Université de Montréal

A Compiler for the dependently typed language Beluga

par
Francisco Ferreira

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en computer science

Mai, 2012

© Francisco Ferreira, 2012.

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

A Compiler for the dependently typed language Beluga

présenté par:

Francisco Ferreira

a été évalué par un jury composé des personnes suivantes:

Marc Feeley,	président-rapporteur
Stefan Monnier,	directeur de recherche
Brigitte Pientka,	codirectrice
Philippe Langlais,	membre du jury

Mémoire accepté le: ....

RÉSUMÉ

Les structures avec des lieurs sont très communes en informatique. Les langages de programmation et les systèmes logiques sont des exemples de structures avec des lieurs. La manipulation de lieurs est délicate, de sorte que l'écriture de programmes qui manipulent ces structures tirerait profit d'un soutien spécifique pour les lieurs. L'environnement de programmation Beluga est un exemple d'un tel système. Nous développons et présentons ici un compilateur pour ce système. Parmi les programmes pour lesquels Beluga est spécialement bien adapté, plusieurs peuvent bénéficier d'un compilateur. Par exemple, les programmes pour valider les types (les "type-checkers"), les compilateurs et les interpréteurs tirent profit du soutien spécifique des lieurs et des types dépendants présents dans le langage. Ils nécessitent tous également une exécution efficace, que l'on propose d'obtenir par le biais d'un compilateur. Le but de ce travail est de présenter un nouveau compilateur pour Beluga, qui emploie une représentation interne polyvalente et permet de partager du code entre plusieurs back-ends. Une contribution notable est la compilation du filtrage de Beluga, qui est particulièrement puissante dans ce langage.

Mots clés: lieurs, filtrage, compilation, types dependents

ABSTRACT

In computer science, structures with variable binders are very common. Programming languages and logical frameworks are examples of structures with binders. Thus writing programs that deal with these kinds of data benefits with explicit support for data binding. The Beluga programming environment is an example of such a system.

In this work we develop and present a compiler for the system. Many of the programs that Beluga is specially well suited for writing can benefit from a compiler. For example, some of the kinds programs that would benefit more are type-checkers, compilers and interpreters that take advantage of the binder support and dependent types present in the language, and also require a reasonably fast run-time.

Our goal in this work, is to present a compiler for the Beluga system, that uses a very versatile internal representation that helps with the development of the system, and allows a sharing of code between several back-ends. Furthermore, we present a way of compiling the uniquely powerful pattern language supported by Beluga.

Keywords: binders, pattern matching, compilation, dependent types.

CONTENTS

RÉSUMÉ . . . iii

ABSTRACT . . . iv

CONTENTS . . . v

LIST OF FIGURES . . . viii

LIST OF ABBREVIATIONS . . . xi

DEDICATION . . . xii

ACKNOWLEDGMENTS . . . xiii

CHAPTER 1: INTRODUCTION . . . 1

CHAPTER 2: THE REPRESENTATION OF BINDERS . . . 5
- 2.1 Plain Old Names . . . 6
- 2.2 de Bruijn Indices . . . 8
- 2.3 Higher-Order Abstract Syntax . . . 9
- 2.4 Nominal, Fresh Look and Others . . . 10
  - 2.4.1 Nominal . . . 10
  - 2.4.2 The “Fresh Look” Approach . . . 11

CHAPTER 3: A BELUGA PRIMER . . . 14
- 3.1 An Introduction to Dependent Types . . . 15
- 3.2 Some Programming Examples . . . 16
  - 3.2.1 Dependent Lists . . . 16

3.2.2 HOAS to de Bruijn 20
3.2.3 Normalization 24
3.3 The Beluga Front-end 25
3.4 Contextual LF 27
3.4.1 Summary of the Techniques Used in the Internal Representation 28
3.4.2 Internal Representation of Contextual LF 29
3.5 Computational Language 30

CHAPTER 4: THE COMPILER 32

4.1 The pipeline 32
4.2 Dependency Erasure 33
4.3 The Fresh-Style Representation 35
4.4 Translating to the Fresh-Style Representation 37
4.4.1 A Simple Example 37
4.4.2 The Simply Typed Fresh Style Contextual LF Syntax 40
4.4.3 Translating Beluga Expressions to the Fresh-Style 42
4.4.4 Typing Rules for the Simply-Typed Calculus 44
4.5 Pattern Matching Compilation 48
4.5.1 A Hand-Coded Example 50
4.5.2 An Algorithm for Patterns with Variables and Constructors . . . 51
4.5.3 Compiling Beluga's Patterns 55
4.6 Operational Semantics and Runtime 61
4.6.1 Operational Semantics 61
4.6.2 Names and Indices from Fresh-Style Variables 67
4.6.3 Back-End and Code-Generation 68

CHAPTER 5: CONCLUSION **74**

5.1 Future Work 75

BIBLIOGRAPHY **76**

LIST OF FIGURES

2.1 An example using de Bruijn indices(public domain from [14]). . . 8
2.2 A simple example of HOAS in Haskell 10
2.3 Entities in the Fresh Look approach. 12
3.1 Natural numbers and lists. 16
3.2 `head` and `tail` functions. 17
3.3 The append function. 18
3.4 The append function with `let` expressions. 19
3.5 The append function with non-erasable proofs. 19
3.6 Untyped λ-calculus. 20
3.7 Untyped λ-calculus with de Bruijn indices. 22
3.8 HOAS translation function. 22
3.9 STLC 24
3.10 Normalization function. 25
3.11 The append function after elaboration. 26
3.12 Contextual LF 29
3.13 Computational language. 31
4.1 The compiler pipeline. 32
4.2 Simple types. 34
4.3 Dependency erasure. 34
4.4 The fresh-style module interface. 35
4.5 The fresh-style approach. 36
4.6 Fresh-style $\lambda\sigma$-calculus. 37
4.7 β-reduction with explicit substitutions. 38

4.8 Explicit substitution equations. 38

4.9 Substitution composition for $\lambda\sigma$-calculus. 39

4.10 $\lambda\sigma$ WHNF. 40

4.11 Fresh-style contextual LF. 41

4.12 Translation of contexts. 43

4.13 Translation of terms. 43

4.14 Translation of substitutions. 44

4.15 Computational language internal representation. 45

4.16 Typing rules for simply typed contextual-LF. 46

4.17 Typing rules for the simply typed computational language. 47

4.18 Matrix of clauses. 52

4.19 $S(c, P \to A)$. 53

4.20 $D(P \to A)$. 53

4.21 Kinds of patterns. 56

4.22 Pattern compilation target language. 57

4.23 Typing rules for switch expressions. 58

4.24 Typing rules for simple patterns. 59

4.25 Big-step operational semantics. 62

4.26 Matching operations. 64

4.27 Weak head normal form. 65

4.28 Substitution composition. 67

4.29 Contextual LF with de Bruijn Indices. 69

4.30 Contextual LF with names. 70

4.31 Bound variable substitution with de Bruijn indices. 71

4.32 Matching operations with de Bruijn indices. 72

4.33 Bound variable substitution with names. 72

4.34 Matching operations with names. 73

LIST OF ABBREVIATIONS

AST	Abstract Syntax Tree
CMTT	Contextual Modal Type Theory
HOAS	Higher-Order Abstract Syntax
STLC	Simply Typed Lambda Calculus
WHNF	Weak Head Normal Form

a mi abuela Nanana aunque no esté para verla.

ACKNOWLEDGMENTS

I would like to thank my advisors Stefan Monnier and Brigitte Pientka for all their incredible help and support. And to my friends and chosen family for all their love and care.

CHAPTER 1

INTRODUCTION

Functional programming languages are well suited for working with syntax. They are well adapted and commonly used to write parser generators, compilers, and type-checkers. In this context, syntax is the structure of the phrases of a language and how they are composed together. In this view a phrase in the language we are handling is a tree, usually called an *Abstract Syntax Tree* (AST). Of particular interest in this work is syntax that deals with identifiers. Interesting examples are the syntax of: programming languages, logics with quantifiers, and calculi used in diverse formalizations like λ-calculus and π-calculus. In particular we care about how identifiers are introduced and how they can be used respecting their scopes and substitution rules. When discussing syntax at this level we can talk about an *Abstract Binding Tree* to be clear that the syntax we are working with supports the concepts of binding and scope [27].

However, most functional programming languages lack explicit support for binders. This complicates dealing with abstract binding trees as issues such as ensuring variable freshness, equality up-to renaming variables, problems that arise from implementing capture avoiding substitution, and variables not escaping their scope need to be addressed by the programmer. Currently, there are two main theoretical approaches which aim at making it easier to represent and manipulate data containing binders: supporting nominals, that is names as first-class notions, in the language ([50], [54]) and Higher-Order Abstract Syntax (HOAS) [42], which represents binders in the object level re-using binders of the meta-language (i.e. the language we are using to develop our program), implemented in systems like Twelf [44] and Delphin[52] and Beluga [12]. In general nominal techniques provide at least α-equivalence of terms and fresh names for

variables while HOAS always provides α-equivalence, substitution for free and fresh names for variables. We will elaborate more about the representation of binders in Section 2 on page 5.

Presently, sophisticated binding support is available for proof assistants as it greatly simplifies proofs. However, it has been difficult to make available such sophisticated support to the average programmer. In the case of languages and libraries, techniques such as FreshML [58], folds or catamorphisms [60], and Hobbits [61] all address the problem but they either lack the formal guarantees, are complicated and inconvenient to use or do not scale to richer type-systems, such as dependent types. In contrast, HOAS and in particular its implementation in Beluga [45] addresses these complications in a convenient way. Beluga provides dependent types, and implements HOAS using contextual objects from contextual modal type theory [40]. Contextual objects are terms that may contain binders and that have an associated context where the free variables in the term exist.

When dealing with syntax with binders we have described some of the advantages of having a high-level support for them. In this research we address, for the first time, the main issues of compiling such a language: the representation of bound variables and the compilation of the rich pattern language that Beluga supports.

Beluga is a functional programming language from the ML family [37] with support for pattern matching. It is built on top of a logical framework in the style of LF [29] using contextual objects as in Contextual Modal Type Theory [40] (CMTT). In our compiler, for the internal representation we use a technique derived from [54] that we call the Fresh-Style representation (see Section 4.3 on page 35). We take advantage of the inherent versatility of the method to be able to generate binders with de Bruijn indices and names in the target language for compilation.

Pattern matching compilation (Section 4.5 on page 48) is done using splitting trees([31],[36]),

but the key idea is the separation between how the tree is generated at compile-time and what is left to decide at runtime. At compile-time, as with ML-style languages, the tree is generated by splitting on constructors and matching on variables. However, when dealing with contextual objects the situation is more complex. The generation of the tree starts by splitting on the shapes of the contexts, and then recursively splitting on constructors, meta-variables and bound variables. During run-time matched terms are compared to patterns, while for constructors the comparison is trivial (as it is in the ML family of languages), comparing the shape of the contexts, bound variables and meta-variables is not trivial and depends on the chosen representation of contextual objects where our compilation schema supports using de Bruijn indices and unique names.

A key element of our algorithm is that compilation and splitting-tree computation is performed on contextual values using the Fresh-Style representation (without committing yet to a concrete representation for contextual objects), while the comparison of terms and patterns is done once the internal representation of contextual values has already been made.

Summarizing, the essential contributions of this work are:

- A framework for compiling contextual objects. This is an essential step in adding HOAS and contextual objects to existing programming languages such as Haskell. The framework provides a representation that bridges higher-order to first-order binders. Concretely, HOAS to de Bruijn indices, and HOAS to unique names.
- Pattern matching compilation for contextual objects. In particular, we support deep pattern matching underneath binders.
- This work connects, for the first time, nominal techniques and Contextual Modal Type Theory through the use of an internal representation based on names.
- A compiler that provides a choice whether to generate code where binders use de Bruijn indices or names. This sets up the stage for an evaluation of the tradeoffs

between these representations and the exploration of other options for the representation of binders.

CHAPTER 2

THE REPRESENTATION OF BINDERS

Mathematics, computer science and other disciplines deal with formal languages. In these languages there is commonly the idea of variables or abstract entities that represent a concept that was abstracted over. They get their meaning by substitution. Most formal languages [1] also have a way of introducing new variables, e.g.: the λ-expression in λ-calculus. This work, will concentrate on formal languages from programming languages, and logic.

There are many approaches to representing binders, and multiple solutions have been and are being proposed which achieve different balances between power, simplicity and safety. We can think about this problem along two dimensions. First, systems that use binders (systems like FreshML [58] or Beluga, for instance) or why using a nominal based approach or HOAS is convenient. Finally, a second dimension which concerns the implementation of the language, in this case the compiler and the concrete implementation of binding used to represent variables, in our case Beluga's variables.

We group the existing approaches in one of three categories: *name* based representations, *nameless* based on indices, and *Higher-Order Abstract Syntax*. The following table shows some example systems for each category.

Names	Nameless	HOAS
Plain-old names [18]/Var. Convention [6]	de Bruin indices [20]/Automath [19]	Twelf [44]
Nominal [23]/Isabelle [59]	Nameless Painless [53]	Delphin [51]
FreshML [58]		Beluga [49]
α-Prolog[15]		

To be concrete, we will discuss binders in the sense of variables introduced by ab-

1. at least the ones that we will consider when using Beluga

stractions in λ-calculus, but these ideas apply to many other uses of binders, e.g.: logical formulas, programming languages, formal systems and more.

De Bruijn introduces an interesting way to evaluate each approach in [20], that is:

1. Easy to read and write for the human reader.
2. Easy to handle in the meta-level discussion.
3. Easy for the computer and the programmer.

The emphasis will be in the last item, as it is more pertinent to the implementation of the compiler. This will allow us to understand the amount of work expected to implement such a system, and certain aspects of its resulting performance. This item is the most important for the implementation of the compiler and the choice of its internal implementation of binders.

2.1 Plain Old Names

Names are used in most *paper* presentations of lambda calculus, [7] is one notable example, and it corresponds to the idea of giving variables different names, as we are used to do in Mathematics. So almost all human oriented discussions will use names, as people are familiar with them. In the implementation of systems, when names are used, instead of strings names are represented by an integer for performance and compactness reasons.

The use of names is simple yet powerful. It is used in almost all *blackboard presentations* of structures with binders but also as the internal representation in the implementation of many systems such as programming languages.

The main advantages of this approach are simplicity and familiarity, but this power comes with disadvantages: avoiding name capture/conflicts, comparing terms for equality and performing capture avoiding substitution are the more obvious ones.

Substitution can be considered the most important aspect of bindings because it provides concrete meaning to a previously abstract variable. Whenever we reason with names, on paper or using a proof assistant, we need to satisfy some proof obligations, typically we need to prove some form of the substitution lemma, like Barendregt does in one line in Section 2.1.6 of [7] or Curry in ten pages in Section 2.E.2 of [18]. These two proofs show how powerful it is to use names on paper, because proofs with names are easy to understand, and details are easy to omit. With formal proofs, proof assistants do not afford us that leniency, and all the details need to be satisfied even those missing in the ten page long version of the proof. Of course, proof assistants use automation and other techniques to minimize the effort. But this is simply an example to show that even if names shine on human to human paper proofs, in formal proofs sometimes they are cumbersome due to all the required lemmas. As a matter of fact, this problem is the motivation for many of the other representations.

If we rename bound variables this should not change a term. We have $\lambda x.x \equiv \lambda y.y$ as a consequence when we compare terms we need to consider them equal up-to the renaming of bound variables. Additionally when defining substitution, we need to avoid name capture. On paper presentations we usually waive these requirements adding a comment mentioning that variables are fresh (i.e.:they do not occur in the term) and by mentioning that equality is up-to variable renaming(α-equivalence). This makes this representation practical for human to human communication and is thus used in the vast majority of cases. On the other hand, when implementing systems that use binders details cannot be simply waived away by a comment, and special attention needs to be paid when comparing terms and performing substitution.

2.2 de Bruijn Indices

Because of the already mentioned difficulties in dealing with names, Nicolas Govert de Bruijn introduced in [20] a nameless representation of λ-terms now known as *de Bruijn Indices*. The paper uses as a motivation the complexity and computational cost of performing capture avoiding substitution, and comparing α-equivalent terms.

As presented in Section 2 of [1] in this representation names are replaced by positive integers, where each such number *n* "refers to the variable bound by the *n*-th surrounding binder". We can see in Figure 2.1 an example illustrating this fact.

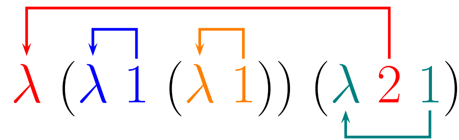

Figure 2.1: An example using de Bruijn indices(public domain from [14]).

In consequence, names disappear because they are replaced by numbers which refer to the distance to their enclosing binder, so no renaming is ever necessary or even possible [2], additionally the comparison of terms is simplified by the fact, that no equivalence up-to names is required, so the two main problems with plain old names are solved.

On the other hand, the representation has its own problems. Readability is compromised, even if the representation is precise. Additionally the manipulation of terms is more complicated; consider for example the term $\lambda.1(\lambda.12)$ where one variable is represented by more than one number, and the same number represents different variables. Performing substitution in this representation entails renumbering of the variable in terms, which is cumbersome and error prone. Further details can be found in [1].

2. for example: if we want to refer to the last variable introduced we use 1 and there is no other option

2.3 Higher-Order Abstract Syntax

So far in Sections 2.1 and 2.2 we have seen some common representations of binders, names and de Bruijn indexes. These are common in real world representations of languages. These are called first order representations because they do not use the binders of the host language. First order representations need to define substitution, and variable comparison which means writing a lot of repetitive code. In the next section we will present a new approach that helps with these deficiencies.

Higher-Order Abstract Syntax introduced in [42] uses the host language's binder support, with the main advantage that this way there is no need to implement substitution (the system reuses the host's language implementation). As an example, in Figure 2.2 we can see the declaration of the syntax of an untyped λ-calculus and its interpreter using a HOAS based representation in Haskell. In this example, λ-terms are represented by a Haskell λ-expression, effectively reusing the binders from Haskell in the representation of λ-calculus. Because we reuse the binders from the host language there is no need to have variables in the type declaration for terms. For this toy example, HOAS works well enough, but an effective use of the technique requires advanced support absent from Haskell and most other languages not designed specifically with HOAS in mind. For example if we want to traverse a term and pattern match on its structure, we cannot do it under `Lam` terms without instantiating binders with dummy-variables or other similar tricks. On the other hand, Beluga was designed to take advantage of the full power of HOAS as we will see in Chapter 3 on page 14.

