S$ by `t-spi`

$\Delta_i, [\![\epsilon]\!][\![\rho_3]\!]\Delta_2; [\![\epsilon]\!][\![\rho_3]\!]\Gamma_2 \vdash [\![\epsilon]\!][\![\rho_3]\!]([[\![\rho_2]\!]C]\ \overrightarrow{Pat})) \Leftarrow$
$[\![\epsilon]\!][\![\rho_3 \circ \rho_2 \circ \rho_1]\!](\Pi^e X : U. T) \rangle [\![\epsilon]\!][\![\rho_3]\!]S$

by properties of substitution

which is what we wanted to show.

Case $\mathcal{F} : \Theta; \Delta \vdash \overrightarrow{pat} : \Pi^i X : U. T \rightsquigarrow \Pi\Delta_1; \Gamma_1.([\![\rho_1]\!]C)\ \overrightarrow{Pat} \rangle S/\Theta_1; \rho_1$

genHole $(?Y : \Delta.U) = C$

$\Theta, ?Y : \Delta.U; \Delta \vdash \overrightarrow{pat} : [C/X]T \rightsquigarrow \Pi\Delta'; \Gamma'.\overrightarrow{Pat}/\Theta'; \rho \rangle S$ by assumption

$\Theta, ?Y : \Delta.U; \Delta \vdash C \Leftarrow U$ by genhole invariant

$\Delta_i, [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]\Delta \vdash [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]C \Leftarrow [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]U$

applying substitutions $\epsilon, \rho_r and \rho_1$

noting that $\Delta_1 = [\![\rho_1]\!]\Delta, \Delta'_1$

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]([\![\rho_1]\!]\Delta, \Delta'_1) \vdash [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]C \Leftarrow [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]U$ by weakening

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma' \vdash [\![\epsilon]\!][\![\rho_r]\!]\overrightarrow{Pat} \Leftarrow [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!][C/X]T \rangle [\![\epsilon]\!][\![\rho_r]\!]S$

by i.h. (3)

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma' \vdash [\![\epsilon]\!][\![\rho_r]\!]\overrightarrow{Pat} \Leftarrow$
$[[\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]C/X]([\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]T) \rangle [\![\epsilon]\!][\![\rho_r]\!]S$

by properties of substitution

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma' \vdash [[\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]C]\, [\![\epsilon]\!][\![\rho_r]\!]\overrightarrow{Pat} \Leftarrow$
$\Pi^i X : [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]U. ([\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]T) \rangle [\![\epsilon]\!][\![\rho_r]\!]S$

by `t-spi`

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma' \vdash [\![\epsilon]\!][\![\rho_r]\!][[\![\rho_1]\!]C]\, \overrightarrow{Pat} \Leftarrow [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!](\Pi^i X : U. T) \rangle$
$[\![\epsilon]\!][\![\rho_r]\!]S$

by properties of substitution

which is what we wanted to show

□

We can now prove the theorem:

Theorem A.0.1 (Soundness)**.**

1. If $\Theta; \Delta; \Gamma \vdash \wr e\ ;\ \theta \int\ :\ T \rightsquigarrow E/\Theta_1; \rho_1$ then for any grounding hole instantiation ρ_g s.t. $\cdot \vdash \rho_g : \Theta_1$ and $\rho_0 = \rho_g \circ \rho_1$, we have $[\![\rho_0]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]E \Leftarrow [\![\rho_0]\!]T$.
2. If $\Theta; \Delta; \Gamma \vdash \wr e\ ;\ \theta \int \rightsquigarrow E : T/\Theta_1; \rho_1$ then for any grounding hole instantiation ρ_g s.t. $\cdot \vdash \rho_g : \Theta_1$ and $\rho_0 = \rho_g \circ \rho_1$, we have $[\![\rho_0]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]E \Rightarrow [\![\rho_g]\!]T$.
3. If $\Delta; \Gamma \vdash \wr pat \mapsto e\ ;\ \theta \int\ :\ S \to T \rightsquigarrow \Pi\Delta'; \Gamma'.Pat\ :\ \theta' \mapsto E$ then $\Delta; \Gamma \vdash \Pi\Delta'; \Gamma'.Pat : \theta' \mapsto E \Leftarrow S \to T$.

Proof. By simultaneous induction on the first derivation.
For (1):

Case $\mathcal{D} : \Theta; \Delta; \Gamma \vdash \wr \mathtt{case}\, e\, \mathtt{of}\, \overrightarrow{b}\ ; \theta \int : T \rightsquigarrow \mathtt{case}\, E\, \mathtt{of}\, \overrightarrow{B}/\Theta'; \rho$

$\Theta; \Delta; \Gamma \vdash \wr e\ ; \theta \int \rightsquigarrow E : S/\cdot; \rho$ by inversion on `el-case`

$[\![\rho]\!]\Delta; [\![\rho]\!]\Gamma \vdash \wr \overrightarrow{b} \,;\, [\![\rho]\!]\theta \wr : S \to [\![\rho]\!]T \rightsquigarrow \overrightarrow{B}$ by inversion on `el-case`
for any grounding hole inst. ρ' we have $[\![\rho]\!]\Delta; [\![\rho]\!]\Gamma \vdash E \Rightarrow S$ by I.H.
noting $\rho' = \cdot$ and $\rho' \circ \rho = \rho$
$[\![\rho]\!]\Delta; [\![\rho]\!]\Gamma \vdash B : S \to [\![\rho]\!]T$ for every branch by (3)
$[\![\rho]\!]\Delta; [\![\rho]\!]\Gamma \vdash \texttt{case}\, E\, \texttt{of}\, \overrightarrow{B} \Leftarrow [\![\rho]\!]T$ by `t-case`

Note that because E is ground then the only grounding hole to instantiate is the empty substitution.