Higher-order representations provide support for safe handling of binders as they forbid using variables out of their scope, and provide functionality like substitution and equality comparison for free. However, from the point of view of a compiler implementor, it is important to point out that eventually these representations need to be transformed to a first-order representation as there is usually no support for binders in the

```haskell
module HOAS where

-- A declaration of a minimalistic lambda calculus using HOAS, notice how
-- lambda terms are represented as functions in the host language

data Term = App Term Term
          | Lam (Term → Term)

-- The eval function re-uses the function application and substitution
-- infrastructure of Haskell

eval :: Term → Term
eval (Lam f) = Lam f
eval (App m n) =
  case eval m of
    | Lam f → eval (f n) -- application is just Haskell's function application

-- omega is the term λ x. x x applied to itself, twice

omega = eval (App f f)
    where
      f = Lam (λx → (App x x))
```

Figure 2.2: A simple example of HOAS in Haskell

target language. Compilers usually generate low level code, machine code or code in another language, but generally the resulting code does not use the higher level features of the target language. As we will see in Chapter 4, this is how our compiler works. HOAS is not the only higher-order representation and in fact we will discuss some other options in Section 2.4.

2.4 Nominal, Fresh Look and Others

2.4.1 Nominal

In Section 2.1 we discussed how names are intuitive, yet difficult to use in formal proofs. This is only true for what we called "plain-old names", there are other representations that use names that provide powerful features trying to preserve the intuitive aspect of the approach. One such approach is using Nominal Logic ([23] and [50]) which contains "primitives for renaming via name-swapping, for freshness of names, and for name-binding" using permutations from Set theory, in particular the Fraenkel–Mostowski per-

mutation model [24].

One of the most well-known implementations of this idea in a proof assistant is Isabelle/Nominal [59]. In [8] there is a proof of the substitution lemma for λ calculus using the system, this proof showcases neatly how concise and convenient the system is. On the other hand, the implementation of this package, the nominal data-type in Isabelle, requires some classical reasoning which makes it impossible to use Isabelle's program extraction facilities to get an executable program out of the proofs.

If we compare this proof to one using HOAS, we notice that no proof of the substitution lemma is required, as it is provided by the meta language. Additionally Beluga's proofs are programs in contrast to the Isabelle/Nominal package which cannot extract the computational content of proofs (e.g. the normalizer from a normalization proof).

There are other systems that use the nominal ideas, two salient examples are α-prolog [15] and FreshML [58] which are respectively variants of Prolog and ML which support binders in the nominal style.

2.4.2 The "Fresh Look" Approach

Many other approaches to binding exist, some interesting ones can be found in [16], [53] and [22]. However in this section we will discuss one final representation. The representation that we will call *Fresh Look* was introduced in [54] and it is significant for this work because, as we will see, it is the internal binder representation in our compiler (Section 4.3).

The main objective of the *Fresh Look* approach is to be able to represent names in a sound way. As a way of clarification the authors offer the following three *informal slogans* to exemplify what they mean by sound:

– *"name abstractions cannot be violated"*; or: *"the representations of two α-equivalent terms cannot be distinguished"*;

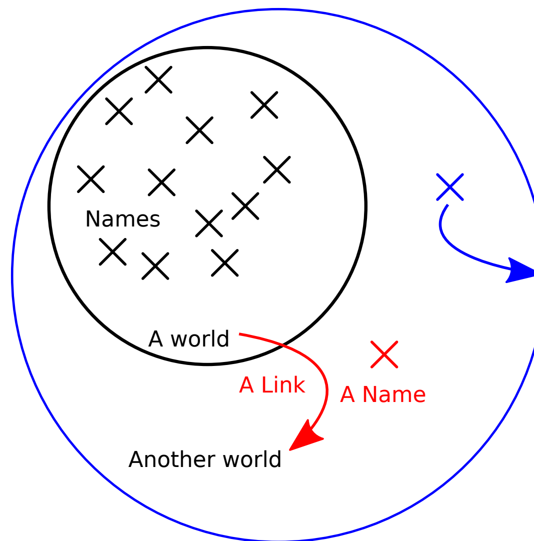

Figure 2.3: Entities in the Fresh Look approach.

– *“names do not escape their scope”*;

– *“names with different scopes cannot be mixed”*

It is important to notice, that low-level representations offer no intrinsic protection for most forms of misuse of binders, such as binders escaping their scopes, or name capture. On the other hand, high level representations like HOAS, the Nominal approach, and Fresh Look satisfy at least some of these requirements. In fact, soundness properties play a central role in being able to have more lightweight proofs (i.e., to avoid to prove certain lemmas, and simplify others).

This representation’s central idea is to have *worlds* which are inhabited by *names*. *Names* are abstract entities that we can compare for equality, and each name inhabits a world. As a consequence, terms and variables using this technique are indexed by a

world. The key concept is that each term cannot mix in the same scope sub-terms or variables from different worlds. The last concept we need is the *link*. *Links* relate one world to another, and each link introduces a name in the destination world. There exists two kinds of links, *hard-links* that introduce a new name in the destination world and *soft-links* that introduce a name that may shadow an existing name. This last distinction is not crucial to our treatment, as our adaptation of the technique only has hard-links, as we will see on Section 4.3 on page 35. Links are used to introduce new binders, by creating a new world which supports all the names from the bigger scope plus the name introduced by the link. Inner worlds are bigger because names can be *imported* from the outer world to the inner world.

Figure 2.3 depicts some of the elements used in this idea:

– names: these are the abstract representation of variables. There is an infinite amount of them, and each name inhabits a world.

– worlds: contain a number of names, and may be related to other worlds by a link.

– links: relate two worlds and introduce a new name in the second world. This is the key concept of the model because a new bigger world contains one extra name that represents a newly bound variables. So in λ calculus, the bigger world represents the sub-term inside a λ abstraction.

In its original presentation, the paper focuses on presenting an implementation done in the dependently-typed language Agda and proving its soundness. In this work, we will focus on using the representation because it is straight-forward to transform binders in this representation to binders in other representations. Of particular interest for our compiler is generating low-level binder representations, namely plain-old names and de Bruijn indices. In a way, this representation is high-level enough that we can delay the choice of concrete implementation of binders, yet sufficiently low-level to be a good target for Beluga's HOAS.

CHAPTER 3

A BELUGA PRIMER

Beluga ([45], [49], [46]) is a dependently-typed programming language and a proof development environment, where proofs are represented as programs. In this discussion we will focus mainly on the system as a programming language.

Beluga provides a two-level language, the data level and the computational level. On the data level objects are represented as Contextual-LF objects [40] and the computational level provides an ML-like functional language with pattern matching facilities to manipulate LF objects.

The data level language is based in the logical framework LF [29] a system able to describe logics and abstract syntax with binders, providing support for HOAS. More specifically Beluga's data language uses Contextual-LF which is an extension of LF based on Contextual Modal Type Theory [40]. In the system, objects, that may contain variables and binders, carry explicit contexts that bind all the free-variables.

A big part of the power of the language is derived from its support of encodings based on higher-order abstract syntax (HOAS), that is values contain binders and contexts where bound variables exist. This allows us to represent structures with binders in a natural way by effectively re-using the binders in the LF-metalanguage. Thus avoiding the burden of dealing with α-renaming and capture avoiding substitutions.

Beluga's type-system includes dependent-types, both at the LF and computational levels. We can capture correctness properties by properly encoding them in the types of our functions; e.g. we may encode the length of an array in its type allowing us to verify the absence of out-of-bounds accesses statically.

3.1 An Introduction to Dependent Types

Beluga derives its strengths from HOAS, described in 2.3 and dependent types that we will introduce and motivate here and in the next section.

In [57] dependent types are discussed in the context of "the semantic gap" [1] which is the gap between what the programmer writes and how to reason about the program. Dependent types narrow this gap by allowing precise specifications of programs.

Types are important because they classify data with regards to different properties, most often how to store them in memory, or how to safely access data. At their core dependent types are types expressed in terms of values, so when before we had a type for lists now we can have a type for lists of a certain length, or have type for terms that depend on a context, etc. As mentioned in [4]and [41] dependent types allow the programmer to choose the degree to which types specify the behaviour of a function. One example could be sorting a list of integers, we could think of four reasonable types (other reasonable types for this function exist):

1. A function from a list of integers to a list of integers.
2. A function from a list of n integers to a list of n integers.
3. A function from a list of n integers to a list of n integers where the result is ordered.
4. A function from a list of n integers to a list of n integers where the result is ordered and it is a permutation of the input list.

The key notion is that we can write the sort function under all of these types, but the specification becomes more and more precise with each of those types, in fact the last type is a complete specification of the behaviour of sort, while for the first two types the identity function would trivially satisfy the type but would not sort anything. For the third type there is also a trivial function that would satisfy the type without sorting (a

1. In the article other aspects of dependent types are discussed, but here we are interested mainly on the one mentioned before.

function that produces a list containing *n* copies of the first element). While the last type, can only really be satisfied by a function that sorts its input. For further details, there is presentation of this example in Section 1.4 of [9]. Additionally, in the next section we discuss some interesting examples using the Beluga language.

3.2 Some Programming Examples

3.2.1 Dependent Lists

In this first example, we discuss how we can encode some non-trivial properties of our data in its type. We present lists of natural numbers that keep track of their lengths in the type. We then can take advantage of these more specific types to write functions that avoid certain runtime checks and exceptions or that encode some correctness properties.

We begin our program by declaring the types we will be using. Types are declared using a similar syntax as in the Twelf implementation of LF, with some additional restrictions because Beluga requires η-long β-normal forms [46].

```
nat  : type.
z : nat.
s : nat → nat.

list : nat → type.
nil : list z.
cons: nat → list X → list (s X).
```

Figure 3.1: Natural numbers and lists.

In Figure 3.1 we declared inductively the type of natural numbers and the `list` family of types. We call it a “family of types” because it takes a parameter of type `nat` to return an instance of the type family. The parameter is used to encode the length of the list. This allows us to define `head` and `tail` functions that work only on non empty lists, avoiding exceptional cases.

Figure 3.2 presents two computational level functions that do exactly what their sim-

```
rec head : (list (s X)) [ ] → nat [ ] =
fn l ⇒ case l of
| [ ] cons H T ⇒ [ ] H
;

rec tail : (list (s X)) [ ] → (list X) [ ] =
fn l ⇒ case l of
| [ ] cons H T ⇒ [ ] T
;
```

Figure 3.2: `head` and `tail` functions.

ply typed counterparts do with the difference that passing an empty list to either of them will be a type error instead of a runtime error. Notice that the `case` expression does not contain a `nil` case because it is not possible for lists of type `list (s X) []`. In Beluga, when we talk about a contextual-value of type `list (s X) []` the square brackets represent the context where this type makes sense. In this particular case, `[]` means the empty context, so it is the type of a list with at least one element and without variables from the context. Additionally, these two functions are total, meaning that they cover all cases in the input, a fact that Beluga is able to check [48]; and they terminate which is obvious in this case but Beluga is not yet able to validate.

This first part shows how to avoid run-time checks and exceptions, as neither function would accept empty lists as a parameter. There is a nuance to the elimination of the check. In order to pass an arbitrary list to these functions, with type `list X`, we need to first refine its type to `list (s X)` through pattern matching before calling the function. The refinement, it may be argued, amounts to doing a run-time check. The power of this approach is that the type checker will fail when such refinements are omitted and will ensure that no runtime exceptions are triggered.

We can also encode correctness properties, for instance we can define an `append` function that returns a list whose length is the sum of the lengths of the two lists passed as parameters and this fact is encoded in the type of the function allowing the type checker to validate it at type-check time.

In order to do this we need to define addition as we defined natural numbers, so we first declare the add type as relation that represent the addition of two natural numbers. Using this type we are ready to encode the type of the append function. The intuition is that we will pass two lists of lengths X and Y and a proof that $X + Y = Z$ and we get a list of length Z in return.

```
add : nat → nat → nat → type.
add_z : add z X X.
add_s : add X Y Z → add (s X) Y (s Z).

rec append : (list X) [ ] → (list Y) [ ] → (add X Y Z) [ ] → (list Z) [ ] =
fn l1 ⇒ fn l2 ⇒ fn a ⇒ case l1 of
| [ ] nil ⇒
   (case a of
    | [ ] add_z ⇒ l2)

| [ ] cons H T ⇒
   (case a of
      | [ ] add_s A1 ⇒
        let [ ] T2 = append ([ ] T) l2 ([ ] A1) in
        [ ] cons H T2)
;
```

Figure 3.3: The append function.

The implementation of append in Figure 3.3 is similar to the simply typed version we could write in SML or Haskell, with the addition of carrying a proof which relates X, Y and Z and the pattern matching on the addition (variable a in the example). It is important to notice that both matches against a have only one branch because there is just one possible branch after matching against the structure of the first list. The proof is thus irrelevant at runtime, and it could be eliminated by a “sufficiently smart compiler”. This optimization is not present in the current version of the compiler [2].

In Figure 3.4 we rewrite the append function more succinctly by using let to replace one branch case expressions.

Proof irrelevance is an important concept in languages with dependent-types, as the erasure of the proofs in compilation eliminates some of the overhead required to con-

2. Ironically, this comment often follows the “sufficiently smart compiler” remark.

```
rec append : (list X) [ ] → (list Y) [ ] → (add X Y Z) [ ] → (list Z) [ ] =
fn l1 ⇒ fn l2 ⇒ fn a ⇒ case l1 of
| [ ] nil ⇒
  let [ ] add_z = a in l2

| [ ] cons H T ⇒
  let [ ] add_s A1 = a in
  let [ ] T2 = append ([ ] T) l2 ([ ] A1) in
  [ ] cons H T2
;
```

Figure 3.4: The append function with `let` expressions.

vince the type-checker that our code has the properties our types say it does. This is something we have to take into account when writing programs, because often there are ways to use proofs in a way they can not be erased, for instance in the following implementation of `append` it is not possible to erase the proof in variable `a`.

```
rec append : (list X) [] → (list Y) [ ] → (add X Y Z) [ ] → (list Z) [ ] =
fn l1 ⇒ fn l2 ⇒ fn a ⇒ case a of
| [ ] add_z ⇒ l2
| [ ] add_s A1 ⇒
  let [ ] cons H T = l1 in
  let [ ] T2 = append ([ ] T) l2 ([ ] A1) in
  [ ] cons H T2
;
```

Figure 3.5: The append function with non-erasable proofs.

Figure 3.5 shows the use of dependent types, how data level terms are defined, and the use of computational-level functions to manipulate data. So far, in this introductory example we used neither binders nor explicit contexts (as the syntax of lists does not itself contain any binders). Those topics will be covered in further examples.

This sample illustrates dependent types and their advantages, but it does not exploit most of the more powerful features of the language. In the next examples we will be able to see some of the unique features of the language.

Some limitations are also apparent from this code, compared to languages with more powerful dependent types like Coq or Agda, the lack of computation in the index language (types can depend on value-level objects not on other computational function)

implies that we have to pass a proof for the sum of lengths instead of just calculating it during type checking. Another limitation is the lack of polymorphism on the data level language and the lack of polymorphic lists as a consequence. Both of these characteristics are related of the objective to support HOAS and the need to be logically sound. So what is a limitation in this example will turn into strengths in the next one.

Beluga was conceived as a language to describe proofs, accordingly its computational functions must be total. The type-checker will signal an error if pattern matching is not exhaustive (technically the error will be signaled by the coverage checker). On the other hand the language supports recursion and the current implementation does not have a termination checker. So in order to consider our functions as proofs we need to argue about termination. This is not a limitation of the theory but just a current technical limitation and work is underway to provide a termination checker using either structural recursion and/or sized types(see [2]).

3.2.2 HOAS to de Bruijn

This example illustrates more features of Beluga, and the compiler. In particular, we show the manipulation of structures with binders. For an untyped λ-calculus we present how we can change the representation of programs that use HOAS to programs that use de Bruijn indices. When talking about programs manipulating other programs and HOAS, we make the distinction between object and meta level languages. The former is the one we are implementing and the latter is Beluga and its contextual LF language.

```
exp : type.
num : nat → exp.
app : exp → exp → exp.
lam : (exp → exp) → exp.
```

Figure 3.6: Untyped λ-calculus.

In Figure 3.6 we give the declaration of our source language, a simple λ-calculus

with support for numbers. Here we declare the `exp` type and its constructors. It is important to notice the declaration of `lam` where the object-level bound variables are represented using meta-level functions.

To represent the identity function in our little language we write: `lam λx.x ;` where we clearly see how to represent the identity function in our object level language by using a function on the meta-level language, in this case `λx.x`.

When representing structures with binders such as programs or logics in Beluga, the natural way as we discussed is to use HOAS. The reasons for this have to do with the supporting infrastructure provided for free in the language. Nonetheless, the user might choose a different representation for whatever reason. In this example we use this as an excuse to explore how HOAS is manipulated. And we show how to convert to a first order representation using de Bruijn indices. We call this representation first order, because it does not use the host language binders to represent its own binders. One important consequence is that now, a lot of the infrastructure provided for free before has to be done by the programmer, things like renumbering of indices, substitution or name generation are now the responsibility of the user.

In this example we show the transformation to a new representation. We begin by defining a new representation as the destination of our transformation. We do so in Figure 3.7 where we define the `exp'` type and its associated constructors. It is important to note that this example also shows that we may want to choose to use our own custom implementation of binders, even if this would mean to avoid one of the strongest features of Beluga. In the proposed calculus, binders are represented with the constructors `one` and `shift` which is a common way of representing de Bruijn indices where `one` is the last bound variable, and we use `shift` to refer to each additional binder (i.e: each `shift` basically performs +1).

With the declaration of the source and target languages we are almost in the position

```
exp' : type.
num' : nat → exp'.
app' : exp' → exp' → exp'.
lam' : exp' → exp'.
one : exp'.
shift : exp' → exp'.
```

Figure 3.7: Untyped λ-calculus with de Bruijn indices.

of writing the computational level function that translates between the two representations. Before writing such a function we need to declare a schema for our contexts because we will be manipulating non-ground terms, i.e. terms with free variables. Schemas classify contexts in the same sense that types classify terms, for this example the declaration is simple as we just need to hold objects of type `exp` in our context.

```
schema ctx = exp ;
```

The final step is the `hoas2db` function in Figure 3.8. The type of the functions from expressions in a context `g` to ground terms of type `exp'`. The destination type does not include a context because there are no binders in that language, therefore there is no context to keep.

```
rec hoas2db : {g:ctx} exp [g] → exp' [ ] =
fn e ⇒ case e of
| [g, x:exp] x ⇒ [ ] one
| [g, x:exp] #p.. ⇒
  let [ ] M' = hoas2db ([g] #p..) in
  [ ] shift M'
| [g] app (M..) (N..) ⇒
  let [ ] M' = hoas2db ([g] M..) in
  let [ ] N' = hoas2db ([g] N..) in
  [ ] app' M' N'
| [g] lam (λx. M..x) ⇒
  let [ ] M' = hoas2db ([g, x:exp] M..x) in
  [ ] lam' M'
| [g] num X ⇒ [ ] num' X
;
```

Figure 3.8: HOAS translation function.

The function proceeds by pattern matching on its parameter, and uses the rich pattern matching language Beluga supports. In particular Beluga supports matching:

- On the shape of the contexts, that is on empty contexts, on contexts with a certain number of elements, or on arbitrary contexts.
- On constructors in the style of ML patterns.
- On particular variables of the context, e.g.: we can pattern match on the last introduced variable or any variable that we list in the context pattern.
- On parameter variables, that is we match on any term as long as it is a name present in the context.
- On meta-variables with substitutions, to check for arbitrary terms that match the substitution.
- On λ patterns, Beluga allows us to pattern match inside abstractions this is an important aspect of the powerful support for HOAS that the language has. That is LF level functions are intensional, meaning their structure can be observed.

The following table shows examples of each kind of pattern present in the `hoas2db` function:

Matching on	Examples from Figure 3.8
The shape of contexts	`[]` for the empty context `[g]` with a context variable for arbitrary contexts `[g, x:exp]` for a context with at least one variable in it.
Constructors	`app` and `lam`.
Variables in the context	`x` the variable of that name in the context pattern.
Parameter variables	`#p..` any variable from the context, the `..` means it can depend only on variables from the context variable.
Meta-variables	`M..x` any term that depends on the context variable plus the variable `x`.
λ-patterns	λ`x.` matches a term that begins with an abstraction and names the variable `x`.

3.2.3 Normalization

This is the last example that we will present, and the motivation is that it combines the features presented in both previous examples and thus is a good running example.

This example consists of a representation for a simply typed λ-calculus (Figure 3.9) and a function to return the normal form of a term (Figure 3.10).

```
tp  : type.
nat : tp.
arr: tp → tp → tp.

exp : tp → type.
lam : (exp T1 → exp T2) → exp (arr T1 T2).
app : exp (arr T2 T) → exp T2 → exp T.

schema ctx = exp T ;
```

Figure 3.9: STLC

The main feature of this representation is that we use dependent types to ensure

that we represent only well-typed terms. Well-typed terms are terms that include the admissible types in their syntax, for example an application in Figure 3.9 has to be a term with function type `T2->T` applied to a term with the type of the parameter `T2` and it produces a term of type `T` exactly as the usual typing rules for STLC prescribe.

```
rec norm : {g:ctx} (exp T) [g] → (exp T) [g] =
fn e ⇒ case e of
| [g] #p.. ⇒ [g] #p..

| [g] lam (λx. M..x) ⇒
  let [g, x:exp _] N..x = norm ([g,x:exp _] M..x) in
  [g] lam λx. N..x

| [g] app (M1..) (M2..) ⇒
  (case norm ([g] M1..) of
   | [g] lam (λx. M'..x) ⇒ norm ([g] M'.. (M2..))
   | [g] N1.. ⇒
     let [g] N2.. = norm ([g] M2..) in
     [g] app (N1..) (N2..))
;
```

Figure 3.10: Normalization function.

This example is interesting because it shows a simple program transformation, and this is the kind of programs that we would like to compile.

3.3 The Beluga Front-end

The compiler complements the current implementation of Beluga.

The Beluga compiler depends on the current implementation, in particular it will reuse the front-end. Beluga's front-end consists of:

- a *parser* for the source level language
- the *type reconstruction* phase to perform type inference, this is a powerful aspect of Beluga as it allows us to write less type annotations.
- the *type checker* that ensures that the program and its subsequent elaborated version are well-typed.

– the *coverage checker* that guarantees that the pattern matching considers all possibilities.

The present implementation also has an *interpreter* that is replaced by the compiler. As a matter of fact the representation fed into the compiler(see Section 3.5 on page 30) is the same as the one used by the interpreter.

Dependent types, as supported by the language, are powerful and expressive, but a naive implementation would require that the programmer specifies many type annotations and variables that could be avoided. However, Beluga supports a powerful type reconstruction algorithm [47] which allows the user to omit as much information as possible. This information is later reconstructed by Beluga itself. As an example, the append function (presented in Figure 3.3 on page 18) gets automatically elaborated to the fully explicit append function shown in Figure 3.11.

```
rec append : {X :: nat[]} {Y :: nat[]} {Z :: nat[]}
                 (list X)[] → (list Y)[] → (add X Y Z)[] → (list Z)[] =
λ□X ⇒ λ□Y ⇒ λ□Z ⇒ fn l1 ⇒ fn l2 ⇒ fn a ⇒
  case l1 of
  | {X :: nat[]} {Y :: nat[]}
    ([] nil) : . Y = Z , . X = Y , . z = X ; ⇒
      (case a of
       | {Y1 :: nat[]}
         ([] add_z Y1) : . Y1 = Y , . Y1 = X ; ⇒
          l2
       )

  | {Z1 :: nat[]} {X1 :: nat[]} {Y1 :: nat[]} {Z :: nat[]} {X :: (list Z1)[]}
    ([] cons Z1 Z X) : . Y1 = Z , . X1 = Y , . s Z1 = X ; ⇒
      (case a of
       | {Z4 :: nat[]} {X4 :: nat[]} {Y4 :: nat[]} {Z3 :: nat[]}
         {X3 :: (list Z4)[]} {Y3 :: (add Z4 X4 Y4)[]}
         ([] add_s Z4 X4 Y4 Y3) : . X3 = X , . Z3 = Z , . s Y4 = Y1 , . X4 =
             X1 ,
          . Z4 = Z1 ; ⇒
           (case append <. Z4> <. X4> <. Y4> ([] X3) l2 ([] Y3) of
            | {X7 :: nat[]} {Y7 :: nat[]} {Z6 :: nat[]} {X6 :: nat[]}
              {Y6 :: (list X7)[]} {Z5 :: (add X7 Y7 Z6)[]} {X5 :: (list
                  Z6)[]}
              ([] X5) : . Z5 = Y3 , . Y6 = X3 , . X6 = Z3 , . Z6 = Y4 ,
               . Y7 = X4 , . X7 = Z4 ; ⇒
               [] cons Z6 X6 X5
            )
       )
```

Figure 3.11: The append function after elaboration.

Reconstruction makes Beluga more user-friendly, otherwise one would have to write a function like the one on Figure 3.11. And the resulting language would be unbearably verbose. In the reconstructed version we can see variables `X`, `Y` and `Z` in the signature of the function. These variables where declared implicitly in the source code have been declared, and their corresponding binders(i.e. $\lambda^{\square}$) explicitly added to the function body. Additionally, after reconstruction all meta-variables are annotated with their types. As explained in detail in [47] type reconstruction is undecidable and sometimes some extra manual annotations are required, but in most cases type reconstruction fills all of the types beyond the signature of the function (as it did for all our simple examples).

3.4 Contextual LF

As already mentioned in 3, Beluga is a two level language. In this section we will present the formalism used at the data level.

The data level is derived from the LF logical framework introduced in [29] with the addition of contextual reasoning from Contextual Modal Logic as introduced in Section 5 of [40]. An in depth discussion of the formal properties of Contextual LF goes beyond the scope of this work. Here we will present the calculus as an input to the compiler. A thorough presentation of this can be found in [46].

We present Contextual LF calculus using the spine notation, as described in [43] and [13] with explicit substitutions as introduced in [1] and described as a tutorial in [55]. This representation is similar to the one used in Twelf [44] with the addition of explicit contexts.

The salient features are:

– dependent types, to encode powerful properties as we have seen in the examples in Sections 3.2.1 and 3.2.3.

– head and spine terms allow for fast access to the head of the term and thus help

with the performance of unification and type-checking.

– explicit substitutions allow for application of the substitutions only when necessary, and in particular to allow meta-variables to refer to a transformed context. We see this use of substitutions in the HOAS to de Bruijn example where in Figure 3.8 the pattern `[g, x:exp] #p..` matches if variable `#p` does not refer to the last introduced variable, a different substitution would be generated for the pattern `[g, x:exp] #p..x` which would match any variable.

3.4.1 Summary of the Techniques Used in the Internal Representation

The internal representation of Beluga programs is described in 3.4.2 and 3.5. To better understand the particular representation of Contextual LF received by the compiler, we will present some ideas in a simplified setting. We will use for this a Simply Typed Lambda Calculus as:

$$\begin{array}{lll} \text{Type} \quad A,B & ::= & P \mid A \to B \\ \text{Term} \quad M,N & ::= & x \mid \lambda x.M \mid M\,N \end{array}$$

This is simple, but as in Beluga there is no computation on the level of values, we might want a representation that only allows normal terms, for that purpose we split terms into normal terms and neutrals in a way that we prevent reducible expressions, that is we cannot represent terms like $(\lambda x.M)N$.

$$\begin{array}{llll} \text{Type} & A,B & ::= & P \mid A \to B \\ \text{Normal Term} & M,N & ::= & \lambda x.M \mid R \\ \text{Neutral Term} & R,S & ::= & x \mid R\,M \end{array}$$

There is an extra technique used in the implementation, that is splitting the Neutral Terms into two categories, heads and spines (technique presented in [43] and also used

in [44]). The objective is to get direct access to the head of an application without having to traverse all the applications before, basically applications in the form $(((c\,M)N)O)$ are converted to $c \cdot M;N;O;\mathsf{nil}$ thus exposing the head and avoiding the nested applications. This is done for simplifying the meta-reasoning and getting an efficient implementation as mentioned in [13].

Type	A,B	$::=$	$P \mid A \rightarrow B$
Normal Term	M,N	$::=$	$\lambda x.M \mid H \cdot S$
Head	H	$::=$	x
Spine	S	$::=$	$M;S \mid \mathsf{nil}$

These three representations serve as an introduction to the techniques used in the internal language of Beluga and thus the compiler. The internal language also uses explicit substitutions, there is a small introduction in Section 4.4.1 and in [1].

3.4.2 Internal Representation of Contextual LF

Kind	K	$::=$	$\mathrm{type} \mid \Pi A.K$
Types	A,B	$::=$	$P \mid \Pi A.B$
Atomic Types	P	$::=$	$a \cdot S$
Normal Terms	M,N	$::=$	$\lambda.M \mid H \cdot S$
Head	H	$::=$	$c \mid x \mid u[\sigma] \mid p[\sigma]$
Spine	S	$::=$	$\mathsf{nil} \mid M\ S$
Substitutions	σ	$::=$	$\uparrow^{c,k} \mid \sigma, M \mid \sigma, H$
Context Shifts	c	$::=$	$-\psi \mid \psi \mid 0$
Contexts	Ψ,Φ	$::=$	$\cdot \mid \psi \mid \Psi, A$
Meta-contexts	Δ	$::=$	$\cdot \mid \Delta, A[\Psi]$

Figure 3.12: Contextual LF

The code produced by the Beluga front-end uses internally de Bruijn indices, notice how λ and Π do not state names for their binders.

Variables, shown as x in Head, that are introduced by λ-terms will be used to represent HOAS. The compiler does not use de Bruijn indices to represent these, in Section 4.4 we show how the compiler transforms the language from Figure 3.12 to a more versatile representation presented in Section 4.4.2.

The data level language distinguishes other two kinds of variables, meta-variables and parameter variables (u and p respectively). The former used as placeholder for terms in the context and the latter as placeholder for variables in the context. These variables will be represented using names, as they are neither used for HOAS nor manipulated at run-time as HOAS variables are.

3.5 Computational Language

Beluga programs consist of types and values declared using the data level language, but also of functions that manipulate them, as we have seen in 3.2. This layer is the computational language presented in Figure 3.13.

The computational language is a dependently typed functional language with support for pattern matching. As we can see in the definition of types, dependent functions can have dependencies on contexts (Π types in Figure 3.13) and dependencies on contextual values ($\Pi^{\Box}$ types in Figure 3.13). The computational language is simple and basically it only provides functions and pattern matching. There are already extensions of Beluga that provide richer type-systems, like Beluga$^{\mu}$ presented in [12].

Types	τ	$::=$	$A[\Psi] \mid \tau_1 \rightarrow \tau_2 \mid \Pi\phi :: W.\tau \mid \Pi^{\Box}u :: A[\Psi].\tau$
Exp. Check.	e	$::=$	$i \mid \mathsf{rec} f.e \mid \mathsf{fn} y.e \mid \Lambda\psi.e \mid \lambda^{\Box}u.e \mid [\![\hat{\Psi}.M]\!] \mid \mathsf{case}\ i\ \mathsf{of}\ \vec{b}$
Exp. Syn.	i	$::=$	$y \mid i\ e \mid i\ \lfloor\Psi\rfloor \mid i\ \lceil\hat{\Psi}.M\rceil \mid e : \tau$
Branch	b	$::=$	$\Pi\Omega'.\Pi\Delta'[\![\hat{\Psi}.M]\!] : \theta, \delta \mapsto e$
Branches	$\vec{b}$	$::=$	$\cdot \mid (b \mid \vec{b})$
Context-var. Context	Ω	$::=$	$\cdot \mid \Omega, \psi : W$
Meta-var. Context	Δ	$::=$	$\cdot \mid \Delta, u : A[\Psi]$
Comp. Contexts	Γ	$::=$	$\cdot \mid \Gamma, y : T$
Meta-substitutions	θ	$::=$	$\cdot \mid \uparrow^n \mid \theta, (M/u) \mid \theta, (H/u)$
Contextual-substitutions	δ	$::=$	$\cdot \mid \uparrow^n \mid \theta, (\Psi/\psi)$

Figure 3.13: Computational language.

CHAPTER 4

THE COMPILER

4.1 The pipeline

The compiler is structured in a pipeline, where each stage reads the Abstract Syntax Tree (AST) of the previous stage and produces a new one with possible a new representation. In particular, new representations are needed for the de Bruijn index and the name based paths.

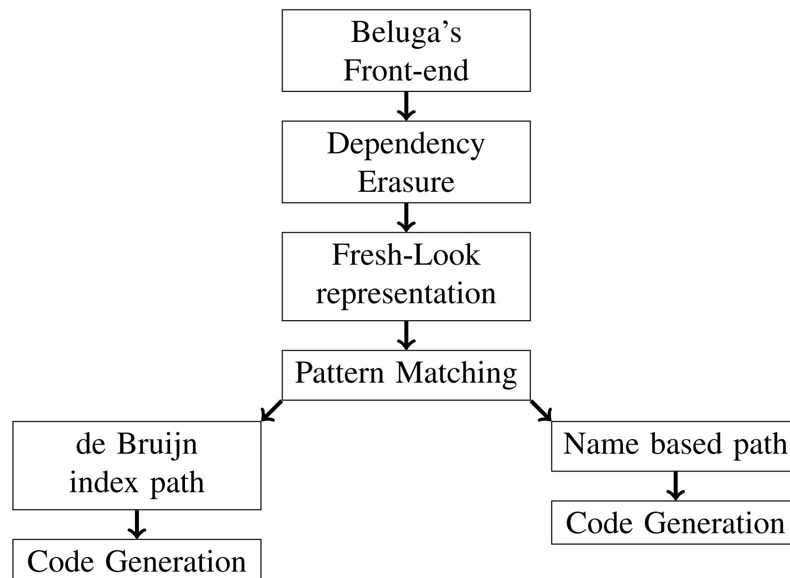

Figure 4.1: The compiler pipeline.

The stages are illustrated in Figure 4.1 and each one will be discussed later in this document. It is important to notice that the first stage is basically the preexisting Beluga implementation, thus all code in this stage is the existing Beluga front-end, and it does all the heavy lifting concerning parsing, type-checking and error reporting. Compilation begins with the internal language after type-checking and reconstruction.

For a language like Beluga, type-checking and reconstruction are important tasks, so much indeed that many programs are not executed but just type-checked. This is the case when Beluga is used as a proof assistant. Nonetheless, many (or perhaps most) use-cases of Beluga are concerned with programming languages, so interpreters, compilers, or transformation phases are programs where execution matters and so they are the focus of the compiler.

Another important consideration is how the source level language, introduced in Section 3.2, relates to the internal languages presented in Figures 3.12 and 3.13. This transformation takes place in the Beluga front-end, and an in-depth discussion is presented in [47]. Dependently typed programs require many type annotations which could require a big effort from the programmer. Beluga minimizes this by performing type reconstruction of the omitted information before type-checking. All these details are hidden from the compiler as it depends on the internal AST.

4.2 Dependency Erasure

The compiler does not deal with dependent types, it drops the dependencies in types. This strategy is common when approaching compilation of dependently typed programs as it is not exactly clear how to scale existing type-preserving compilation techniques (such as [39], [26], and [38]) to dependent types. This is an interesting and active area of research but the focus of this work is on compilation of HOAS and the use of a versatile representation of it.

Dependency erasure is commonly used by some proof assistants that extract functional programs from dependently typed specifications. One salient example is *extraction* in Coq described in [33] and [34] and famously used to implement CompCert a certified C compiler as described in [32].

The implementation of dependency erasure follows the LF literature, a similar era-

$$\begin{array}{lcll} \text{Simple Kinds} & \kappa & ::= & \mathsf{type}^- \mid \tau \to \kappa \\ \text{Simple Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \end{array}$$

Figure 4.2: Simple types.

sure is called *dependency-less translation* in definition A9 in [29], also the *eraser function* in [28] and *erasure operation* in Section 4.3 of [47]. Our erasure operation ($(()^-)$) is presented in Figures 4.2 and 4.3.

$$\begin{array}{rcl} (\mathsf{type})^- & = & \mathsf{type}^- \\ (\Pi A.K)^- & = & (A)^- \to (K)^- \\ \\ (\Pi A_1.A_2)^- & = & (A_1)^- \to (A_2)^- \\ (a \cdot S)^- & = & a \\ \\ (\cdot)^- & = & \cdot \\ (\psi)^- & = & \psi \\ (\Gamma, A)^- & = & (\Gamma)^-, (A)^- \\ \\ (\cdot)^- & = & \cdot \\ (\Delta, A[\Psi])^- & = & (\Delta)^-, (A^-)[(\Psi)^-] \\ \\ (A[\Psi])^- & = & (A)^-[(\Psi)^-] \\ (\tau_1 \to \tau_2)^- & = & (\tau_1)^- \to (\tau_2)^- \\ (\Pi^\square u :: A[\Psi].\tau)^- & = & (A)^-[(\Psi)^-] \to (\tau)^- \\ (\Pi \phi :: W.\tau)^- & = & \Pi \phi :: W.(\tau)^- \end{array}$$

Figure 4.3: Dependency erasure.