Case $\mathcal{D} : \Theta; \Delta; \Gamma \vdash \wr \texttt{fn}\, x \Rightarrow e \,;\, \theta \wr : T_1 \to T_2 \rightsquigarrow \texttt{fn}\, x \Rightarrow E/\Theta_1; \rho_1$

$\Theta; \Delta; \Gamma, x : T_1 \vdash \wr e \,;\, \theta \wr : T_2 \rightsquigarrow E/\Theta_1; \rho_1$ by assumption
for any grounding hole inst. ρ_g we have:
$[\![\rho_0]\!]\Delta; [\![\rho_o]\!](\Gamma, x : T_1) \vdash [\![\rho_g]\!]E \Leftarrow [\![\rho_0]\!]T_2$ by i.h. (1) with $\rho_0 = \rho_g \circ \rho_1$
$[\![\rho_0]\!]\Delta; ([\![\rho_o]\!]\Gamma), x : ([\![\rho_0]\!]T_1) \vdash [\![\rho_g]\!]E \Leftarrow [\![\rho_0]\!]T_2$
by properties of substitution
$[\![\rho_0]\!]\Delta; [\![\rho_o]\!]\Gamma \vdash \texttt{fn}\, x \Rightarrow ([\![\rho_g]\!]E) \Leftarrow ([\![\rho_0]\!]T_1) \to ([\![\rho_0]\!]T_2)$ by `t-fn`
$[\![\rho_0]\!]\Delta; [\![\rho_o]\!]\Gamma \vdash [\![\rho_g]\!](\texttt{fn}\, x \Rightarrow E) \Leftarrow [\![\rho_0]\!](T_1) \to T_2)$
by properties of substitution
which is what we wanted to show

Case $\mathcal{D} : \Theta; \Delta; \Gamma \vdash \wr \texttt{mlam}\, X \Rightarrow e \,;\, \theta \wr : \Pi^e X : U.\, T \rightsquigarrow \texttt{mlam}\, X \Rightarrow E/\Theta_1; \rho_1$

$\Theta; \Delta, X : U; \Gamma \vdash \wr e \,;\, \theta, X/X \wr : T \rightsquigarrow E/\Theta_1; \rho_1$ by assumption
for any grounding hole inst. ρ_g we have:
$[\![\rho_0]\!](\Delta, X : U); [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]E \Leftarrow [\![\rho_o]\!]T$ by i.h.(1) with $\rho_0 = \rho_g \circ \rho_1$
$[\![\rho_o]\!]\Delta, X : ([\![\rho_0]\!]U); [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]E \Leftarrow [\![\rho_o]\!]T$ by properties of subst
$[\![\rho_o]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash \texttt{mlam}\, X \Rightarrow [\![\rho_g]\!]E \Leftarrow \Pi^e X : [\![\rho_0]\!]U.\, ([\![\rho_o]\!]T)$ by `t-mlam`
$[\![\rho_o]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]\texttt{mlam}\, X \Rightarrow E \Leftarrow [\![\rho_0]\!]\Pi^e X : U.\, T$
by properties of substitution
which is what we wanted to show

Case $\mathcal{D} : \Theta; \Delta; \Gamma \vdash \wr e\ ;\ \theta \int : \Pi^i X : U.\, T \rightsquigarrow \mathtt{mlam}\, X \Rightarrow E/\Theta_1; \rho_1$

this case follows the same structure as the previous

Case $\mathcal{D} : \Theta; \Delta; \Gamma \vdash \wr [c]\ ;\ \theta \int : [U] \rightsquigarrow [C]/\Theta_1; \rho_1$

$\Theta; \Delta \vdash \wr c\ ;\ \theta \int : U \rightsquigarrow C/\Theta_1; \rho_1$ by assumption
for any grounding inst. ρ_g we have $[\![\rho_0]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]C \Leftarrow [\![\rho_0]\!]U$
by properties of the index language and $\rho_0 = \rho_g \circ \rho_1$
$[\![\rho_0]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!][C] \Leftarrow [\![\rho_0]\!][U]$ by `t-box` and properties of subst.
which is what we wanted to show

Case $\mathcal{D} : \Theta; \Delta; \Gamma \vdash \wr e\ ;\ \theta \int : T \rightsquigarrow [\![\rho_2]\!]E/\Theta_2; \rho_2 \circ \rho_1$

$\Theta; \Delta; \Gamma \vdash \wr e\ ;\ \theta \int \rightsquigarrow E : T_1/\Theta_1; \rho_1$
$\Theta_1; [\![\rho_1]\!]\Delta \vdash T_1 \doteq [\![\rho_1]\!]T/\Theta_2; \rho_2$ by assumption
for any grounding inst. ρ_g we have $[\![\rho_o]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]E \Rightarrow [\![\rho_g]\!]T_1$ by i.h. (2) where $\rho_o = \rho_g \circ \rho_1$ (†)
for any grounding inst. ρ'_g we have $[\![\rho'_g \circ \rho_2]\!]T_1 = [\![\rho'_g \circ \rho_2 \circ \rho_1]\!]T$ by prop of unification and applying a grounding subst (‡)
$[\![\rho'_g \circ \rho_2 \circ \rho_1]\!]\Delta; [\![\rho'_g \circ \rho_2 \circ \rho_1]\!]\Gamma \vdash [\![\rho'_g \circ \rho_2]\!]E \Rightarrow [\![\rho'_g \circ \rho_2]\!]T_1$ from (†)using $\rho_g = \rho'_g \circ \rho_2$
$[\![\rho'_g \circ \rho_2 \circ \rho_1]\!]\Delta; [\![\rho'_g \circ \rho_2 \circ \rho_1]\!]\Gamma \vdash [\![\rho'_g \circ \rho_2]\!]E \Rightarrow [\![\rho'_g \circ \rho_2 \circ \rho_1]\!]T$ by (‡)
$[\![\rho'_g \circ \rho_2 \circ \rho_1]\!]\Delta; [\![\rho'_g \circ \rho_2 \circ \rho_1]\!]\Gamma \vdash [\![\rho'_g \circ \rho_2]\!]E \Leftarrow [\![\rho'_g \circ \rho_2 \circ \rho_1]\!]T$ by `t-syn`
which is what we wanted to show

For(2):

Case $\mathcal{E} : \Theta; \Delta; \Gamma \vdash \wr e\,[c]\ ;\ \theta \int \rightsquigarrow E_1\,[C] : [C/X]([\![\rho_2]\!]T)/\Theta_2; \rho_2 \circ \rho_1$