The dependency erasure operation does not eliminate all forms of dependency, as we see in the case shown in Figure 4.3, we do not eliminate the dependencies on contexts and by doing this it is possible to still type-check the resulting representation.

The result of this phase is a simply typed representation, that we can type-check after each stage as a way of sanity check. This is useful even as we do not prove type preservation in each phase as in [25].

In the implementation of the compiler, this phase and the next one are performed at

the same time with just one code walk, so the resulting representation will be discussed in the next section.

4.3 The Fresh-Style Representation

The representation of bound variables in the compiler is based on an adaptation and simplification of the *Fresh-Look* already discussed in 2.4.2. There are two important differences: the first one, as we do not have dependent types in our implementation language; the OCaml implementation is not able to enforce all the compile type invariants that the version by the original authors does [54]. The second difference is about the objective, in this work the intention is to use a representation that does not commit to the concrete implementation of binders, so that the compiler can have optimizations, and important phases like pattern matching compilation using this abstract representation.

```
type world
type name
type link

val empty : world
val world : name → world
val fresh : world → link
val name_of : link → name
val import : link → name → name

val name_to_db : name → int
val name_to_name : name → int option
```

Figure 4.4: The fresh-style module interface.

As presented in Figure 4.4 there are three abstract concepts:

- names: that are an abstract representation of bound variable names. Each name inhabits a world.
- worlds: are inhabited by names, and each world contains countable infinite many names. Names in different worlds should not be part of the same expression, with some exceptions for expressions which introduce new binders.

– links: are inclusion relations between two worlds and each link introduces exactly one new name in the destination world.

One difference with regards to the original presentation [54] is that our approach only has one kind of links that corresponds to *strong links* which do not support shadowing, and we ignore *weak links*. This is simplification is acceptable simply because we have no need for reusing variable names.

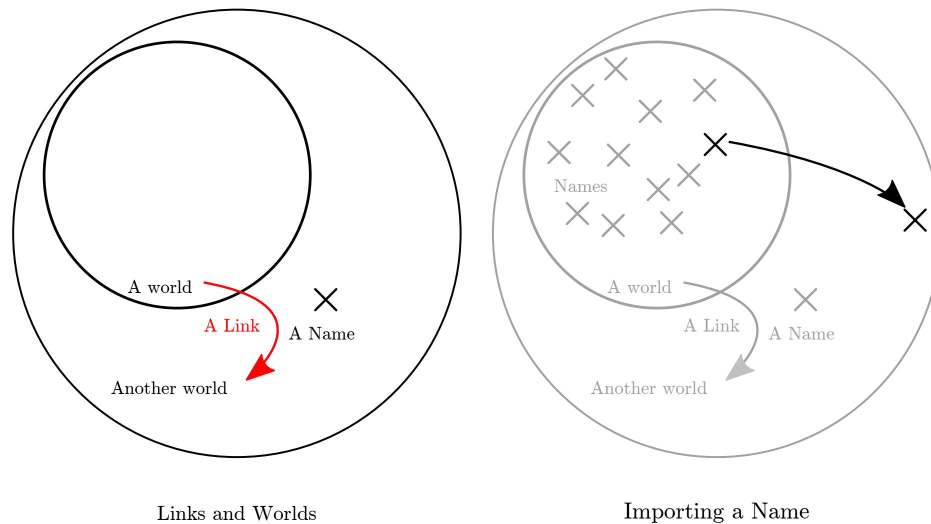

Figure 4.5: The fresh-style approach.

In the first pane of Figure 4.5 we see a diagram of two worlds related by a link. The link not only relates to worlds but it also introduces a new name that inhabits the bigger world and not the smaller world. As the diagram suggests, the *link* implies a form of inclusion between worlds. This is because all the names in the smaller world can be *imported* to the bigger world, an operation illustrated by the second pane of Figure 4.5.

These concepts form the basis of the approach, as we use *worlds* as scopes, and *links* to get into the inner scopes of expressions with binders, the more typical example being λ-terms. Inner scopes are bigger because they contain one extra name, introduced by the link, and because all the names in the former world can be brought to the latter by using *import*.

The rest of the interface functions are less important:

– *empty* : returns a world with no names in it.

– *fresh* : given a world, returns the link to a bigger world.

– *name_of* : given a link returns the name it introduces to the bigger world.

The final two functions *name_to_db* and *name_to_name* are the functions that we will use to extract de Bruijn indices and plain-old names from the abstract *names* in the Fresh-Style.

In the next section we are going to discuss how to translate the AST we receive from the front-end to one where we use Fresh-Style for representing variables. It is important to notice that dependency erasure and Fresh-Style translation are done at the same time by the compiler. We will also see, how after the translation we will be able to check the *well-worldness* of terms, i.e. that we do not mix variables from different scopes.

4.4 Translating to the Fresh-Style Representation

4.4.1 A Simple Example

In order to get familiar with the Fresh-Style representation we will describe how to use Fresh-Style variables in an untyped λ-calculus with explicit substitutions. This follows a similar presentation from [1] but instead of using de Bruijn indices we use Fresh-Style variables.

$$\begin{array}{lll} \text{Term} & M_\alpha, N_\alpha & ::= \quad x_\alpha \mid M_\delta[\overset{\delta\to\alpha}{\sigma}] \mid \lambda\alpha \leftharpoonup \beta . M_\beta \mid M_\alpha N_\alpha \\ \text{Substitutions} & \overset{\alpha\to\beta}{\sigma} & ::= \quad id \mid \uparrow^n \mid \overset{\gamma\to\beta}{\sigma}, M_\beta \end{array}$$

Figure 4.6: Fresh-style $\lambda\sigma$-calculus.

Figure 4.6 presents the syntax of the calculus. The main idea is that terms of this

calculus exist in a world, i.e. if a term exists in a world α then all sub-terms belong to the same world. Nonetheless, terms which introduce binders are an exception. In this case only λ-terms contain a link to a different world and its body will be in this new world. Additionally, if there is a link between worlds, names can be imported from the smaller to the bigger world.

Furthermore, this example uses explicit substitutions, as they are used by Beluga's elaborated internal representation. The purpose of substitutions is to avoid the full evaluation of terms when they are not used and to keep track of the relationships among contexts. Much of the complexity of the translation is due to the presence of substitutions, so this example will serve as an introduction and then the complete translation strategy will be easier to follow.

Substitutions move terms between worlds, we represent them by $\overset{\alpha\to\beta}{\sigma}$ to show that σ is a substitution that takes terms in the world α to the world β.

$$(\lambda\alpha \leftharpoondown \beta.M_\beta)N_\alpha \Longrightarrow_R M_\beta[id, N_\alpha]$$

Figure 4.7: β-reduction with explicit substitutions.

This calculus has two kinds of substitutions, first the shift substitution ($\uparrow^n$) that basically imports all names in the term to the nth bigger world, so if there is a link $\alpha \leftharpoondown \beta$ the substitution $M_\alpha[\uparrow]$ will result in a term in the world β. The *id* substitution is a short form of $\uparrow^0$ that does not change the term it is applied to. The second substitution is

$$\begin{array}{lcl}
M[\overset{\alpha\to\beta}{\sigma_1}][\overset{\beta\to\gamma}{\sigma_2}] & = & M[\overset{\alpha\to\beta}{\sigma_1} \circ \overset{\beta\to\gamma}{\sigma_2}] \\
(\lambda\alpha \leftharpoondown \beta.M_\beta)[\overset{\alpha\to\gamma}{\sigma}] & = & \lambda\gamma \leftharpoondown \delta.(M_\beta[\overset{\alpha\to\gamma}{\sigma} \circ \uparrow,\ d_\delta]) \text{ where } d_\delta \text{ is the name in } \gamma \leftharpoondown \delta \\
(M_\alpha N_\alpha)[\overset{\alpha\to\beta}{\sigma}] & = & M_\alpha[\overset{\alpha\to\beta}{\sigma}]\, N_\alpha[\overset{\alpha\to\beta}{\sigma}]
\end{array}$$

Figure 4.8: Explicit substitution equations.

$\overset{\alpha\to\beta}{\sigma_1} = (\overset{\gamma\to\beta}{\sigma_2}, M_\beta)$ this assumes that there is a link $\alpha \leftharpoondown \gamma$ and it substitutes M_β for all the appearances of the name introduced by $\alpha \leftharpoondown \gamma$, recursively applies $\overset{\gamma\to\beta}{\sigma_2}$ to the rest of the term.

$$\begin{array}{llcl}
id & \circ \; \overset{\alpha\to\beta}{\sigma} & = & \overset{\alpha\to\beta}{\sigma} \\
\overset{\alpha\to\beta}{\uparrow^{n}} & \circ \; \overset{\beta\to\gamma}{\uparrow^{k}} & = & \overset{\alpha\to\gamma}{\uparrow^{n+k}} \\
\overset{\alpha\to\beta}{\uparrow^{n+1}} & \circ \; \overbrace{(\overset{\delta\to\gamma}{\sigma}, M_\gamma)}^{\beta\to\gamma} & = & \overset{\alpha\to\delta}{\uparrow^{n}} \circ \overset{\delta\to\gamma}{\sigma} \\
(\overset{\gamma\to\beta}{\sigma_1}, M_\beta) & \circ \; \overset{\beta\to\delta}{\sigma_2} & = & (\overset{\gamma\to\beta}{\sigma_1} \circ \overset{\beta\to\delta}{\sigma_2},\ M_\beta[\overset{\beta\to\delta}{\sigma_2}])
\end{array}$$

Figure 4.9: Substitution composition for $\lambda\sigma$-calculus.

The main idea of explicit substitutions is to not perform eagerly the substitutions during β-reduction but instead accumulating them in a big substitution, that is applied when/if needed. Figure 4.7 shows how beta reduction is performed. Figure 4.8 shows equations that allow us to push substitutions inside terms, these equations use the substitution composition defined in 4.9. In the next figure we show how worlds are preserved when we push a substitution inside a λ-expression.

$$\overset{\gamma}{\overline{(\lambda\alpha \leftharpoondown \beta.M_\beta)[\overset{\alpha\to\gamma}{\sigma}]}} = \underset{\gamma}{\underline{\lambda\gamma \leftharpoondown \delta.(\underset{\delta}{\underline{\overset{\beta\to\delta}{\overline{M_\beta\,[\overset{\alpha\to\delta}{\overline{\overset{\alpha\to\gamma}{\sigma} \circ \overset{\gamma\to\delta}{\uparrow}}},\ d_\delta]}}}})}}$$

In the previous figure we annotate each term and substitution with their corresponding worlds in order to show how the worlds are preserved when applying this transformation, while the general structure corresponds to the de Bruijn case almost directly.

The objective is not postponing application, but to compose substitutions thus reducing the number of traversals of a potentially large term. Sometimes the term will not be used anymore, or only the head of the term will be required for pattern matching. This

leads us to the final ingredient of the system: Weak Head Normal Form (WHNF).

$$
\begin{array}{llcll}
\mathsf{whnf} & (\lambda\alpha \leftharpoondown \beta.M_\beta, \overset{\alpha\to\gamma}{\sigma}) & = & (\lambda\alpha \leftharpoondown \beta.M_\beta, \overset{\alpha\to\gamma}{\sigma}) & \\
\mathsf{whnf} & (M_\alpha N_\alpha, \overset{\alpha\to\gamma}{\sigma_1}) & = & \mathsf{whnf}(M'_\rho, (\overset{\delta\to\gamma}{\sigma_2}, N_\alpha[\overset{\alpha\to\gamma}{\sigma_1}])) & \text{if } \mathsf{whnf}(M_\alpha, \overset{\alpha\to\gamma}{\sigma_1}) = \\
 & & & & (\lambda\delta \leftharpoondown \rho.M'_\rho, \overset{\delta\to\gamma}{\sigma_2}) \\
\mathsf{whnf} & (M_\alpha N_\alpha, \overset{\alpha\to\gamma}{\sigma_1}) & = & (M'_\gamma(N_\alpha[\overset{\alpha\to\gamma}{\sigma_1}]), id) & \text{if } \mathsf{whnf}(M_\alpha, \overset{\alpha\to\gamma}{\sigma_1}) = (M'_\gamma, id) \\
\mathsf{whnf} & (x_\alpha, id) & = & (x_\alpha, id) & \\
\mathsf{whnf} & (y_\alpha, \uparrow^n) & = & \mathsf{whnf}(\underbrace{\mathsf{import}\ \alpha \leftharpoondown \beta\ y_\alpha}_{y_\beta}, \uparrow^{n-1}) & \text{where } \alpha \leftharpoondown \beta \\
\mathsf{whnf} & (M_\alpha[\overset{\alpha\to\gamma}{\sigma_1}], \overset{\gamma\to\rho}{\sigma_2}) & = & \mathsf{whnf}(M_\alpha, \overset{\alpha\to\gamma}{\sigma_1} \circ \overset{\gamma\to\rho}{\sigma_2}) & \\
\mathsf{whnf} & (a_\alpha, (\overset{\beta\to\gamma}{\sigma_1}, M_\delta[\overset{\delta\to\gamma}{\sigma_2}])) & = & \mathsf{whnf}(M_\delta, \overset{\delta\to\gamma}{\sigma_2}) & \text{where } a_\alpha \text{ is the new name in } \alpha \\
\mathsf{whnf} & (x_\alpha, (\overset{\beta\to\gamma}{\sigma_1}, M_\delta[\overset{\delta\to\gamma}{\sigma_2}])) & = & \mathsf{whnf}(x_\beta, \overset{\beta\to\gamma}{\sigma_1}) & \text{where } x_\alpha \text{ is not the new name in } \alpha
\end{array}
$$

Figure 4.10: $\lambda\sigma$ WHNF.

The objective of WHNF is to be able to evaluate only the head of the term by pushing the substitutions inside it, Figure 4.10 shows how to implement this functionality for the $\lambda\sigma$-calculus. For the last two cases we have to make a distinction, if the name was introduced when this world was created, then it must be replaced by the term M_δ. Otherwise, it is a different name and we continue with the substitution recursively.

At this point we have presented all we need for a calculus with explicit substitutions that uses binders with the fresh-style representation. Even if in this section we used a rather simple calculus, in the next section we will extend these ideas to Contextual-LF and we will present a translation from the de Bruijn based representation presented in Figure 3.12 to the Fresh-Style.

4.4.2 The Simply Typed Fresh Style Contextual LF Syntax

Figure 4.11 presents the syntax of the representation of Beluga programs that we use after the dependency elimination and translation to the Fresh-Style.

Type	A, B	$::=$	$P \mid A \to B$
Atomic Type	P	$::=$	a
Normal Term	M_α, N_α	$::=$	$\lambda \alpha \leftharpoondown \beta . M_\beta \mid H_\alpha \cdot S_\alpha$
Head	H_α	$::=$	$\mathsf{const} \mid \mathsf{n}_\alpha \mid u[\overset{\gamma\to\alpha}{\sigma}] \mid p[\overset{\gamma\to\alpha}{\sigma}]$
Spine	S_α	$::=$	$\text{nil} \mid M_\alpha S_\alpha$
Substitution	$\overset{\alpha\to\beta}{\sigma}$	$::=$	$\uparrow^{c,k} \mid \overset{\gamma\to\beta}{\sigma}, M_\beta \mid \overset{\gamma\to\beta}{\sigma}, H_\beta$
Context Shift	c	$::=$	$-\psi_{[\cdot]} \mid \psi_{[\cdot]} \mid 0$
Context	Ψ_α, Φ_α	$::=$	$\cdot \mid \Psi_\beta, \beta \leftharpoondown \alpha : A \mid \psi_{[\cdot]}$
Meta-Context	Δ	$::=$	$\cdot \mid \Delta, u : A[\Psi_\gamma]$

Figure 4.11: Fresh-style contextual LF.

Beluga uses several different kinds of variables:

- Computational variables that are part of the computational language.
- Meta-variables ($u[\overset{\gamma\to\alpha}{\sigma}]$ in the grammar) that represent other LF terms.
- Parameter variables ($p[\overset{\gamma\to\alpha}{\sigma}]$) which are similar to meta-variables but represent only variables from the context and not arbitrary LF terms.
- Context variables($\psi_{[\cdot]}$) which represent contexts.
- Finally, LF variables (n_α in the grammar) these are the only variables that will be represented with the Fresh-Style binders the rest will use plain old names for their representations.

Everything in this section will be analogous to the $\lambda\sigma$-calculus but in the more complex setting of simply typed Contextual-LF. After the dependency elimination, we perform the translation to this calculus so all terms and contexts get annotated with the world they inhabit and we get a spine calculus with explicit substitutions. One simplification of this presentation is that in Figure 4.11 we only show substitutions associated

to meta-variables as we will consider explicit substitutions only at runtime. Also, contextual variables exist on an abstract world as we can not say anything about it until we instantiate the variable. To denote this we annotate ψ with $\psi_{[\cdot]}$.

The Beluga core uses explicit substitutions in the interpreter and also as part of type reconstruction. Nonetheless, the internal representation that we work on core has all the substitutions applied and thus the compiler will not need to apply them. On the other hand the runtime uses them for all the same reasons that we mentioned in the presentation of explicit substitutions in Section 4.4.1.

4.4.3 Translating Beluga Expressions to the Fresh-Style

Figures 4.12, 4.13 and 4.14 show the definition of the translation for contexts, terms and substitutions respectively. The simplest case is the translation of contexts where we recursively add links for each element starting from the empty world, this function does not call the two others. However, the other two functions trans and $\mathsf{transSubst}$ are mutually recursive and they translate terms and substitutions. The trans function is straight-forward and the only interesting detail is how the body of a λ-abstraction gets translated inside the bigger context. Finally substitutions are a bit more complex because they move terms between two different worlds related by an arbitrary long chain of links. Substitutions in the fresh-style (e.g.: $\overset{\alpha\to\beta}{\sigma}$) have domain world α and range world β. The function $\mathsf{transSubst}$ takes the context in the range world, and returns the context in the domain world.