$\Theta; \Delta; \Gamma \vdash \wr e\ ;\ \theta \int \rightsquigarrow E_1 : \Pi^e X : U.\, T/\Theta_1; \rho_1$
$\Theta_1; [\![\rho_1]\!]\Delta \vdash \wr c\ ;\ [\![\rho_1]\!]\theta \int : U \rightsquigarrow C/\Theta_2; \rho_2$ by assumption

for any grounding instantiation ρ_g s.t. $\cdot \vdash \rho_g : \Theta_1$ we have
$[\![\rho_g \circ \rho_1]\!]\Delta; [\![\rho_g \circ \rho_1]\!]\Gamma \vdash [\![\rho_g]\!]E_1 \Rightarrow [\![\rho_g]\!]\Pi^e X : U.\, T$ by i.h. (2)(†)
for any grounding instantiation ρ'_g s.t. $\cdot \vdash \rho'_g : \Theta_2$ we have
$[\![\rho'_g \circ \rho_2 \circ \rho_1]\!]\Delta \vdash [\![\rho'_g]\!]C \Leftarrow [\![\rho'_g \circ \rho_2]\!]U$ by soundness of index reconstruction
$[\![\rho'_g \circ \rho_2 \circ \rho_1]\!]\Delta; [\![\rho'_g \circ \rho_2 \circ \rho_1]\!]\Gamma \vdash [\![\rho'_g \circ \rho_2]\!]E_1 \Rightarrow [\![\rho'_g \circ \rho_2]\!]\Pi^e X : U.\, T$ Note that in (†)$\cdot \vdash \rho_g : \Theta_1$ so we can instantiate $\rho_g = \rho'_g \circ \rho_2$
$[\![\rho'_g \circ \rho_2 \circ \rho_1]\!]\Delta; [\![\rho'_g \circ \rho_2 \circ \rho_1]\!]\Gamma \vdash [\![\rho'_g \circ \rho_2]\!]E_1 \Rightarrow \Pi^e X : ([\![\rho'_g \circ \rho_2]\!]U).\, ([\![\rho'_g \circ \rho_2]\!]T)$
by properties of substitutions
$[\![\rho'_g \circ \rho_2 \circ \rho_1]\!]\Delta; [\![\rho'_g \circ \rho_2 \circ \rho_1]\!]\Gamma \vdash ([\![\rho'_g \circ \rho_2]\!]E_1)\, [\![\rho'_g]\!]C \Rightarrow [[\![\rho'_g]\!]C\}/X]([\![\rho'_g \circ \rho_2]\!]T)$
by `t-app-index`
$[\![\rho'_g \circ \rho_2 \circ \rho_1]\!]\Delta; [\![\rho'_g \circ \rho_2 \circ \rho_1]\!]\Gamma \vdash [\![\rho'_g]\!](([\![\rho_2]\!]E_1)\, C) \Rightarrow [\![\rho'_g]\!]([C/X]([\![\rho_2]\!]T)$
by properties of substitutions
which is what we wanted to show.

Case $\mathcal{E} : \Theta; \Delta; \Gamma \vdash \wr x\ ;\ \theta \int \rightsquigarrow E_1 : T_1\ /\ \Theta_1; \mathtt{id}(\Theta_1)$

$\Gamma(x) = T$
$\Theta; \Delta; \Gamma \vdash x : T \rightsquigarrow E_1 : T_1 / \Theta_1$ by assumption
$\Delta; \Gamma \vdash x \Rightarrow T$ by rule `t-var`(†)
for any grounding inst. ρ_g s.t. $\cdot \vdash \Theta_1$ we have:
$[\![\rho_g \circ \rho_1]\!]\Delta; [\![\rho_g \circ \rho_1]\!]\Gamma \vdash [\![\rho_g]\!]E_1 : [\![\rho_g]\!]T_1$ by (†), weakening and lemma 1 with $\rho_1 = \mathtt{id}(\Theta_1)$
which is what we wanted to show

For (3):

Case $\mathcal{F} : \Delta; \Gamma \vdash \wr pat \mapsto e\ ;\ \theta \int : S \to T \rightsquigarrow \Pi\Delta_r; \Gamma_r.Pat' : \theta \mapsto E$

$\Delta \vdash pat : S \rightsquigarrow \Pi\Delta_r; \Gamma_r.Pat : \theta_r \mid \theta_e$ by assumption
$\cdot; \cdot \vdash pat \rightsquigarrow Pat : S' / \Theta_p; \Delta_p; \Gamma_p \mid \cdot$

$\Delta'_p \vdash \rho : \Theta_p$ and $\Gamma_r = [\theta_p][\![\rho]\!]\Gamma_p$, $Pat' = [\theta_p][\![\rho]\!]Pat$ by inversion on `el-subst`

$\Delta'_p, [\![\rho]\!]\Delta_p; [\![\rho]\!]\Gamma_p \vdash [\![\rho]\!]Pat \Leftarrow [\![\rho]\!]S'$ by pattern elaboration lemma

$\Delta, \Delta'_p, [\![\rho]\!]\Delta_p \vdash [\![\rho]\!]S' \doteq S/\Delta_r, \theta$ by inversion on `el-subst`

where we can split θ as $\theta = \theta_r, \theta_i, \theta_e$ so that: $\begin{cases} \Delta_r \vdash \theta_r : \Delta \\ \Delta_r \vdash \theta_i : \Delta'_p \\ \Delta_r \vdash \theta_i, \theta_e : \Delta'_p, [\![\rho]\!]\Delta_p \end{cases}$

let $\theta_p = \theta_i, \theta_e$

$[\underbrace{\theta_i, \theta_e}_{\theta_p}][\![\rho]\!]S' = [\theta_r]S$ by soundness of unification and the fact that Δ and $\Delta'_p, [\![\rho]\!]\Delta_p$ are distinct

$\Delta_r; [\theta_p][\![\rho]\!]\Gamma_p \vdash [\theta_p][\![\rho]\!]Pat \Leftarrow [\theta_p][\![\rho]\!]S'$ by substitution lemma

$\Delta_r; \underbrace{[\theta_p][\![\rho]\!]\Gamma_p}_{\Gamma_r} \vdash \underbrace{[\theta_p][\![\rho]\!]Pat}_{Pat'} \Leftarrow [\theta_r]S$ by $[\theta][\![\rho]\!]S' = [\theta_r]S$

$\cdot; \Delta_r; [\theta_r]\Gamma, \Gamma_r \vdash \wr e \,;\, \theta_r \circ \theta,\ \theta_e \int : [\theta_r]T \rightsquigarrow E/\cdot;\cdot$ by assumption

$\Delta_r; [\theta_r]\Gamma, \Gamma_r \vdash E \Leftarrow [\theta_r]T$ by (1)

$\Delta; \Gamma \vdash \Pi\Delta_r; \Gamma_r.Pat' : \theta_r \mapsto E \Leftarrow S \to T$ by `t-branch`

which is what we wanted to show.

□

B Babybel's Translation Meta-theory

Lemma 3 (Ambient Context)**.** If $\Gamma(\mathsf{u}) = [\Psi \vdash \mathbf{a}]$ then
$\ulcorner\Gamma\urcorner(u) = \mathtt{sftm}[\ulcorner\Psi\urcorner, \mathbf{a}]$.

Proof. Induction on the structure of Γ. □

Lemma 4 (Terms)**.**

1. If $\Gamma; \Psi \vdash M : A$ then $\cdot; \ulcorner\Gamma\urcorner \vdash \ulcorner M \urcorner_{\Psi \vdash A} : \ulcorner \Psi \vdash A \urcorner$.

2. If $\Gamma; \Psi \vdash \sigma : \Phi$ then $\cdot; \ulcorner\Gamma\urcorner \vdash \ulcorner \sigma \urcorner_{\Psi \vdash \Phi} : \ulcorner \Psi \vdash \Phi \urcorner$

Proof. Induction on the typing derivation.

Case $\mathcal{D} = \dfrac{\Psi(x) = \mathbf{a}}{\Gamma; \Psi \vdash x : \mathbf{a}}\ \mathtt{t\text{-}var}$

$\cdot; \ulcorner\Gamma\urcorner \vdash \mathtt{Var}[\ulcorner\Psi\urcorner, \mathbf{a}]\ k : \mathtt{sftm}[\ulcorner\Psi\urcorner, \mathbf{a}]$ by the correctness of our translation function that computes the position k of x in Ψ.

$\cdot; \ulcorner\Gamma\urcorner \vdash \ulcorner x \urcorner_{\Psi \vdash \mathbf{a}} : \ulcorner \Psi \vdash \mathbf{a} \urcorner$ by definition

Case $\mathcal{D} = \dfrac{\Gamma; \Psi \vdash M : A \quad \Gamma; \Phi \vdash \sigma : \Psi}{\Gamma; \Phi \vdash M[\sigma]^{\Phi}_{\Psi} : A}\ \mathtt{t\text{-}sub}$

$\cdot; \ulcorner\Gamma\urcorner \vdash \ulcorner M \urcorner_{\Psi \vdash A} : \ulcorner \Psi \vdash A \urcorner$ by i.h.

$\cdot; \ulcorner\Gamma\urcorner \vdash \ulcorner \sigma \urcorner_{\Phi \vdash \Psi} : \ulcorner \Phi \vdash \Psi \urcorner$ by i.h.

$e = \mathtt{apply_sub}\ \ulcorner M \urcorner_{\Psi \vdash A}\ \ulcorner \sigma \urcorner_{\Phi \vdash \Psi}$ and
$\cdot; \ulcorner\Gamma\urcorner \vdash e : \ulcorner \Phi \vdash A \urcorner$ by property of `apply_sub`

Case $\mathcal{D} = \dfrac{\Gamma(\mathsf{u}) = [\Psi \vdash \mathbf{a}]}{\Gamma; \Psi \vdash \mathtt{'u} : \mathbf{a}}\ \mathtt{t\text{-}qvar}$

$\ulcorner\Gamma\urcorner(\mathsf{u}) = \mathtt{sftm}[\ulcorner\Psi, \mathbf{a}\urcorner]$ by Lemma 3

$\cdot; \ulcorner\Gamma\urcorner \vdash \ulcorner\texttt{'u}\urcorner : \ulcorner\Psi \vdash \mathbf{a}\urcorner$ by rule `g-var` and definition

Case $\mathcal{D} = \dfrac{\Gamma; \Psi, x : \mathbf{a} \vdash M : A}{\Gamma; \Psi \vdash \lambda x.M : \mathbf{a} \to A}$ `t-lam`

$\cdot; \ulcorner\Gamma\urcorner \vdash \ulcorner M\urcorner_{\Psi,\mathbf{a}\vdash A} : \ulcorner\Psi, \mathbf{a} \vdash A\urcorner$ by i.h.

$\ulcorner\Psi \vdash \mathbf{a} \to A\urcorner = \texttt{sftm}[\ulcorner\Psi\urcorner, \ulcorner\mathbf{a} \to A\urcorner] = \texttt{sftm}[\ulcorner\Psi\urcorner, \texttt{arr}[\mathbf{a}, \ulcorner A\urcorner]]$ by definition

$\cdot; \ulcorner\Gamma\urcorner \vdash \texttt{Lam [cons [}\ulcorner\Psi\urcorner, \mathbf{a}, \ulcorner A\urcorner\texttt{]]}\ \ulcorner M\urcorner_{\Gamma,\mathbf{a}\vdash A} : \ulcorner\Psi \vdash \mathbf{a} \to A\urcorner$ by using `g-con`

Similar for the other cases.

□

Lemma 5 (Pat.)**.** If $\vdash pat : \tau \downarrow \Gamma$ then $\cdot \vdash \ulcorner pat\urcorner^{\Gamma}_{\Psi\vdash A} : \ulcorner\tau\urcorner \downarrow \Gamma$.

Proof. By induction on the type derivation for patterns.

□

Lemma 6 (Ctx. Pat.)**.** If $\Psi \vdash R : A \downarrow \Gamma$ then $\cdot \vdash \ulcorner R\urcorner^{\Gamma}_{\Psi\vdash A} : \ulcorner\Psi \vdash A\urcorner \downarrow \Gamma$.

Proof. By induction on the typing derivation. The interesting case is the one where R is a pattern variable.