So, having the dependency erasure and the translation in place, we now can translate from the representation in Figure 3.12 to the one in Figure 4.11. The focus is on the data-level language as this is the only one with variables in the fresh-style as all other variables (context, computational and meta-variables) use a name based representation. Computational level terms change in that they now have world annotations for contexts

$$
\begin{array}{llcl}
\textsf{transCtx} :: & \textsf{ctx} \rightarrow & & \textsf{ctx}_\alpha \\
\textsf{transCtx} & \cdot & = & \cdot \\
\textsf{transCtx} & (\Psi, A) & = & \textsf{let } \Psi'_\beta = \textsf{transCtx } \Psi \textsf{ in } (\Psi'_\beta, \textsf{fresh } \beta : A) \\
\textsf{transCtx} & \psi & = & \psi_{[\cdot]}
\end{array}
$$

Figure 4.12: Translation of contexts.

$$
\begin{array}{lllcl}
\textsf{trans} :: & \textsf{ctx} \rightarrow & \textsf{term} & \rightarrow & \textsf{term}_\alpha \\
\textsf{trans} & \Psi_\alpha & \lambda.M & = & \lambda(\textsf{fresh } \alpha).\textsf{trans } (\Psi_\alpha, \textsf{fresh } \alpha)M \\
\textsf{trans} & \Psi_\alpha & (H \cdot S) & = & (\textsf{trans } \Psi_\alpha\, H) \cdot (\textsf{trans } \Psi_\alpha\, S) \\
\textsf{trans} & \Psi_\alpha & c & = & c \\
\textsf{trans} & \Psi_\alpha & x & = & \textsf{name_of}(\Psi_\alpha(x)) \\
\textsf{trans} & \Psi_\alpha & u[\sigma] & = & u[\overset{\gamma \rightarrow \alpha}{\sigma_1}] \text{ where} \\
 & & & & \textsf{transSubst } \Psi_\alpha\, \sigma = (\overset{\gamma \rightarrow \alpha}{\sigma_1}, _) \\
\textsf{trans} & \Psi_\alpha & p[\sigma] & = & p[\overset{\gamma \rightarrow \alpha}{\sigma_1}] \text{ where} \\
 & & & & \textsf{transSubst } \Psi_\alpha\, \sigma = (\overset{\gamma \rightarrow \alpha}{\sigma_1}, _) \\
\textsf{trans} & \Psi_\alpha & \textsf{nil} & = & \textsf{nil} \\
\textsf{trans} & \Psi_\alpha & MS & = & (\textsf{trans } \Psi_\alpha\, M)(\textsf{trans } \Psi_\alpha\, S)
\end{array}
$$

Figure 4.13: Translation of terms.

$$
\begin{array}{lllll}
\mathsf{transSubst} :: & \mathsf{ctx} \to & \mathsf{subst} & \to & \overset{\alpha\to\beta}{\mathsf{subst}} * \mathsf{ctx}_\beta \\
\mathsf{transSubst} & \Psi_\alpha & \uparrow^{0,0} & = & (\uparrow^{0,0}, \Psi_\alpha) \\
\mathsf{transSubst} & \Psi_\alpha & \uparrow^{0,k} & = & (\uparrow^{0,k}, (\mathsf{transSubst}\ (\Psi_\alpha,\ \mathsf{fresh}\ \alpha) \uparrow^{0,k-1}).2) \\
\mathsf{transSubst} & \psi_{[\cdot]} & \uparrow^{\psi,0} & = & (\uparrow^{\psi_{[\cdot]},0}, \cdot) \\
\mathsf{transSubst} & \cdot & \uparrow^{-\psi,0} & = & (\uparrow^{-\psi_{[\cdot]},0}, \psi_{[\cdot]}) \\
\mathsf{transSubst} & \psi_{[\cdot]} & \uparrow^{\psi,k} & = & (\uparrow^{\psi_{[\cdot]},k}, \Psi_0) \text{ where } |\Psi_0| = k \\
\mathsf{transSubst} & \cdot & \uparrow^{-\psi,k} & = & (\uparrow^{-\psi_{[\cdot]},k}, \psi_{[\cdot]}, \Psi_0) \text{ where } |\Psi_0| = k \\
\mathsf{transSubst} & \Psi_\alpha & (\mathsf{id}, M) & = & ((\mathsf{id}, \mathsf{trans}\ \Psi_\alpha\ M), \Psi_\alpha) \text{ where } \Psi_\alpha \text{ may be empty} \\
\mathsf{transSubst} & \Psi_\alpha & (\mathsf{id}, H) & = & ((\mathsf{id}, \mathsf{trans}\ \Psi_\alpha\ H), \Psi_\alpha) \text{ where } \Psi_\alpha \text{ may be empty} \\
\mathsf{transSubst} & \Psi_\alpha & (\sigma, M) & = & ((\overset{\gamma\to\alpha}{\sigma'}, \mathsf{trans}\ \Psi_\alpha M), (\Psi'_\gamma, \mathsf{fresh}\ \gamma)) \text{ where} \\
 & & & & \mathsf{transSubst}\ \Psi_\alpha\ \sigma\ = (\overset{\gamma\to\alpha}{\sigma'}, \Psi'_\gamma) \\
\mathsf{transSubst} & \Psi_\alpha & (\sigma, H) & = & ((\overset{\gamma\to\alpha}{\sigma'}, \mathsf{trans}\ \Psi_\alpha H), (\Psi'_\gamma, \mathsf{fresh}\ \gamma)) \text{ where} \\
 & & & & \mathsf{transSubst}\ \Psi_\alpha\ \sigma\ = (\overset{\gamma\to\alpha}{\sigma'}, \Psi'_\gamma)
\end{array}
$$

Figure 4.14: Translation of substitutions.

and contextual LF terms.

Figure 4.15 shows the internal representation that the compiler uses for the computational language. This language will be expanded with new simpler terms to represent the pattern matching compilation that will be discussed in Section 4.5.

4.4.4 Typing Rules for the Simply-Typed Calculus

There are many options when dealing with types in a compiler, the simplest approach would be to erase all type information after type-checking. However it is a good idea to keep all or some information to help with the development. Again there are many options on how much typing information to keep, also how to relate the typing of the program at the beginning and the end of each phase. For example, type preserving compilation requires the type before and after each phase to be equal. In our compiler we take

Types	T	$::=$	$A[\Psi_\alpha] \mid T_1 \to T_2 \mid A[\Psi_\alpha] \stackrel{\Box}{\to} T \mid \Pi\psi{:}W.T \mid T_1 * T_2 * \cdots * T_n$
Expr. Synth.	I	$::=$	$x \mid IE \mid I\lceil\hat{\Psi}_{\hat{\alpha}}.M_\alpha\rceil \mid I\lfloor\Psi_\alpha\rfloor \mid E:T$
Expr. Checked	E	$::=$	$I \mid \mathsf{rec}\ f.E \mid \mathsf{fn}\ x.E \mid \lambda^{\Box}\ X.E \mid \Lambda\ \psi_{[\cdot]}.E \mid [\![\hat{\Psi}_{\hat{\alpha}}.M_\alpha]\!] \mid$
			$\mathsf{case}\ I\ \mathsf{of}\ \vec{B} \mid (E_1, \ldots, E_n)$
Branches	$\vec{B}$	$::=$	$\cdot \mid (B\vert\vec{B})$
Branch	B	$::=$	$\Pi\Omega'.\Pi\Delta'.[\![\hat{\Psi}_{\hat{\alpha}}.M_\alpha]\!] : \theta, \delta \mapsto E$
Context-var. Context	Ω	$::=$	$\cdot \mid \Omega, \psi{:}W$
Metavar. Context	Δ	$::=$	$\cdot \mid \Delta, u{:}A[\Psi_\alpha] \mid \Delta, p{:}A[\Psi_\alpha]$
Comp. Context	Γ	$::=$	$\cdot \mid \Gamma, x:T$
Context Schema	W	$::=$	$\mathbb{N}$
Meta-substitutions	θ	$::=$	$\cdot \mid \uparrow^n \mid \theta, (M_\alpha/u) \mid \theta, (M_\alpha/p) \mid \theta, (H_\alpha/u) \mid \theta, (H_\alpha/p)$
Contextual-substitutions	δ	$::=$	$\cdot \mid \uparrow^n \mid \theta, (\Psi_\alpha/\psi_{[\cdot]})$

Figure 4.15: Computational language internal representation.

a simple approach, keeping only simple types(the types that result from dependency erasure) and not trying to establish type preservation on each phase. This approach is simple, but it is also helpful in the development of the compiler, catching lots of bugs in the different phases of the compiler. One consequence of deciding to keep simple types is that we need the typing rules for the program after the dependency erasure phase.

Since the compiler works with the output generated by the Beluga front-end it receives a well-typed fully explicit program with all the substitutions applied; the only purpose of the typing rules for the fresh-style representation is to ensure that each phase of the compiler results in a well-typed program. This proved helpful to find bugs and problems during the development. The type-checker is a simple bidirectional type-checker similar to the one presented in [47] but using our simple types after the dependency erasure. The typing rules can be classified in two categories, the first is for terms for which we can synthesize their types, i.e.: we can establish their type from the information in the term itself and in the context ($\Psi_\alpha \vdash R_\alpha \Rightarrow A$ is read as we can synthesize type A from the term M_α). The second category is for rules that check a term against a provided type, for $\Psi_\alpha \vdash M_\alpha \Leftarrow A$ we say that M_α checks against type A.

Typing rules for normal terms:

$$\frac{\Delta;\Psi_\alpha,\alpha \leftharpoondown \beta : A \vdash M_\beta \Leftarrow B}{\Delta;\Psi_\alpha \vdash \lambda\alpha \leftharpoondown \beta . M_\beta \Leftarrow A \to B} \qquad \frac{\Delta;\Psi_\alpha \vdash H_\alpha \Rightarrow A \quad \Delta;\Psi_\alpha \vdash S_\alpha : A \Leftarrow P}{\Delta;\Psi_\alpha \vdash H_\alpha \cdot S_\alpha \Leftarrow P}$$

Typing rules for heads:

$$\frac{\Sigma(c) = A}{\Delta;\Psi_\alpha \vdash c \Rightarrow A} \qquad \frac{\Delta;\Psi_\alpha(\mathsf{n}_\alpha) = A}{\Delta;\Psi_\alpha \vdash \mathsf{n}_\alpha \Rightarrow A}$$

$$\frac{\Delta(u) = A[\Phi_\gamma] \quad \Delta,\Psi_\alpha \vdash \overset{\gamma\to\alpha}{\sigma} \Leftarrow \Phi_\gamma}{\Delta;\Psi_\alpha \vdash u[\overset{\gamma\to\alpha}{\sigma}] \Rightarrow A} \qquad \frac{\Delta(p) = A[\Phi_\gamma] \quad \Delta,\Psi_\alpha \vdash \overset{\gamma\to\alpha}{\sigma} \Leftarrow \Phi_\gamma}{\Delta;\Psi_\alpha \vdash p[\overset{\gamma\to\alpha}{\sigma}] \Rightarrow A}$$

Typing rules for spines:

$$\frac{}{\Delta;\Psi_\alpha \vdash \mathsf{nil} : P \Leftarrow P} \qquad \frac{\Delta;\Psi_\alpha \vdash M_\alpha \Leftarrow A \quad \Delta;\Psi_\alpha \vdash S_\alpha : B \Leftarrow P}{\Delta;\Psi_\alpha \vdash M_\alpha S_\alpha : A \to B \Leftarrow P}$$

Typing rules for substitutions:

$$\frac{}{\Delta;\psi_{[\cdot]} \vdash \uparrow^{\psi_{[\cdot]},0} \Leftarrow \cdot} \qquad \frac{}{\Delta;\cdot \vdash \uparrow^{-\psi_{[\cdot]},0} \Leftarrow \psi_{[\cdot]}}$$

$$\frac{\Delta;\Psi_\beta \vdash \uparrow^{c,k} \Leftarrow \Psi_\gamma}{\Delta;\Psi_\beta,\beta \leftharpoondown \alpha \vdash \uparrow^{c,k+1} \Leftarrow \Psi_\gamma} \qquad \frac{\Delta;\Psi'_\alpha = \Psi_\beta,\Psi_0 \quad |\Psi_0| = k}{\Delta;\Psi'_\alpha \vdash \uparrow^{0,k} \Leftarrow \Psi_\beta}$$

$$\frac{\Delta;\Psi'_\gamma \vdash \overset{\beta\to\gamma}{\sigma} \Leftarrow \Psi_\beta \quad \Delta;\Psi'_\gamma \vdash M_\gamma \Leftarrow A}{\Delta;\Psi'_\gamma \vdash \underbrace{\overset{\beta\to\gamma}{\sigma}, M_\gamma}_{\overset{\alpha\to\gamma}{\sigma}} \Leftarrow \Psi_\beta,\beta \leftharpoondown \alpha : A} \qquad \frac{\Delta;\Psi'_\gamma \vdash \overset{\beta\to\gamma}{\sigma} \Leftarrow \Psi_\beta \quad \Delta;\Psi'_\gamma \vdash H_\gamma \Rightarrow A}{\Delta;\Psi'_\gamma \vdash \underbrace{\overset{\beta\to\gamma}{\sigma}, H_\gamma}_{\overset{\alpha\to\gamma}{\sigma}} \Leftarrow \Psi_\beta,\beta \leftharpoondown \alpha : A}$$

Figure 4.16: Typing rules for simply typed contextual-LF.

$$\frac{\Omega;\Delta;\Gamma, f:T \vdash E \Leftarrow T}{\Omega;\Delta;\Gamma \vdash \mathsf{rec}\ f.E \Leftarrow T} \qquad \frac{\Omega;\Delta;\Gamma, y:T_1 \vdash E \Leftarrow T_2}{\Omega;\Delta;\Gamma \vdash \mathsf{fn}\ y.E \Leftarrow T_1 \to T_2}$$

$$\frac{\Omega;\Delta, X:A[\Psi_\alpha];\Gamma \vdash E \Leftarrow T}{\Omega;\Delta;\Gamma \vdash \mathsf{mlam}\ X.E \Leftarrow A[\Psi_\alpha] \stackrel{\Box}{\to} T} \qquad \frac{\Omega, \psi:W;\Delta;\Gamma \vdash E \Leftarrow T}{\Omega;\Delta;\Gamma \vdash \Lambda\ \psi.E \Leftarrow \Pi\psi:W.T}$$

$$\frac{\Omega;\Delta;\Gamma \vdash I \Rightarrow T \quad T = T'}{\Omega;\Delta;\Gamma \vdash I \Leftarrow T'} \qquad \frac{\Omega;\Delta;\Gamma \vdash I \Rightarrow T_1 \quad \Omega;\Delta;\Gamma, x:T_1 \vdash E \Leftarrow T}{\Omega;\Delta;\Gamma \vdash \mathsf{let}\ x = I\ \mathsf{in}\ E \Leftarrow T}$$

$$\frac{\Omega;\Delta;\Gamma \vdash I \Rightarrow T_1 \quad \Omega;\Delta, X:T_1;\Gamma \vdash E \Leftarrow T}{\Omega;\Delta;\Gamma \vdash \mathsf{mlet}\ X = I\ \mathsf{in}\ E \Leftarrow T} \qquad \frac{\Delta;\Psi_\alpha \vdash M_\alpha \Leftarrow A}{\Omega;\Delta;\Gamma \vdash [\![\hat{\Psi}_{\hat{\alpha}}.M_\alpha]\!] \Leftarrow A[\Psi_\alpha]}$$

$$\frac{\Gamma(y) = T}{\Omega;\Delta;\Gamma \vdash y \Rightarrow T} \qquad \frac{\Omega;\Delta;\Gamma \vdash I \Rightarrow T_2 \to T \quad \Omega;\Delta;\Gamma \vdash E \Leftarrow T_2}{\Omega;\Delta;\Gamma \vdash IE \Rightarrow T}$$

$$\frac{\Omega;\Delta;\Gamma \vdash I \Rightarrow A[\Psi_\alpha] \stackrel{\Box}{\to} T \quad \Delta;\Psi_\alpha \vdash M_\alpha \Leftarrow A}{\Omega;\Delta;\Gamma \vdash I\lceil \hat{\Psi}_{\hat{\alpha}}.M_\alpha \rceil \Rightarrow T} \qquad \frac{\Omega;\Delta;\Gamma \vdash I \Rightarrow \Pi\psi:W.T \quad \Omega;\Delta \vdash \Psi_\alpha \Leftarrow W}{\Omega;\Delta;\Gamma \vdash I\lfloor \Psi_\alpha \rfloor \Rightarrow [\Psi_\alpha/\psi]T}$$

$$\frac{\Omega;\Delta;\Gamma \vdash E \Leftarrow T}{\Omega;\Delta;\Gamma \vdash E:T \Rightarrow T}$$

$$\frac{\Omega;\Delta;\Gamma \vdash I \Rightarrow A[\Psi] \quad \text{for all } k\ \Omega;\Delta;\Gamma \vdash B_k \Leftarrow A[\Psi] \to T}{\Omega;\Delta;\Gamma \vdash \mathsf{case}\ I\ \mathsf{of}\ B_1|\ldots|B_n \Leftarrow T}$$

$$\frac{\Omega';\Delta';[\delta]\Psi_\alpha \vdash M_\alpha \Leftarrow A \quad \Omega';\Delta';[\delta]\Gamma \vdash E \Leftarrow [\delta]T \quad \Omega' \vdash \delta \Leftarrow \Omega}{\Omega;\Delta;\Gamma \vdash \Pi\Omega'.\Pi\Delta'.[\![\hat{\Psi}_{\hat{\alpha}}.M_\alpha]\!] : \delta \mapsto E \Leftarrow A[\Psi_\alpha] \to T}$$

$$\frac{\text{for all } k\ \Omega;\Delta;\Gamma \vdash E_k \Leftarrow T_k}{\Omega;\Delta;\Gamma \vdash (E_1, E_2, \ldots, E_n) \Leftarrow T_1 * T_2 * \cdots * T_n} \qquad \frac{\Omega;\Delta;\Gamma \vdash E_k \Leftarrow T_k}{\Omega;\Delta;\Gamma \vdash (E_1, E_2, \ldots, E_n).k \Leftarrow T_k}$$

$$\frac{\Omega;\Delta;\Gamma \vdash I \Rightarrow T' \quad T = T'}{\Omega;\Delta;\Gamma \vdash I \Leftarrow T}$$

Figure 4.17: Typing rules for the simply typed computational language.

The typing rules follow closely those presented in [46] only simplified by the lack of dependent types after the dependency erasure phase, and the annotations that need to be added so that terms in the Contextual LF level keep track of their worlds. Most of the rules are not particularly surprising or interesting, however the rules that type-check substitutions illustrate how contexts and worlds interact, this will be important when thinking about the composition of substitutions and the weak head normal functions in Section 4.6.3.

One important aspect of the typing rules is that they validate two error-prone areas of the compiler: the substitutions, and binders escaping their scope.