Case: $\mathcal{D} = \dfrac{}{\Psi \vdash \texttt{'u} : \mathbf{a} \downarrow u : [\Psi \vdash \mathbf{a}]}$ `tp-mvar`

$\cdot \vdash u : \texttt{sftm}[\ulcorner\Psi\urcorner, \mathbf{a}] \downarrow \cdot; u : \texttt{sftm}[\ulcorner\Psi\urcorner, \mathbf{a}]$ by `gp-var`

$\cdot \vdash \ulcorner\texttt{'u}\urcorner^{u:[\Psi\vdash\mathbf{a}]}_{\mathbf{a}} : \ulcorner\Psi \vdash \mathbf{a}\urcorner \downarrow \cdot; \ulcorner u : [\Psi \vdash \mathbf{a}]\urcorner$ by definition

The other cases are similar.

□

Theorem B.0.1 (Main)**.**

1. If $\Gamma \vdash e \Leftarrow \tau$ then $\cdot;\ulcorner\Gamma\urcorner \vdash \ulcorner e\urcorner_{\Gamma\vdash\tau} : \ulcorner\tau\urcorner$.

2. If $\Gamma \vdash i \Rightarrow \tau$ then $\cdot;\ulcorner\Gamma\urcorner \vdash \ulcorner i\urcorner_{\Gamma\vdash\tau} : \ulcorner\tau\urcorner$.

Proof. By mutual induction on the type derivations.

Case $\mathcal{D} = \dfrac{\Gamma;\Psi \vdash M : \mathbf{a}}{\Gamma \vdash [\hat{\Psi} \vdash M] \Leftarrow [\Psi \vdash \mathbf{a}]}$ t-ctx-obj

$\cdot;\ulcorner\Gamma\urcorner \vdash \ulcorner M\urcorner_{\Psi\vdash\mathbf{a}} : \ulcorner\Psi \vdash \mathbf{a}\urcorner$ from Lemma 4.

$\ulcorner\Psi \vdash \mathbf{a}\urcorner = \texttt{sftm}[\ulcorner\Psi\urcorner, \ulcorner A\urcorner]$ and $\ulcorner\Psi \vdash \mathbf{M}\urcorner_{\Psi\vdash\mathbf{a}} = \ulcorner M\urcorner_{\Psi\vdash\mathbf{a}}$

$\ulcorner\Gamma\urcorner \vdash \ulcorner\hat{\Psi} \vdash M]\urcorner_{\Psi\vdash\mathbf{a}} : \ulcorner[\Psi \vdash \mathbf{a}]\urcorner$ by definition

Case $\mathcal{D} = \dfrac{\Gamma \vdash i \Rightarrow [\Psi \vdash \mathbf{a}] \quad \forall b \in \overrightarrow{b}\ .\ \Gamma \vdash b \Leftarrow [\Psi \vdash \mathbf{a}] \to \tau}{\Gamma \vdash \texttt{cmatch}\, i\, \texttt{with}\ \overrightarrow{b} \Leftarrow \tau}$ t-cm

We note that each $b_i \in \overrightarrow{b}$ is of the form $[\Psi \vdash R] \mapsto e$.

$\Psi \vdash R : A \downarrow \Gamma$

$\Gamma, \Gamma' \vdash e \Leftarrow \tau$ by typing inversion

$\cdot \vdash \ulcorner R\urcorner^{\Gamma'}_{\Psi\vdash\mathbf{a}} : \mathbf{a} \downarrow \Gamma'$ by Lemma 6

$\cdot;\ulcorner\Gamma, \Gamma'\urcorner \vdash \ulcorner e\urcorner_{\Gamma,\Gamma'\vdash\tau} : \ulcorner\tau\urcorner$ by i.h. (1).

$\cdot;\Gamma, \Gamma' \vdash \ulcorner R\urcorner^{\Gamma'}_{\Psi\vdash\mathbf{a}} \mapsto \ulcorner e\urcorner_{\Gamma,\Gamma'} : \mathbf{a} \to \ulcorner\tau\urcorner$ by g-branch

$\cdot;\ulcorner\Gamma\urcorner \vdash \ulcorner i\urcorner_{\Gamma\vdash\mathbf{a}} : \mathbf{a}$ by I.H (2).

$\cdot;\ulcorner\Gamma\urcorner \vdash \ulcorner\texttt{cmatch}\, i\, \texttt{with}\ \overrightarrow{b}\urcorner_{\Gamma\vdash\tau} : \ulcorner\tau\urcorner$ by g-match

Case $\mathcal{D} = \dfrac{\Gamma \vdash i \Rightarrow \tau' \qquad \tau = \tau'}{\Gamma \vdash i \Leftarrow \tau}$ t-emb

$\cdot;\ulcorner\Gamma\urcorner \vdash \ulcorner i\urcorner_{\Gamma\vdash\tau} : \ulcorner\tau\urcorner$ by i.h.(2)

The other cases for part 1) are similar.

Case $\mathcal{D} = \dfrac{\Gamma(x) = \tau}{\Gamma \vdash x \Rightarrow \tau}$ `t-var`

$\cdot; \ulcorner\Gamma\urcorner \vdash \ulcorner x \urcorner_{\Gamma \vdash \tau} : \ulcorner\tau\urcorner$ trivial using `g-var`.

Case $\mathcal{D} = \dfrac{\Gamma \vdash i \Rightarrow \tau' \to \tau \quad \Gamma \vdash e \Leftarrow \tau'}{\Gamma \vdash i\, e \Rightarrow \tau}$ `t-app`

$\cdot; \ulcorner\Gamma\urcorner \vdash \ulcorner i \urcorner_{\Gamma \vdash \tau' \to \tau} : \ulcorner\tau' \to \tau\urcorner$ by i.h.

$\cdot; \ulcorner\Gamma\urcorner \vdash \ulcorner i \urcorner_{\Gamma \vdash \tau' \to \tau} : \ulcorner\tau'\urcorner \to \ulcorner\tau\urcorner$ by definition

$\cdot; \ulcorner\Gamma\urcorner \vdash \ulcorner e \urcorner_{\Gamma \vdash \tau'} : \ulcorner\tau'\urcorner$ by i.h.

$\cdot; \ulcorner\Gamma\urcorner \vdash \ulcorner i\, e \urcorner_{\Gamma \vdash \tau} : \ulcorner\tau\urcorner$ by `g-app`

The other cases for part 2) are similar.

$\square$

C A Beluga Implementation of Orca's Translation Example

This chapter contains an implementation of the example from Section 5.2 written using the Beluga language. We do not explain this example in detail, but it is here as a reference to show how similar the two languages are.