Accordingly, in Section 4.3 we introduced the fresh-style representation implemented in OCaml. We also said that the lack of dependent types on our implementation meant that the powerful invariants tracked by the representation proposed in [53] and described in 2.4.2 cannot be guaranteed in our implementation. However, the world-annotated typing rules (Figures 4.16 and 4.17) will fail to type-check if we forget to do an import or if we try to use a name outside of its scope, which is powerful even if it is not as convenient as having it validated by Agda's type-checker as in [53]. The advantage of this representation with regards to de Bruijn indices, is that bugs that originate in a missing shift (usually very difficult to trace and discover) will manifest themselves as typing errors.

4.5 Pattern Matching Compilation

We have seen Beluga's pattern matching in the programming examples Section 3.2 and the examples 3.2.2 and 3.2.3 show the power of patterns in Beluga. This pattern matching can be seen as the regular ML-style pattern matching with support for contextual objects. The addition of contextual objects augments the well-known matching against variables and constructors common in many functional programming languages with matching against the shape of the context, inside binders, and against specific vari-

ables in the context. We will begin our discussion with the general techniques we use that we adapted from regular pattern matching. In particular, we will follow the treatment presented in [36] and [31] and then we will discuss how to extend this technique to the more powerful patterns of Beluga.

In Beluga, pattern matching is performed by the `case` expression of the computational language(Figure 4.15), and it gives us an expression that needs to be evaluated and matched against a list of branches which contain patterns and the branch bodies. The naive way to compile this is to try to match the input one pattern at a time until a match is found. This could be inefficient, and better options have been available for a long time. Many presentations of pattern compilation revolve around the idea of compiling the patterns into a discrimination tree that can be executed efficiently but depending on the expression and the exact technique it may produce an exponential blowup of the size of program. One of the first discussions of this approach appears in the description of the compiler for the HOPE programming language from 1980 presented in [10], it was further discussed in the case of a compiler for ML in 1984 in [11] and [5].

The same way pattern matching allows a clear way of thinking about data structures when writing your program, it provides lots of information to the compiler which then gives the chance to the optimizer to generate good code. Already in 1984, given the state of the art at the time, we can read in [11]: "...a compiler can produce better code from a pattern matching description than from a corresponding sequence of discrimination and selection operations."

The algorithm for computing this discrimination tree is far from trivial, and different choices result in different performance and memory consumption. For our compiler we derive our technique from the more recent presentations in [31] where the authors propose to use a form of backtracking automata to keep the size of the generated code as small as possible sacrificing some performance, and [36] where a technique using trees

for high-performance and some heuristics to try to minimize the size of the tree are used.

4.5.1 A Hand-Coded Example

Before going too deep in how the dependency trees are generated we will discuss a small hand generated example(extracted from [36]).

Suppose we have the following OCaml function:

```
let f x y z = match x,y,z with
| _, F, T → 1
| F, T, _ → 2
| _, _, F → 3
| _, _, T → 4
```

It is a function from booleans (here represented by T and F) to integers, compilation goes by recursively splitting all the cases for each pattern, in this case each of the three boolean variables x, y, z.

So if we start splitting x we obtain an expression like:

```
let f1 x y z =
  if x then
    if y then
      if z then 4 else 3
    else
      if z then 1 else 3
  else
    if y then 2
    else
      if z then 1 else 3
```

It is important to see that the functions f and f1 both provide the same answers when called with the same parameters, e.g.: both f F T F and f1 F T F evaluate to 2. In this example we got a discrimination tree that expresses the high-level notion of pattern matching just using if expressions which are lower level and thus, generating code for them is easier.

Notice how in the previous example we split first by x, but it is important to see that there is a choice in this case, we could as easily have chosen y or z. In the following code snippet we show the function would look if we choose to split first on y.

```
let f2 x y z =
  if y then
    if x then
      if z then 4 else 3
    else 2
  else
    if z then 1 else 3
```

For each choice, in general, we get a different decision tree and it is not hard to see, that f2 is more efficient because for a false y, f2 performs one test less than f1, and for all the other cases they perform the same number of tests.

In conclusion, the point of this artificial example is to show what it means to generate these discrimination trees and also to illustrate that the choices the compiler makes change the size and performance of the resulting code.

4.5.2 An Algorithm for Patterns with Variables and Constructors

In this section we will present the algorithm described in [36] which will not support the full power of Beluga patterns, but it will be simpler to describe and from this we will add the support for the missing features.

For this section we will work with a simple language for patterns:

$$\text{Pattern} \;\; p ::= _ \mid c(p_1, \ldots, p_n)$$

These patterns support only variables, that we represent here with _ and we suppose all different as their names are not relevant. Of course, in the complete implementation names are important because they are bound in the bodies of each branch, but for the

purpose of this discussion the names are not important.

From the case expression we form a matrix P where the rows contains all the patterns of each branch (there can be more than one when we do simultaneous pattern matching, or after splitting the patterns during compilation) and one row for each branch, and a vector A of all the bodies of the branches, with P and A we build $P \to A$ exactly as shown in 4.18.

$$P \to A = \begin{pmatrix} p_{1,1} & \cdots & p_{1,n} & \to & a_1 \\ p_{2,1} & \cdots & p_{2,n} & \to & a_2 \\ \vdots & \ddots & \vdots & & \vdots \\ p_{m,1} & \cdots & p_{m,n} & \to & a_m \end{pmatrix}$$

Figure 4.18: Matrix of clauses.

If we were to calculate $P \to A$ for the small example from Section 4.5.1, the result would be:

$$P \to A = \begin{pmatrix} _ & F & T & \to & 1 \\ F & T & _ & \to & 2 \\ _ & _ & F & \to & 3 \\ _ & _ & T & \to & 4 \end{pmatrix}$$

In addition to the $P \to A$ matrices we define two decomposition operations on them, the first one *specializes* the matrix with respect to one constructor written $S(c, P \to A)$, and the second one produces a *default* matrix, written $D(P)$. In a nutshell, specialization selects all those branches that match a certain constructor and default all the branches that would not match any of the branches. So we build the tree by branching on each of the specialized matrices and the default matrix for the rest of the cases.

Again, to illustrate the specialization operation, we will specialize the matrix from

$$\begin{array}{rcl} \text{Pattern } P_{i,1} & & \text{Row of } S(c, P \to A) \\ c(p'_1, \ldots, p'_a) & \rightsquigarrow & \underbrace{p'_1 \ldots p'_a}_{\text{from } c} \; p_{i,2} \ldots p_{i,n} \to a_i \\ c'(p'_1, \ldots, p'_a)(c' \neq c) & \rightsquigarrow & \text{no row} \\ _ & \rightsquigarrow & \overbrace{_ \cdots _}^{(\times a)} \; p_{i,2} \ldots p_{i,n} \to a_i \end{array}$$

Figure 4.19: $S(c, P \to A)$.

the example in Section 4.5.1:

$$S(T, P \to A) = \begin{pmatrix} F & T & \to & 1 \\ _ & F & \to & 3 \\ _ & T & \to & 4 \end{pmatrix} \quad S(F, P \to A) = \begin{pmatrix} F & T & \to & 1 \\ T & _ & \to & 2 \\ _ & F & \to & 3 \\ _ & T & \to & 4 \end{pmatrix}$$

We define both operations as presented in [36]. The result of $S(c, P \to A)$ is a new matrix with only the rows whose first pattern ($P_{i,1}$) admits the constructor c. Furthermore, it splits the first column according to the arity of constructor c. On the other hand, the default operation $D(P)$ retains all the rows of P whose first pattern admits all values $c'(v_1, \ldots, v_a)$, where c' is not in the first column of P. Figures 4.19 and 4.20 show the definition of both operations.

$$\begin{array}{rcl} \text{Pattern } P_{i,1} & & \text{Row of } D(P) \\ c(p'_1, \ldots, p'_a) & \rightsquigarrow & \text{no row} \\ _ & \rightsquigarrow & p_{i,2} \ldots p_{i,n} \quad \to \quad a_i \end{array}$$

Figure 4.20: $D(P \to A)$.

To illustrate the default operation, we use the same example as before, and we show

that the result would be:

$$D(P \to A) = \begin{pmatrix} F & T & \to & 1 \\ _ & F & \to & 3 \\ _ & T & \to & 4 \end{pmatrix}$$

4.5.2.1 The Compilation Scheme

In this section we define the compilation scheme $CC(\vec{o}, P \to A)$ where $\vec{o}$ are the expressions we pattern match on, and of course, $P \to A$ is the clause matrix.

1. If the matrix P has no rows pattern matching fails.

$$CC(\vec{o}, \emptyset \to A) \stackrel{def}{=} \mathsf{fail}$$

2. If the first row of P consists only of variables the matching yields the first action.

$$CC(\vec{o}, \begin{pmatrix} _ & \cdots & _ & \to & a_1 \\ p_{2,1} & \cdots & p_{2,n} & \to & a_2 \\ \vdots & \ddots & \vdots & & \vdots \\ p_{m,1} & \cdots & p_{m,n} & \to & a_m \end{pmatrix}) \stackrel{def}{=} \mathsf{leaf}\ a_1$$

3. Otherwise, P is not empty and there is at least one column i of the first row which is not a variable. So we swap the first column and the i column in P and in $\vec{o}$, so now column 1 contains a constructor.
   (a) We calculate Σ_1 as the set of constructors of the first column of P, it cannot be empty by hypothesis (we swapped the columns exactly for this).
   (b) We calculate:

$$A_k \stackrel{def}{=} CC((o_1.1 \dots o_1.a\ o_2 \dots o_n), S(c_k, P \to A))$$

for all $c_k \in \Sigma_1$ and we calculate a default case:

$$A_d \stackrel{def}{=} CC((o_2 \dots o_n), D(P \to A))$$

(c) We create a switch expression in the decision tree by associating to each c_i its corresponding action A_i and the default case with the action A_d

And this is the basic compilation algorithm for simple patterns, there is one important remark that we need to make: as we mentioned before the algorithm offers choice in step 3 because when we choose the column i there may be more than one possible option. Currently the Beluga compiler uses the trivial heuristic of choosing the first suitable column, this leaves space for optimizations but as we will see, the powerful pattern matching of Beluga makes finding a good heuristic a bit challenging.

To continue with the example from Section 4.5.1, we would call $CC([x,y,z], P \to A)$ to compile the patterns in the example. Of course as the first column contains one constructor, rule 3 would apply and we would split with the two constructors, calling specialize with T and F as we did in the example before. For the first branch we would recursively call the compilation with $CC([y,z], S(T, P \to A)$, for the second branch $CC([y,z], S(T, P \to A)$ and we would have a third branch with $CC([y,z], D(P \to A)$ in case x did not match any of the two previous branches, the *default* branch.

4.5.3 Compiling Beluga's Patterns

Section 4.5.2 on page 51 discusses how to compile simple patterns composed of variables and type constructors, but practical languages have more complicated patterns. In particular, Beluga has powerful and expressive patterns. As example let's consider the normalization function (Figure 3.10) and the hoas2db function (Figure 3.8).

Where before we had only one rule for constructors and variables, in Beluga we have

the following six rules:

① The shape of contexts

② Constructors

③ Variables from the value context

④ Parameter variable

⑤ Meta-variable with a substitution

⑥ Lambda patterns

```
      rec norm : {g:ctx} (exp T) [g] -> (exp T) [g]
      fn e => case e of
④     | [g] #p.. => [g] #p..

②⑥⑤   | [g] lam (\x. M..x) =>
        let [g, x:exp _] N..x = norm ([g,x:exp _] M..x) in
        [g] lam \x. N..x

②⑤    | [g] app (M1..) (M2..) =>
        (case norm ([g] M1..) of
         | [g] lam (\x. M'..x) => norm ([g] M'.. (M2..)
         | [g] N1.. =>
           let [g] N2.. = norm ([g] M2..) in
           [g] app (N1..) (N2..))
      ;
```

```
      rec hoas2db : {g:ctx} exp [g] -> exp' [ ] =
      fn e => case e of
①③    | [g, x:exp] x  => [ ] one
④     | [g, x:exp] #p.. =>
        let [ ] M' = hoas2db ([g] #p..) in
        [ ] shift M'
②⑤    | [g] app (M..) (N..) =>
        let [ ] M' = hoas2db ([g] M..) in
        let [ ] N' = hoas2db ([g] N..) in
        [ ] app' M' N'
②⑥⑤   | [g] lam (\x. M..x) =>
        let [ ] M' = hoas2db ([g, x:exp] M..x) in
        [ ] lam' M'
      | [g] num X => [ ] num' X
      ;
```

Figure 4.21: Kinds of patterns.

Figure 4.21 illustrates with some examples all the different kinds of pattern matching supported by the system. Some lines are annotated with the rule number they showcase best.

4.5.3.1 The Target Language for Pattern Matching Compilation

In the algorithm from Section 4.5.2.1 step 3(c) we create a switch statement that branches the tree for a set of constructors of the same type. In Beluga we need more than one switch expression that splits on different constructors. In Figure 4.22 we see the extension to the computational language required to support pattern matching compilation.

The compilation of the rich pattern language for a language that supports HOAS, is as we said in the introduction, one of the main contributions of this work. This is important to make programming languages with HOAS more practical, but it also leads to interesting questions related to an extension of the splitting tree to dependent types, and the coverage checking algorithms for dependently typed languages. This is because coverage checking generally proceeds by splitting on patterns([48], [56] and [17]) with the additional difficulty that in a dependently typed setting the splitting is, in general, undecidable.

In 4.5.2 pattern compilation goes from case expressions to switch expressions where we only inspect the first level constructors. In Figure 4.22 we add two new expressions ctx_switch and switch and the simple patterns that prevent nesting.

$$
\begin{array}{lcll}
\text{Expr. Checked} & E & ::= & \ldots \mid \mathsf{switch}\ I\ \mathsf{of}\ \vec{S} \mid \mathsf{ctx_switch}\ I\ \mathsf{of}\ \vec{C} \\
\text{Ctx. Switch Branches} & \vec{C} & ::= & \cdot \mid (C|\vec{C}) \\
\text{Ctx. Switch Branch} & C & ::= & \Pi\Omega'.[\![\hat{\Psi}_{\hat{\alpha}}]\!] : \delta \mapsto E \\
\text{Switch Branches} & \vec{S} & ::= & \cdot \mid (S|\vec{S}) \\
\text{Switch Branch} & S & ::= & \Pi\Delta'.[\![\hat{\Psi}_{\alpha}.Q_{\alpha}]\!] \mapsto E \\
\text{Simple Pattern} & Q_{\alpha} & ::= & \lambda\alpha \leftharpoondown \beta, V_{\beta} \mid \mathrm{con}\ \vec{V}_{\alpha} \mid \mathsf{n}_{\alpha} \mid V_{\alpha} \\
\text{Pattern Variable} & V_{\alpha} & ::= & u[\overset{\gamma\to\alpha}{\sigma}] \mid p[\overset{\gamma\to\alpha}{\sigma}] \\
\text{Pattern Variable List} & \vec{V}_{\alpha} & ::= & \cdot \mid (V_{\alpha}|\vec{V}_{\alpha})
\end{array}
$$

Figure 4.22: Pattern compilation target language.

The pattern compilation language augments the computational language with two new expressions ctx_switch and switch. The former splits the tree by matching on the shape of the context and thus its patterns are contexts $\hat{\Psi}_{\hat{\alpha}}$. These contexts used as patterns, will often contain context variables ($\psi_{[\cdot]}$) and the contextual substitution δ is there to replace the context variable for the right variable. On the other hand, the latter expression splits on all the other patterns. To keep switch low-level it does not support nested patterns, that is why we defined the simple patterns. Simple patterns support all

the required patterns without allowing nesting.

4.5.3.2 Typing Rules for the Expanded Language

Pattern compilation requires an extension (switch and ctx_switch) to the computational language, it would be interesting to be able to run the type-checker we developed for the language before the extension, so in this section we present the new typing rules for the computational language. Figures 4.23 and 4.24 show the typing rules. These rules are derived from the rule for the case expression.

Context Switch:

$$\frac{\Omega;\Delta;\Gamma \vdash I \Rightarrow A[\Psi] \quad \text{for all } k\ \Omega;\Delta;\Gamma \vdash C_k \Leftarrow T}{\Omega;\Delta;\Gamma \vdash \mathsf{ctx_switch}\ I\ \mathsf{of}\ C_1|\ldots|C_n \Leftarrow T}$$

Branches:

$$\frac{[\delta]\Omega';\Delta';[\delta]\Gamma \vdash E \Leftarrow [\delta]T}{\Omega;\Delta;\Gamma \vdash \Pi\Omega'.[\![\hat{\Psi}_{\hat{\alpha}}]\!] : \delta \mapsto E \Leftarrow T}$$

General Switch:

$$\frac{\Omega;\Delta;\Gamma \vdash I \Rightarrow A[\Psi_\alpha] \quad \text{for all } k\ \Omega;\Delta;\Gamma \vdash S_k \Leftarrow A[\Psi_\alpha] \rightarrow T}{\Omega;\Delta;\Gamma \vdash \mathsf{switch}\ I\ \mathsf{of}\ S_1|\ldots|S_n \Leftarrow T}$$

Branches:

$$\frac{\Omega;\Delta';[\delta]\Psi_\alpha \vdash Q_\alpha \Leftarrow A \quad \Omega;\Delta';[\delta]\Gamma \vdash E \Leftarrow [\delta]T \quad \Delta' \vdash \theta \Leftarrow \Delta}{\Omega;\Delta;\Gamma \vdash \Pi\Delta'.[\![\hat{\Psi}_\alpha.Q_\alpha]\!] : \theta \mapsto E \Leftarrow A[\Psi_\alpha] \rightarrow T}$$

Figure 4.23: Typing rules for switch expressions.

The rules for simple patterns are similar to those of their more complex siblings (LF terms).

$$\frac{\Delta;\Psi_\alpha \vdash \Sigma(\mathsf{con}) \Rightarrow A \quad \Delta;\Psi_\alpha \vdash \vec{V_\alpha} : A \Leftarrow P}{\Delta;\Psi_\alpha \vdash \mathsf{con}\ \vec{V_\alpha} \Leftarrow P} \qquad \frac{\Delta;\Psi_\alpha(\mathsf{n}_\alpha) = A}{\Delta;\Psi_\alpha \vdash \mathsf{n}_\alpha \Leftarrow A}$$

$$\frac{\Delta(u) = A[\Phi_\gamma] \quad \Delta,\Psi_\alpha \vdash \overset{\gamma\to\alpha}{\sigma} \Leftarrow \Phi_\gamma}{\Delta;\Psi_\alpha \vdash u[\overset{\gamma\to\alpha}{\sigma}] \Rightarrow A} \qquad \frac{\Delta(p) = A[\Phi_\gamma] \quad \Delta,\Psi_\alpha \vdash \overset{\gamma\to\alpha}{\sigma} \Leftarrow \Phi_\gamma}{\Delta;\Psi_\alpha \vdash p[\overset{\gamma\to\alpha}{\sigma}] \Rightarrow A}$$

$$\frac{\Delta(V_\beta) = B[\Psi'_\beta] \quad \Delta;\Psi_\alpha, \alpha \leftharpoondown \beta : A \vdash id \Leftarrow \Psi'_\beta}{\Delta;\Psi_\alpha \vdash \lambda \alpha \leftharpoondown \beta . V_\beta \Leftarrow A \to B}$$

Figure 4.24: Typing rules for simple patterns.

4.5.3.3 With Beluga's Patterns

In Beluga each branch of a case statement starts with a context they match on. It is possible to specify empty contexts, arbitrary contexts, contexts with only one variable, contexts with at least one variable, etc. One particularity of context patterns is that they are always the first pattern, and that they are not nested, so when compiling, the discrimination tree begins by discriminating on the contexts. The contextual discrimination is done by adding a ctx_switch expression that groups the branches with similar patterns. Because there is no possibility of nesting this is simple.

Of the six rules for patterns, ctx_switch accounts for the first, the other five consist in modifications to the specialization operation defined in Section 4.5.2.1.

When presenting the pattern matching algorithm for variables and constructors (see Section 4.5.2 on page 51) we show the definition of the specialization operation for a simplistic pattern language (Figure 4.19). So, in Beluga's full pattern language the main change is that besides constructors, patterns can be any of the elements from rules 2 to 6 in 4.5.3. The purpose of the specialization operation is to group together all the branches which match the same pattern (we only had constructors in the previous example), so all we need to do is to extend the operation with new rules to support the new patterns.

The new specialization operation will be extended in the following ways:

– Constructors: when specializing we group together all the constructors of the same name, exactly as we did before.

– Variables from the value context: they will be the same when they refer to the same element of the context, independent of name, they will match when they refer to analogous elements in the context (called *pronominal* variables in [35]).

– Parameter variable and Meta-variables: in Section 4.5.2.1, variables are all grouped together, but parameter and meta-variables have an associated substitution, to adapt the contexts when performing substitutions, so when specializing for this variables, we only combine variables with the same substitution, otherwise we would be mixing incompatible contexts.

– Lambda patterns: all lambdas in the head of a pattern are equivalent, and the name of the bound variable is irrelevant because we compare names pro-nominally [35].

The current implementation of the compiler, the extension is simply built using an ML functor with a parameter to specify a rule. There is an implementation for each one of the rules. As mentioned before, pattern matching compilation under these conditions provides many opportunities to the heuristic of choosing the column, but this has not been thoroughly explored, and the code simply chooses the first column that does not consist only of meta-variables with the same substitution.

In conclusion, the pattern matching compilation contains three elements, first the language extension containing the simple patterns and switch expressions, the compilation algorithm with the parameterized specialization and finally the new type-checking rules to be able to check the result of the compilation with the extended language.

4.6 Operational Semantics and Runtime

4.6.1 Operational Semantics

In this section we discuss the operational semantics of Beluga using the Fresh-Style representation, after compiling the pattern matching. In Section 4.6.2 we will discuss the transformation back to names or de Bruijn indices.

Values	v	$::=$	$x \mid \text{fn}\ x.E \mid \lambda^{\square}\ X.E \mid \Lambda\ \psi.E \mid [\![\hat{\Psi}_{\hat{\alpha}}.M_{\alpha}]\!]$
Extended Values	w	$::=$	$v \mid (\mathsf{rec}\ f.E)[\theta;\delta;\rho]$
Environments	ρ	$::=$	$\cdot \mid \rho, W/x$

For the semantics we use a similar set-up as the presentation found in [12] and [46]. It will be an environment based approach in which we evaluate an expression E in a suspended substitution containing the meta-variable substitution(θ), the context substitution(δ), and the computational substitution (ρ). So the evaluation of $E[\theta;\delta;\rho]$ is done using the rules defined in Figure 4.25.

The rules for applications, functional values (λ, $\lambda^{\square}$, and Λ), and recursion are equivalent to the rules presented in Section 4.2 of [46]. What is different are the rules for the ctx_switch and switch expressions (the last four rules in Figure 4.25). The first two of these show the semantics for ctx_switch expressions and the latter two for switch expressions.

As described in Section 4.5.3, contextual discrimination is the first branching of the tree and is done using ctx_switch, the semantics show that the $\overset{\mathsf{ctx}}{\approx}$ operation is used to find the first branch whose context matches the pattern and the expression evaluates to the result of evaluating the body of the branch in the right context.

Figure 4.26 shows the inductive definition of $\overset{\mathsf{ctx}}{\approx}$. It tests that both contexts are the same length with two particularities: any two context variables are considered equal

$$\frac{I[\theta;\delta;\rho] \Downarrow (\mathsf{fn}\ x.E')[\theta';\delta';\rho'] \quad E[\theta;\delta;\rho] \Downarrow v_a \quad E'[\theta';\delta';\rho',v_a/x] \Downarrow v}{(IE)[\theta;\delta;\rho] \Downarrow v}$$

$$\frac{I[\theta;\delta;\rho] \Downarrow \lambda^{\Box}X.E'[\theta';\delta';\rho'] \quad E'[\theta',M_\alpha/X;\delta';\rho'] \Downarrow v}{(I[\![\Psi_\alpha.M_\alpha]\!])[\theta;\delta;\rho] \Downarrow v}$$

$$\frac{I[\theta;\delta;\rho] \Downarrow \Lambda\psi_{[\cdot]}.E'[\theta';\delta';\rho'] \quad E'[\theta';\delta',\Psi_\alpha/\psi_{[\cdot]};\rho'}{(I\lfloor\Psi_\alpha\rfloor)[\theta;\delta;\rho] \Downarrow v}$$

$$\frac{\rho(x) = v}{x[\theta;\delta;\rho] \Downarrow v} \qquad \frac{E[\theta;\delta;\rho] \Downarrow v}{E:T[\theta;\delta;\rho] \Downarrow v}$$

$$\frac{\rho(x) = (\mathsf{rec}\ f.E')[\theta';\delta';\rho'] \quad \rho(x) \Downarrow v}{x[\theta;\delta;\rho] \Downarrow v} \qquad \frac{}{(\mathsf{fn}\ x.E)[\theta;\delta;\rho] \Downarrow (\mathsf{fn}\ x.E)[\theta;\delta;\rho]}$$

$$\frac{}{(\lambda^{\Box}\ X.E)[\theta;\delta;\rho] \Downarrow (\lambda^{\Box}\ X.E)[\theta;\delta;\rho]} \qquad \frac{}{(\Lambda\psi_{[\cdot]}.E)[\theta;\delta;\rho] \Downarrow (\Lambda\psi_{[\cdot]}.E)[\theta;\delta;\rho]}$$

$$\frac{E[\theta;\delta;\rho,\mathsf{rec}\ f.E[\theta;\delta;\rho]/f] \Downarrow v}{\mathsf{rec}\ f.E[\theta;\delta;\rho] \Downarrow v} \qquad \frac{E_1[\theta;\delta;\rho] \Downarrow v_1 \quad \ldots \quad E_n[\theta;\delta;\rho] \Downarrow v_n}{(E_1,\ldots,E_n)[\theta;\delta;\rho] \Downarrow (v_1,\ldots,v_n)}$$

$$\frac{I[\theta;\delta;\rho] \Downarrow [\![\Phi_\gamma.N_\gamma]\!] \quad \Phi_\gamma \stackrel{ctx}{\not\approx} \hat{\Psi}_{\hat{\alpha}} \quad (\mathsf{ctx_switch}\ I\ \mathsf{of}\ \vec{C})[\theta;\delta;\rho] \Downarrow v}{(\mathsf{ctx_switch}\ I\ \mathsf{of}\ \Pi\Omega'.[\![\hat{\Psi}_{\hat{\alpha}}]\!] : \delta' \mapsto E|\vec{C})[\theta;\delta;\rho] \Downarrow v}$$

$$\frac{I[\theta;\delta;\rho] \Downarrow [\![\Phi_\gamma.N_\gamma]\!] \quad \Phi_\gamma \stackrel{ctx}{\approx} \hat{\Psi}_{\hat{\alpha}} \quad E[\theta;[\delta']\delta;\rho] \Downarrow v}{(\mathsf{ctx_switch}\ I\ \mathsf{of}\ \Pi\Omega'.[\![\hat{\Psi}_{\hat{\alpha}}]\!] : \delta' \mapsto E|\vec{C})[\theta;\delta;\rho] \Downarrow v}$$

$$\frac{I[\theta;\delta;\rho] \Downarrow [\![\Phi_\gamma.N_\gamma]\!] \quad \hat{\Psi}_\alpha.Q_\alpha \stackrel{M}{\not\approx} \Phi_\gamma.\mathsf{whnf}(N_\gamma) \quad (\mathsf{switch}\ I\ \mathsf{of}\ \vec{S})[\theta;\delta;\rho] \Downarrow v}{(\mathsf{switch}\ I\ \mathsf{of}\ \Pi\Delta'.[\![\hat{\Psi}_\alpha.Q_\alpha]\!] : \theta' \mapsto E|\vec{S})[\theta;\delta;\rho] \Downarrow v}$$

$$\frac{I[\theta;\delta;\rho] \Downarrow [\![\Phi_\gamma.N_\gamma]\!] \quad \hat{\Psi}_\alpha.Q_\alpha \stackrel{M}{\approx} \Phi_\gamma.\mathsf{whnf}(N_\gamma) \quad E[[\theta']\theta;\delta;\rho] \Downarrow v}{(\mathsf{switch}\ I\ \mathsf{of}\ \Pi\Delta'.[\![\hat{\Psi}_\alpha.Q_\alpha]\!] : \theta' \mapsto E|\vec{S})[\theta;\delta;\rho] \Downarrow v}$$

Figure 4.25: Big-step operational semantics.

as one will be replaced by the other when applying the δ' contextual substitution, and finally context variables match any context.

The switch expression is more complex, and it depends on the $\stackrel{M}{\approx}$ match operation. This operation compares simple patterns (P_α) as defined in Figure 4.22 to the head of the contextual LF terms. In a switch we only need to much against heads of terms because of simple patterns that do not allow nesting. In this setting where we have explicit substitutions, we need to partially apply the substitutions to be able to inspect the head of the term. Therefore before using the matching operation we calculate the weak head normal form of the term.

The matching operation $\stackrel{M}{\approx}$ is defined in Figure 4.26.

A pattern and a term match when:

- The head of the term and the pattern are the same constructor.
- The head and the pattern are λ-expressions.
- The pattern is a meta-variable and inverse substitution operation is defined on the term. That is, the substitution maps all the free variables in the term to variables in the right context.
- The pattern is a parameter variable, the inverse substitutions is defined as in the meta-variable case, and the head of the term is a name (an LF variable).
- The pattern and the head of the term are LF variables declared in the same position in their contexts.

Previously, we defined a small untyped λ-calculus with explicit substitutions (Section 4.4.1) and then we showed how to compute the weak head normal form of a term (Figure 4.8), that is how we calculate the head of a term without completely applying the explicit substitutions, but just enough to establish the head of the term. In our representation of Contextual LF we need to perform the same operation to be able to use

Context Matching:

$$\frac{}{\cdot \overset{ctx}{\approx} \cdot} \quad \frac{}{\psi_{[\cdot]} \overset{ctx}{\approx} \phi_{[\cdot]}} \quad \frac{}{_ \overset{ctx}{\approx} \psi_{[\cdot]}} \quad \frac{\Psi_\alpha \overset{ctx}{\approx} \Phi_\alpha}{\Psi_\alpha, \alpha \leftharpoonup \beta : A \overset{ctx}{\approx} \Phi_\alpha, \alpha \leftharpoonup \beta : A}$$

Term Matching:

$$\frac{\mathsf{con} = \mathsf{const}}{\hat{\Psi}_\alpha.\mathsf{con}\ \vec{V}_\alpha \overset{M}{\approx} \Phi_\gamma.\mathsf{const} \cdot S_\gamma} \quad \frac{}{\hat{\Psi}_\alpha.\lambda\alpha \leftharpoonup \beta.V_\beta \overset{M}{\approx} \Phi_\gamma.\lambda\gamma \leftharpoonup \delta.M_\delta}$$

$$\frac{[\overset{\alpha\to\gamma}{\sigma}]^{-1} M_\gamma}{\hat{\Psi}_\alpha.u[\overset{\alpha\to\gamma}{\sigma}] \overset{M}{\approx} \Phi_\gamma.M_\gamma} \quad \frac{[\overset{\alpha\to\gamma}{\sigma}]^{-1} \mathsf{n}_\gamma}{\hat{\Psi}_\alpha.p[\overset{\alpha\to\gamma}{\sigma}] \overset{M}{\approx} \Phi_\gamma.\mathsf{n}_\gamma}$$

$$\frac{\mathsf{name_of}\ \alpha \leftharpoonup \beta = \mathsf{n}_\alpha \quad \mathsf{name_of}\ \delta \leftharpoonup \gamma = \mathsf{n}'_\gamma}{\hat{\Psi}'_\beta, \beta \leftharpoonup \alpha.\mathsf{n}_\alpha \overset{M}{\approx} \Phi'_\delta, \delta \leftharpoonup \gamma.\mathsf{n}'_\gamma}$$

$$\frac{\begin{array}{l}\mathsf{name_of}\ \delta \leftharpoonup \gamma \neq \mathsf{n}'_\gamma \\ \mathsf{name_of}\ \alpha \leftharpoonup \beta \neq \mathsf{n}_\alpha \quad \hat{\Psi}'_\beta.\mathsf{n}_\beta \overset{M}{\approx} \Phi'_\delta.\mathsf{n}'_\delta\end{array}}{\hat{\Psi}'_\beta, \beta \leftharpoonup \alpha.\mathsf{n}_\alpha \overset{M}{\approx} \Phi'_\delta, \delta \leftharpoonup \gamma.\mathsf{n}'_\gamma}$$

Figure 4.26: Matching operations.

$$
\begin{array}{llcl}
\mathsf{whnf} & \underbrace{(\overbrace{\lambda\alpha \leftharpoonup \beta.M_\beta}^{\alpha}, \overset{\alpha\to\gamma}{\sigma})}_{\gamma} & = & (\lambda\alpha \leftharpoonup \beta.M_\beta, \overset{\alpha\to\gamma}{\sigma}) \\
\mathsf{whnf} & (\mathsf{clo}(M_\alpha, \overset{\alpha\to\delta}{\sigma_1}), \overset{\delta\to\beta}{\sigma_2}) & = & \mathsf{whnf}(M_\alpha, \overset{\alpha\to\delta}{\sigma_1} \circ \overset{\delta\to\beta}{\sigma_2}) \\
\mathsf{whnf} & (u[\overset{\gamma\to\delta}{\sigma_1}] \cdot \mathsf{nil}, \overset{\delta\to\alpha}{\sigma_2}) & = & (u[\overset{\gamma\to\delta}{\sigma_1} \circ \overset{\delta\to\alpha}{\sigma_2}] \cdot \mathsf{nil}, \mathsf{id}) \\
\mathsf{whnf} & (c \cdot S_\alpha, \overset{\alpha\to\beta}{\sigma}) & = & (c \cdot \mathsf{clo}(S_\alpha, \overset{\alpha\to\beta}{\sigma}), \mathsf{id}) \\
\mathsf{whnf} & (\mathsf{n}_\alpha \cdot S_\alpha, \overset{\alpha\to\beta}{\sigma}) & = & \begin{cases} \mathsf{whnfRedex}\ \mathsf{clo}(M_\beta, \mathsf{id})\ \mathsf{clo}(S_\alpha, \overset{\alpha\to\beta}{\sigma}) & \text{if } [\overset{\alpha\to\beta}{\sigma}]\mathsf{n}_\alpha = M_\beta \\ (H_\beta \cdot \mathsf{clo}(S_\alpha, \overset{\alpha\to\beta}{\sigma}), \mathsf{id}) & \text{if } [\overset{\alpha\to\beta}{\sigma}]\mathsf{n}_\alpha = H_\beta \end{cases}
\end{array}
$$

$$
\begin{array}{lllcl}
\mathsf{whnfRedex} & (H_\gamma \cdot S_\gamma, \overset{\gamma\to\alpha}{\sigma_1}) & (\mathsf{nil}, \overset{\delta\to\alpha}{\sigma_2}) & = & \mathsf{whnf}(H_\gamma \cdot S_\gamma, \overset{\gamma\to\alpha}{\sigma_1}) \\
\mathsf{whnfRedex} & (\lambda\gamma \leftharpoonup \delta.M_\delta, \overset{\gamma\to\alpha}{\sigma_1}) & (N_\tau\ S_\tau, \overset{\tau\to\alpha}{\sigma_2}) & = & \mathsf{whnfRedex}(M_\delta, (\overset{\gamma\to\alpha}{\sigma_1}, \mathsf{clo}(N_\tau, \overset{\tau\to\alpha}{\sigma_2}))(S_\tau, \overset{\tau\to\alpha}{\sigma_2}) \\
\mathsf{whnfRedex} & (M_\gamma, \overset{\gamma\to\alpha}{\sigma_1}) & (\mathsf{clo}(S_\tau, \overset{\tau\to\delta}{\sigma_3}), \overset{\delta\to\alpha}{\sigma_2}) & = & \mathsf{whnfRedex}(M_\gamma, \overset{\gamma\to\alpha}{\sigma_1})(S_\tau, \overset{\tau\to\delta}{\sigma_3} \circ \overset{\delta\to\alpha}{\sigma_2}) \\
\mathsf{whnfRedex} & (\mathsf{clo}(M_\tau, \overset{\tau\to\gamma}{\sigma_3}), \overset{\gamma\to\alpha}{\sigma_1}) & (S_\delta, \overset{\delta\to\alpha}{\sigma_2}) & = & \mathsf{whnfRedex}(M_\tau, \overset{\tau\to\gamma}{\sigma_3} \circ \overset{\gamma\to\alpha}{\sigma_1})(S_\delta, \overset{\delta\to\alpha}{\sigma_2})
\end{array}
$$

Figure 4.27: Weak head normal form.

the match operator. In this setting the operation is slightly more complicated by the fact that our representation (Figure 4.11) forbids non-normal terms. The purpose of having separate syntactical categories for *normal terms* and for *head* and *spine* terms prevents us from having a term consisting of a λ-expression applied to another term. As a consequence if after performing a substitution the weak head normal form of a term is not representable, we need to continue performing the substitution until we reach a representable term. This type of substitution operation is called hereditary substitution and it is the justification for function whnfRedex in Figure 4.27. For an in-depth discussion of hereditary substitutions please refer to [40].