We start by defining the languages and translation of types using the logical framework LF, this is similar to the Orca implementation, except that where Orca uses a function to implement the translation of types here we have a relation that is inhabited by the pairs of source types and their translations. Notice the schema declaration that allows for storing blocks of assumptions in the context thus avoiding the need to declare a context relation.

```
LF s-tp : type =
| bool : s-tp
| arr : s-tp → s-tp → s-tp
;

LF s-tm : s-tp → type =
| app : s-tm (arr S T) → s-tm S → s-tm T
| lam : (s-tm S → s-tm T) → s-tm (arr S T)
| tt : s-tm bool
| ff : s-tm bool
| ife : s-tm bool → s-tm T → s-tm T → s-tm T
;

LF t-tp : type =
| tunit : t-tp
| tsum : t-tp → t-tp → t-tp
| tarr : t-tp → t-tp → t-tp
;

LF t-tm : t-tp → type =
| tapp : t-tm (tarr S T) → t-tm S → t-tm T
| tlam : (t-tm S → t-tm T) → t-tm (tarr S T)
```

```
| tone : t-tm tunit
| tinl : t-tm s → t-tm (tsum S T)
| tinr : t-tm t → t-tm (tsum S T)
| tcase : t-tm (tsum S T) →
          (t-tm S → t-tm R) →
          (t-tm T → t-tm R) →
          t-tm R
;

LF tran-tp : s-tp → t-tp → type =
| t-bool : tran-tp bool (tsum tunit tunit)
| t-arr : tran-tp S S’ →
          tran-tp T T’ →
          tran-tp (arr S T) (tarr S’ T’)
;

schema ctx = block (s: s-tm S, t: t-tm T, tr: tran-tp S T;
```

The type `tran-tp` replaces the Orca function with the same name. But then, we need to show that the relation is defined for all terms, and that it is deterministic. We prove both facts in the `comp-tran-tp` and `unique-tran-tp` lemmas, respectively.

```
LF ex-tran : s-tp → type =
| ex : {T : t-tp} tran-tp S T → ex-tran S
;

rec comp-tran-tp : {S : [⊢ s-tp]} [⊢ ex-tran S] =
mlam S ⇒ case [⊢ S] of
| [⊢ bool] ⇒ [⊢ ex (tsum tunit tunit) t-bool]
| [⊢ arr S S’] ⇒
  let [⊢ ex T TR] = comp-tran-tp [⊢ S] in
  let [⊢ ex T’ TR’] = comp-tran-tp [⊢ S’] in
  [⊢ ex (tarr T T’) (t-arr TR TR’)]
;

LF eq : t-tp → t-tp → type =
| refl : eq T T
;

rec unique-tran-tp : (g:ctx) [g ⊢ tran-tp S[] T[]] → [⊢ tran-tp S T
    ’] → [⊢ eq T T’] =
fn tr1 ⇒ fn tr2 ⇒ case tr1 of
```

```
| [g ⊢ t-bool] ⇒
  let [⊢ t-bool] = tr2 in
  [⊢ refl]
| [g ⊢ t-arr T1 T2] ⇒
  let [⊢ t-arr T1' T2'] = tr2 in
  let [⊢ refl] = unique-tran-tp [g ⊢ T1] [⊢ T1'] in
  let [⊢ refl] = unique-tran-tp [g ⊢ T2] [⊢ T2'] in
  [⊢ refl]
;
```

With these lemmas in place, the translation can be implemented, notice how the context relation was made superfluous by the information carried in the context schema. Context schemas for Orca remain an unexplored subject.

```
rec tran : (g : ctx) [⊢ tran-tp S T] → [g ⊢ s-tm S[]] → [g ⊢ t-tm
    T[]] =
fn tr ⇒ fn e ⇒ case e of
| {M : [g⊢ s-tm (arr S[] T[])]} [g ⊢ app M N] ⇒
  let [⊢ ex S' TR] = comp-tran-tp [⊢ S] in
  let [⊢ TR'] = tr in
  let [g ⊢ M'] = tran [⊢ t-arr TR TR'] [g ⊢ M] in
  let [g ⊢ N'] = tran [⊢ TR] [g ⊢ N] in
  [g ⊢ tapp M' N']

| [g ⊢ lam λx.M] ⇒
  let [⊢ t-arr TR TR'] :
        [⊢ tran-tp (arr S T) (tarr S' T')] = tr
  in
  let [g, x: block s:s-tm S[]
        , t:t-tm S'[]
        , tr:tran-tp S[] S'[] ⊢ M'[..,x.t]] =
      tran [⊢ TR']
           [g, x: block s:s-tm S[]
             , t:t-tm S'[]
             , tr:tran-tp S[] S'[] ⊢ M[..,x.s]]
  in
  [g ⊢ tlam λx.M']

| [g ⊢ tt] ⇒
  let [⊢ t-bool] = tr in
  [g ⊢ tinr tone]

| [g ⊢ ff] ⇒
  let [⊢ t-bool] = tr in
  [g ⊢ tinl tone]

| [g ⊢ ife C M N] ⇒
  let [g ⊢ C'] = tran [⊢ t-bool] [g ⊢ C] in
  let [g ⊢ M'] = tran tr [g ⊢ M] in
  let [g ⊢ N'] = tran tr [g ⊢ N] in
  [g ⊢ tcase C' (λx. M'[..]) (λx. N'[..])]
```

```
| {#p:[g ⊢ block s:s-tm S[]
               , t:t-tm T[]
               , tr:tran-tp S[] T[]]} [g ⊢ #p.s] ⇒
  let [⊢ refl] = unique-tran-tp [g ⊢ #p.tr] tr in
  [g ⊢ #p.t]
;
```

Bibliography

A. Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians University, 2006.

A. Abel and B. Pientka. Higher-order dynamic pattern unification for dependent types and records. In L. Ong, editor, *10th International Conference on Typed Lambda Calculi and Applications (TLCA'11)*, Lecture Notes in Computer Science (LNCS 6690), pages 10–26. Springer, 2011.

G. Allais, J. Chapman, C. McBride, and J. McKinna. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 195–207. ACM, 2017.

T. Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In M. Bezem and J. F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications (TLCA '93)*, volume 664 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 1993. ISBN 3-540-56517-5.

A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical Methods in Computer Science*, 8:1–49, 2012.

L. Augustsson. Compiling pattern matching. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture (FPCA'85)*, volume 201 of *Lecture Notes in Computer Science (LNCS)*, pages 368–381. Springer, 1985.

D. Baelde, K. Chaudhuri, A. Gacek, D. Miller, G. Nadathur, A. Tiu, and Y. Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014.

H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 2)*, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992. ISBN 0-19-853761-1.

O. S. Belanger, S. Monnier, and B. Pientka. Programming type-safe transformations using higher-order abstract syntax. In G. Gonthier and M. Norrish, editors, *Third International Conference on Certified Programs and Proofs (CPP'13)*, Lecture Notes in Computer Science (LNCS 8307), pages 243–258. Springer, 2013.

N. Benton, C. Hur, A. Kennedy, and C. McBride. Strongly typed term representations in coq. *J. Autom. Reasoning*, 49(2):141–159, 2012a.

N. Benton, C.-K. Hur, A. J. . Kennedy, and C. McBride. Strongly typed term representations in Coq. *Journal of Automated Reasoning*, 49(2):141–159, 2012b.

M. Boespflug and B. Pientka. Multi-level contextual modal type theory. In G. Nadathur and H. Geuvers, editors, *6th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP'11)*, Electronic Proceedings in Theoretical Computer Science (EPTCS), 2011.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

A. Cave and B. Pientka. Programming with binders and indexed datatypes. In *39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 413–424. ACM Press, 2012.

A. Cave and B. Pientka. First-class substitutions in contextual type theory. In *Proceedings of the Eighth ACM SIGPLAN International Workshop on*

Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13), pages 15–24. ACM Press, 2013. ISBN 978-1-4503-2382-6.

A. Cave, F. Ferreira, P. Panangaden, and B. Pientka. Fair reactive programming. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*, pages 361–372. ACM, 2014.

I. Cervesato, F. Pfenning, D. Walker, and K. Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002.

C. Chen and H. Xi. Combining programming with theorem proving. In O. Danvy and B. C. Pierce, editors, *10th International Conference on Functional Programming*, pages 66–77, 2005.

J. Cheney and R. Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003a.

J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003b.

A. J. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In J. Hook and P. Thiemann, editors, *13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 143–156. ACM, 2008.

D. Christiansen and E. Brady. Elaborator reflection: Extending idris in idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 284–297, New York, NY, USA, 2016. ACM.

A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 6 1940.

R. L. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. ISBN 0-134-51832-2.

T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988.

N. A. Danielsson and U. Norell. *Parsing Mixfix Operators*, pages 80–99. Springer Berlin Heidelberg, 2011.

R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001. doi: 10.1145/382780.382785.

N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indag. Math*, 34(5):381–392, 1972.

N. G. de Bruijn. *AUTOMATH, a Language for Mathematics*, pages 159–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

N. G. de Bruijn. Checking Mathematics with Computer Assistance. *Notices of the American Mathematical Society*, 38(1):8–15, 1991.

J. Despeyroux and A. Hirschowitz. Higher-order abstract syntax with induction in coq. In *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, LPAR '94, pages 159–173, London, UK, UK, 1994. Springer-Verlag. ISBN 3-540-58216-9.

J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher-order abstract syntax. In *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163. Springer, 1997. Extended version available as Technical Report CMU-CS-96-172, Carnegie Mellon University.

G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Joint International Conference and Symposium on Logic Programming*, pages 259–273. MIT Press, Sept. 1996.

J. Dunfield and B. Pientka. Case analysis of higher-order data. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08)*, volume 228 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 69–84. Elsevier, June 2009.

P. Dybjer. Inductive Families. *Formal Aspects of Computing*, 6(4):440–465, Jul 1994. ISSN 1433-299X.

A. Felty and A. Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012.

F. Ferreira and B. Pientka. Bidirectional elaboration of dependently typed programs. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*, PPDP '14, pages 161–174. ACM, 2014.

F. Ferreira and B. Pientka. Programs using syntax with first-class binders. In H. Yang, editor, *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Uppsala, Sweden*, pages 504–529. Springer Berlin Heidelberg, 2017.

F. Ferreira, D. Thibodeau, and B. Pientka. Dependent type theory with contextual types. In *23rd International Conference on Types for Proofs and Programs TYPES 2017, Budapest, Hungary*, pages 61–62, 2017.

M. Fiore and C.-K. Hur. Term equational systems and logics: (extended abstract). *Electronic Notes in Theoretical Computer Science*, 218:171 – 192,

2008. ISSN 1571-0661. Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIV).

M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224. IEEE Computer Society Press, 1999.

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

A. Gacek, D. Miller, and G. Nadathur. A two-level logic approach to reasoning about computations. *Journal of Automated Reasoning*, 49(2): 241–273, 2012.

J. Garrigue and D. Rémy. Ambivalent types for principal type inference with gadts. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS'13)*, Lecture Notes in Computer Science (LNCS 8301), pages 257–272. Springer, 2013. ISBN 978-3-319-03541-3.

G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. *A Machine-Checked Proof of the Odd Order Theorem*, pages 163–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

R. Harper and D. R. Licata. Mechanizing Metatheory in a Logical Framework. *Journal of Functional Programming*, 17(4-5):613–673, 2007.

R. Harper and F. Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic*, 6(1):61–101, 2005.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

J. Harrison. Hol light: An overview. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009. ISBN 978-3-642-03358-2.

J. Harrison, J. Urban, and F. Wiedijk. History of interactive theorem proving. In J. H. Siekmann, editor, *Computational Logic*, volume 9 Suplement C of *Handbook of the History of Logic*, pages 135 – 214. North-Holland, 2014.

W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479 – 490. Academic Press, 1980.

J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 410–423, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3.

R. Jacob-Rao. Well-founded recursion in terms and types. Master's thesis, McGill University, 2017.

L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit. In *13th ACM SIGPLAN International Conference on Functional Programming*, pages 27–38. ACM, 2008.

G. Lee, B. C. D. S. Oliveira, S. Cho, and K. Yi. *GMeta: A Generic Formal Metatheory Framework for First-Order Representations*, pages 436–455. Springer Berlin Heidelberg, 2012.

X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml System Release 4.03 – Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, 2016a.

X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system (release 4.04): Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, 2016b.

Z. Luo. *Computation and reasoning: a type theory for computer science.* Oxford University Press, Inc., New York, NY, USA, 1994.

M. Luther. More on implicit syntax. In R. Gore, A. Leitsch, and T. Nipkow, editors, *First International Joint Conference on Automated Reasoning (IJCAR'01)*, Lecture Notes in Artificial Intelligence (LNAI) 2083, pages 386–400. Springer, 2001.

P. Martin-Löf. Constructive mathematics and computer programming. In *6-th International Congress for Logic, Methodology and Philosophy of Science, 1979*, pages 153–175. North-Holland, 1982.

P. Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory Lecture Notes*. Bibliopolis, Napoli, 1984.

D. Miller. Unification of simply typed lambda-terms as logic programming. In *8th International Logic Programming Conference*, pages 255–269. MIT Press, 1991.

D. Miller and C. Palamidessi. Foundational aspects of syntax. *ACM Comput. Surv.*, 31(3es), Sept. 1999.

D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, 6(4):749–783, 2005. ISSN 1529-3785.

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

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löfs Type Theory. An introduction*. Clarendon Press, Oxford, 1990.

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

C. Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In M. Bezem and J. F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications(TLCA '93)*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.

L. C. Paulson. Isabelle: The next seven hundred theorem provers. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction (CADE-9)*, pages 772–773, Argonne, Illinois, 1988. Springer Verlag Lecture Notes in Computer Science (LNCS) 310. System abstract.

F. Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 221–230, Boston, Massachusetts, June 2001. IEEE Computer Society Press.

F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Symposium on Language Design and Implementation (PLDI'88)*, pages 199–208, June 1988.

F. Pfenning and C. Paulin-Mohring. *Inductively defined types in the Calculus of Constructions*, pages 209–228. Springer-Verlag, New York, NY, 1990.

F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th International Conference on Automated Deduction (CADE-16)*, Lecture Notes in Artificial Intelligence (LNAI 1632), pages 202–206. Springer, 1999.

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

B. Pientka. Higher-order term indexing using substitution trees. *ACM Transactions on Computational Logic*, 11(1):1–40, 2009.

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

B. Pientka and A. Abel. Structural recursion over contextual objects. In T. Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*. Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl, 2015.

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

B. Pientka and J. Dunfield. Covering all bases: design and implementation of case analysis for contextual objects. Technical report, McGill University, 2010a.

B. Pientka and J. Dunfield. Beluga: a framework for programming and reasoning with deductive systems (System Description). In J. Giesl and R. Haehnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pages 15–21. Springer-Verlag, 2010b.

B. Pientka and F. Pfenning. Optimizing higher-order pattern unification. In F. Baader, editor, *19th International Conference on Automated Deduction (CADE-19)*, Lecture Notes in Artificial Intelligence (LNAI) 2741, pages 473–487. Springer-Verlag, 2003.

B. C. Pierce and D. N. Turner. Local type inference. *ACM Transaction on Programming Languages and Systems*, 22(1):1–44, jan 2000.

A. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, Nov. 2003. ISSN 0890-5401.

R. Pollack. Implicit syntax. In G. Huet and G. Plotkin, editors, *Proceedings of First Workshop on Logical Frameworks*, pages 421–434, 1990.

A. Poswolsky and C. Schürmann. System description: Delphin—a functional programming language for deductive systems. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08)*, volume 228 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 135–141. Elsevier, 2009.

A. B. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. In *17th European Symposium on Programming (ESOP '08)*, volume 4960, pages 93–107. Springer, 2008.

F. Pottier. An overview of Cαml. *Electronic Notes in Theoretical Computer Science*, 148(2):27 – 52, 2006. Proceedings of the ACM-SIGPLAN Workshop on ML (ML'05).

F. Pottier. Static name control for FreshML. In *22nd IEEE Symposium on Logic in Computer Science (LICS'07)*, pages 356–365. IEEE Computer Society, July 2007.

N. Pouillard. Nameless, painless. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 320–332. ACM, 2011. ISBN 978-1-4503-0865-6.

S. Schäfer, T. Tebbi, and G. Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In C. Urban and X. Zhang, editors, *6th International Conference of Interactive Theorem Proving (ITP)*, Lecture Notes in Computer Science (9236), pages 359–374. Springer, Aug 2015.

T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for gadts. In *14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 341–352. ACM, 2009.

C. Schürmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266(1-2):1–57, 2001.

T. Sheard and E. Pasalic. Meta-programming with built-in type equality. *Electronic Notes in Theoretical Computer Science*, 199:49 – 65, 2008. Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004).

M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: programming with binders made simple. In *8th International Conference on Functional Programming (ICFP'03)*, pages 263–274. ACM Press, 2003.

K. Slind and M. Norrish. *A Brief Overview of HOL4*, pages 28–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

P. Stansifer and M. Wand. Romeo: A system for more flexible binding-safe programming. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 53–65, 2014.

The Coq Developement Team. *The Coq Proof Assistant Reference Manual v. 8.6.1*. Institut National de Recherche en Informatique et en Automatique, 2016.

C. Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Journal of Functional Programming*, 18(01):87–140, 2008.

K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002.

S. Weirich, A. Voizard, P. H. A. de Amorim, and R. A. Eisenberg. A specification for dependent types in Haskell. *Proc. ACM Program. Lang.*, 1(ICFP):31:1–31:29, Aug. 2017. ISSN 2475-1421.

E. Westbrook, N. Frisby, and P. Brauner. Hobbits for Haskell: a library for higher-order encodings in functional programming languages. In *4th ACM Symposium on Haskell (Haskell'11)*, pages 35–46. ACM, 2011.

H. Xi. Applied type system. In *TYPES 2003*, volume 3085 of *Lecture Notes in Computer Science*, pages 394–408. Springer, 2004.

H. Xi. Dependent ml an approach to practical programming with dependent types. *Journal of Functional Programming*, 17:215–286, 3 2007.

H. Xi and F. Pfenning. Dependent types in practical programming. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, 1999.

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*, pages 224–235. ACM Press, 2003.

C. Zenger. Indexed types. *Theoretical Computer Science*, 187(1-2):147–165, 1997.

B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: A monad for typed tactic programming in coq. *Journal of Functional Programming*, 25, 2015.