Also, to avoid having to eagerly apply substitutions to terms and spines, we well add to each category a closure, that combines a term or a spine with a substitution. This way we can be lazy about applying substitutions. The extensions to the contextual LF

language are as follows:

$$\begin{array}{llcl} \text{Normal Term} & M_\alpha, N_\alpha & ::= & \ldots \mid \mathsf{clo}(M_\gamma, \overset{\gamma\to\alpha}{\sigma}) \\ \text{Spine} & S_\alpha & ::= & \ldots \mid \mathsf{clo}(S_\gamma, \overset{\gamma\to\alpha}{\sigma}) \end{array}$$

The matching operation for meta-variables and parameter-variables requires applying substitutions in the reverse order. We could calculate the inverse of the substitution, and then apply it but it is simpler to define an operation that applies the inverse of a substitution([21] and [3]). For a substitution $\overset{\alpha\to\beta}{\sigma}$ and a term $t ::= M_\beta \mid H_\beta \mid \overset{\gamma\to\beta}{\sigma_x}$ we define the partial function $[\overset{\alpha\to\beta}{\sigma}]^{-1}t$ by:

$$\begin{array}{rcll} [\overset{\alpha\to\beta}{\sigma}]^{-1}\mathsf{n}_\beta & = & \mathsf{m}_\alpha & \text{if } \mathsf{n}_\beta/\mathsf{m}_\alpha \in \overset{\alpha\to\beta}{\sigma} \text{ and there is no} \\ & & & \mathsf{o}_\alpha \neq \mathsf{o}_\alpha \text{ with } \mathsf{n}_\beta/\mathsf{m}_\alpha \in \overset{\alpha\to\beta}{\sigma}, \\ & & & \text{undefined otherwise} \\ [\overset{\alpha\to\beta}{\sigma}]^{-1}\mathsf{const} & = & \mathsf{const} & \\ [\overset{\alpha\to\beta}{\sigma}]^{-1}u[\overset{\gamma\to\beta}{\sigma_2}] & = & u[\overset{\gamma\to\alpha}{\sigma_3}] & \text{where } \overset{\gamma\to\alpha}{\sigma_3} = [\overset{\alpha\to\beta}{\sigma}]^{-1}\overset{\gamma\to\beta}{\sigma_2} \end{array}$$

and homeomorphic in all other cases. As mentioned before, the matching operation for meta-variables and parameter-variables succeeds only when this operation is defined.

Function whnf from Figure 4.27 requires composing substitutions, our definition of composition is presented in Figure 4.28 it is similar to Figure 4.9, with the addition of context variables to shifts, and the extra rules because now we support replacing for terms and heads when in the simple example we only had terms.

In this section we presented the operational semantics of the language, and two important components of the run-time that are specific to Beluga (or more generally, necessary to support explicit substitutions) that is the weak head normal form function and the substitution composition, in these discussions we kept the Fresh-Style representation

$$
\begin{array}{rcccl}
\mathsf{id} & \circ & \overset{\alpha\to\beta}{\sigma_2} & = & \overset{\alpha\to\beta}{\sigma_2} \\
\underbrace{(\overset{\delta\to\gamma}{\sigma_1}, M_\gamma)}_{\overset{\alpha\to\gamma}{\sigma}} & \circ & \overset{\gamma\to\beta}{\sigma_2} & = & (\overset{\delta\to\gamma}{\sigma_1} \circ \overset{\gamma\to\beta}{\sigma_2}; [\overset{\gamma\to\beta}{\sigma_2}]M_\gamma) \\
\underbrace{(\overset{\delta\to\gamma}{\sigma_1}, H_\gamma)}_{\overset{\alpha\to\gamma}{\sigma}} & \circ & \overset{\gamma\to\beta}{\sigma_2} & = & (\overset{\delta\to\gamma}{\sigma_1} \circ \overset{\gamma\to\beta}{\sigma_2}, [\overset{\gamma\to\beta}{\sigma_2}]H_\gamma) \\
\overset{\alpha\to\gamma}{\uparrow^{c,k}} & \circ & \overset{\gamma\to\beta}{\uparrow^{0,k'}} & = & \overset{\alpha\to\beta}{\uparrow^{c,k+k'}} \\
\overset{\alpha\to\gamma}{\uparrow^{-\psi_{[\cdot]},0}} & \circ & \overset{\gamma\to\delta}{\uparrow^{\psi_{[\cdot]},k}} & = & \overset{\alpha\to\delta}{\uparrow^{0,k}} \quad \text{with } |\Psi_\delta| = k \\
\overset{\alpha\to\gamma}{\uparrow^{\psi_{[\cdot]},0}} & \circ & \overset{\gamma\to\beta}{\uparrow^{-\psi_{[\cdot]},k}} & = & \overset{\alpha\to\beta}{\uparrow^{0,k}} \\
\overset{\alpha\to\gamma}{\uparrow^{c,k+1}} & \circ & \overbrace{(\overset{\delta\to\beta}{\sigma_2}, M_\beta)}^{\gamma\to\beta} & = & \overset{\alpha\to\delta}{\uparrow^{c,k}} \circ \overset{\delta\to\beta}{\sigma_2} \\
\overset{\alpha\to\gamma}{\uparrow^{c,k+1}} & \circ & \overbrace{(\overset{\delta\to\beta}{\sigma_2}, H_\beta)}^{\gamma\to\beta} & = & \overset{\alpha\to\delta}{\uparrow^{c,k}} \circ \overset{\delta\to\beta}{\sigma_2}
\end{array}
$$

Figure 4.28: Substitution composition.

because these components appear in one form or another in whatever back-end we may have.

4.6.2 Names and Indices from Fresh-Style Variables

Figure 4.4 on page 35 presents the interface of the module that implements the Fresh-style bindings. The approach is based in an abstract notion of names that inhabit worlds and links that relate these worlds by introducing a new, always fresh, name. In Section 4.5 we take advantage of this representation to compile pattern matching to a simple language of non-nested patterns. Finally, even if not described here, many optimizations could be written using this representations; this fact simplifies the support of several back-ends each supporting different binding strategy, as it minimizes code duplication.

However, for code generation we need a concrete representation of binders, we will

discuss the two more common, plain-old names and de Bruijn indices. As we can see in Figure 4.4, we have two functions that transform abstract names to de Bruijn indices and to plain old-names.

Function name_to_db uses the information stored in the name abstract type to count how many times it has been imported since it was extracted from the link that introduced the binder, and this is exactly what the de Bruijn index representation is.

On the other hand, function name_to_name simply uses the fact that each name is introduced by a unique link from the previous world to the next, so if we have a $\alpha \leftharpoondown \beta$ which introduces a n_β the functions returns β as the plain old name. In fact, instead of a Greek letter the system uses a number. Calls to this function may fail for names that depend on abstract worlds as we need to substitute the abstract world by a concrete one to be able to figure out the unique name of this variable. This is the reason why the return type for this function is an optional type (it fails for abstract worlds and worlds linked from them).

4.6.3 Back-End and Code-Generation

In a nutshell, a Beluga program consists of computational level functions that manipulate data represented using Contextual LF. Our system compiles the computational level down to a simple λ-calculus like language (with general recursion). This language is a simple functional language without support for nested patterns in pattern matching, making it very easy to generate code in any programming language that supports higher-order functions. As target, the prototype the compiler generates JavaScript code. The choice of language was done purely because of ease of code-generation, but it is not relevant as it is used only to validate the prototype.

Compiling towards more low-level languages or native code is straight-forward, but requires implementing lambda lifting [30] for higher-order functions, a garbage collec-

tor and the low-level run-time support. The choice of (the functional core of) JavaScript provides us with a functional language with support for these features and more than one reasonably fast implementations, so a big part of the run-time is provided by the host language. In the prototype, JavaScript is used as untyped functional language that supports general recursion and high-order functions, so targeting scheme would be similar. It is important to note that the generated code does not depend on the higher-level features of JavaScript, such as object oriented features, prototype inheritance or dynamic dispatch facilities.

However, the manipulation of contextual terms requires that the compiler provides additional run-time support which is detailed in Section 4.6.1. In particular after generating either names or de Bruijn indices (as per Section 4.6.2) we need to provide specific implementations of the matching operations and the whnf function to the run-time library.

4.6.3.1 Contextual LF with Names and de Bruijn Indices

Figures 4.29 and 4.30 show the representation of Contextual LF terms using de Bruijn indices and names respectively, these look similar and the main difference is in the representation of contexts and substitutions. At this point in the compilation types

Normal Term	M,N	$::=$	$\lambda.M \mid H \cdot S$
Head	H	$::=$	$\mathsf{const} \mid x \mid u[\sigma]$
Spine	S	$::=$	$\mathsf{nil} \mid M\,S$
Substitution	σ	$::=$	$\uparrow^{c,k} \mid \sigma, M \mid \sigma, H$
Context Shift	c	$::=$	$-\psi \mid \psi \mid 0$
Context	Ψ,Φ	$::=$	$(\cdot,n) \mid (\psi,n)$

Figure 4.29: Contextual LF with de Bruijn Indices.

are no longer needed so they are eliminated. Their absence simplifies the representation of contexts.

When using de Bruijn indices(Figure 4.29) the translation is straight-forward. Contexts just become a pair of a context variable, that may or may not be present, and an integer with keeps track of the length of the context.

On the other hand, when we use names(Figure 4.30) contexts are lists of variable names that start with either the empty context or a contextual variable. Substitutions still use the same pronominal representation, i.e.: σ, M means replace the top variable in Ψ by M and continue applying σ after dropping the first element of the context as appearances of this element have been replaced.

These two languages correspond to the data level of each of the two back-ends introduced in Section 4.1. They look similar but the implementation of the run-time functions that manipulate them are different. In particular when calculating the weak head normal form of terms, the substitution operation is naturally different, and the matching operations also differs.

Normal Term	M,N	$::=$	$\lambda x.M \mid H \cdot S$
Head	H	$::=$	$\mathsf{const} \mid x \mid u[\sigma] \mid p[\sigma]$
Spine	S	$::=$	$\mathsf{nil} \mid MS$
Substitution	σ	$::=$	$\uparrow^{c,k} \mid \sigma, M \mid \sigma, H$
Context Shift	c	$::=$	$-\psi \mid \psi \mid 0$
Context	Ψ, Φ	$::=$	$\cdot \mid \Psi, x \mid \psi$

Figure 4.30: Contextual LF with names.

4.6.3.2 WHNF and Matching with de Bruijn Indices

The implementation of whnf is straight-forward. Nonetheless, when getting the weak head normal form of a term that consists of a name and a spine, i.e.:$x \cdot S$ we need to perform the substitution in x. Figure 4.31 shows how to perform the substitution on bound LF variable represented by a de Bruijn index.

Furthermore, the matching operations are defined in Figure 4.32. Context matching succeeds as long as the there are *enough variables* in the matched context. It is important to know that this definition is just for clarity and not performance, in a more realistic implementation lengths of the contexts are compared and the length of the context on the branch needs to be less or equal than that of the the matched context.

4.6.3.3 WHNF and Matching with Names

Similarly to the previous section, Figures 4.33 and 4.34 show how to perform bound variable substitution and how to implement the matching operations when using names in the data-level language.

Performing a substitution for a variable name is straightforward and it follows the presentation in [7] and [18].

Finally the match operations, are defined in Figure 4.34 which again are similar to previous definitions of the function but accounts for the different structure of contexts and variable binders.

$$
\begin{array}{rcrcl}
[\sigma, M] & 1 & = & M \\
[\sigma, H] & 1 & = & H \\
[\sigma, M] & n & = & [\sigma](n-1) \\
[\sigma, H] & n & = & [\sigma](n-1) \\
[\uparrow^{c,k}] & n & = & n+k
\end{array}
$$

Figure 4.31: Bound variable substitution with de Bruijn indices.

Context Matching:

$$\frac{}{_ \overset{ctx}{\approx} (\cdot, 0)} \quad \frac{}{_ \overset{ctx}{\approx} (\psi, 0)} \quad \frac{(c, n-1) \overset{ctx}{\approx} (d, n-1)}{(c, n) \overset{ctx}{\approx} (d, n)}$$

Term Matching:

$$\frac{\mathsf{con} = \mathsf{const}}{\Psi.\mathsf{con}\ \vec{V} \overset{M}{\approx} \Phi.\mathsf{const} \cdot S} \quad \frac{}{\Psi.\lambda.V \overset{M}{\approx} \Phi.\lambda.M}$$

$$\frac{[\sigma]^{-1}M}{\Psi.u[\sigma] \overset{M}{\approx} \Phi.M} \qquad \frac{[\sigma]^{-1}x}{\Psi.p[\sigma] \overset{M}{\approx} \Phi.x}$$

$$\frac{x = y}{\Psi.x \overset{M}{\approx} \Phi.y}$$

Figure 4.32: Matching operations with de Bruijn indices.

$$\begin{array}{llllll}
\Psi & [\sigma, M] & x & = & M & \text{if } \Psi = \Psi', y \text{ and } x = y \\
\Psi & [\sigma, H] & x & = & H & \text{if } \Psi = \Psi', y \text{ and } x = y \\
\Psi & [\sigma, M] & x & = & \Psi'\ [\sigma]x & \text{if } \Psi = \Psi', y \text{ and } x \neq y \\
\Psi & [\sigma, H] & x & = & \Psi'\ [\sigma]x & \text{if } \Psi = \Psi', y \text{ and } x \neq y \\
\Psi & \left[\uparrow^{c,k}\right] & x & = & x &
\end{array}$$

Figure 4.33: Bound variable substitution with names.

Context Matching:

$$\frac{}{\cdot \overset{ctx}{\approx} \cdot} \quad \frac{}{\psi \overset{ctx}{\approx} \phi} \quad \frac{}{_ \overset{ctx}{\approx} \psi} \quad \frac{\Psi \overset{ctx}{\approx} \Phi}{\Psi, x \overset{ctx}{\approx} \Phi, y}$$

Term Matching:

$$\frac{\mathsf{con} \overset{M}{\approx} \mathsf{const}}{\Psi.\mathsf{con}\ \vec{V} \overset{M}{\approx} \Phi.\mathsf{const} \cdot S} \quad \frac{}{\Psi.\lambda\, x.V \overset{M}{\approx} \Phi.\lambda\, y.M}$$

$$\frac{[\sigma]^{-1} M}{\Psi.u[\sigma] \overset{M}{\approx} \Phi.M} \quad \frac{[\sigma]^{-1} x}{\Psi.p[\sigma] \overset{M}{\approx} \Phi.x}$$

$$\frac{}{\Psi', x.\ x \overset{M}{\approx} \Phi', y.\ y} \quad \frac{\Psi'.\ x \overset{M}{\approx} \Phi'.\ y}{\Psi', x'.\ x \overset{M}{\approx} \Phi', y'.\ y}$$

Figure 4.34: Matching operations with names.

CHAPTER 5

CONCLUSION

The goal of this work is to present a compiler for the Beluga programming language and proof development environment. The compiler uses an abstract representation of bound variables to be able to implement as much as possible of the system without committing to using names or de Bruijn indices. To motivate our work, we presented some of the existing techniques used to represent binders in Chapter 2. Then we introduced the Beluga language, dependent types and the internal representation of such programs in Chapter 3. Finally in Chapter 4 we described the concrete implementation of the compiler. Of particular importance are the representation of binders using the *Fresh-style* approach to compile the *very expressive pattern language*. The use of an abstract representation (i.e.: the fresh-style binders) for bound variables permitted us to share the pattern matching compilation between the name-based back-end path and the de Bruijn index based one.

The main contributions of this work are a compilation scheme for contextual objects, an internal representation that is able to bring higher-order representations of binders to two first order techniques, names and de Bruijn indices, this work relates for the first time Contextual Modal Type Theory and nominal techniques. These contributions are significant because they are a step forward to making HOAS and contextual objects available in general purpose functional programming languages. Finally, Beluga's contextual values require a very powerful pattern matching language, in the beginning it was not clear how to compile this language, and if it would be possible to do it using the fresh-style binders. This work shows an implementation of pattern compilation for the rich language that Beluga supports and it shows what operations can be performed at

compile-time (computing the splitting tree) and which need to be performed at run-time (the term and contextual matching operations).

5.1 Future Work

The focus of this work is in the fresh-style representation, but many important questions remain, some of the paths now open to exploration are:

- The fresh-style representation is very versatile, and in many ways it combines the strengths and weaknesses of nominal and index based approaches, it remains to explore how and which optimizations can be expressed in this representation, and which optimizations are specific to one representation and need to be applied after the conversion to names or indices.
- The compiler uses a simply typed internal representation, and this fact was very useful to find bugs and problems with the code, it would be interesting to understand how to extend this to dependent types and to explore the feasibility of type preserving compilation.
- The computation of the splitting tree for pattern matching compilation and the coverage checking algorithm ([48]) use similar ideas, exploring the exact relationship between them could be interesting and also help expand this work to dependent types.
- Having a compiler that generates code using names and de Bruijn indices gives the opportunity of experimentally evaluating the performance of both approaches.

BIBLIOGRAPHY

[1] M. Abadi, L. Cardelli, P. l. Curien, and J. j. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1:31–46, 1991.

[2] Andreas Abel. Termination checking with types. *RAIRO – Theoretical Informatics and Applications*, 38(4):277–319, 2004. Special Issue: Fixed Points in Computer Science (FICS'03).

[3] Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. In Luke 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.

[4] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, 2005.

[5] Lennart Augustsson. Compiling pattern matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture (FPCA'85)*, volume 201 of *Lecture Notes in Computer Science (LNCS)*, pages 368–381. Springer, 1985.

[6] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics (Studies in Logic and the Foundations of Mathematics, Volume 103). Revised Edition*. North Holland, revised edition, 1985.

[7] Henk Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[8] Stefan Berghofer and Christian Urban. A head-to-head comparison of de bruijn indices and names. *Electroninc Notes in Theoretical Computer Science*, 174(5): 53–67, June 2007.

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

[10] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM Conference on LISP and functional programming*, LFP '80, pages 136–143, New York, NY, USA, 1980. ACM.

[11] Luca Cardelli. Compiling a functional language. In *ACM Symposium on LISP and Functional Programming (LFP'84)*, pages 208–217. ACM, 1984.

[12] Andrew Cave and Brigitte Pientka. Programming with binders and indexed datatypes. In *39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. ACM Press, 2012.

[13] Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

[14] Kaustuv Chaudhuri. Illustration for de bruijn index. https://en.wikipedia.org/wiki/File:De_Bruijn_index_illustration_1.svg.

[15] J. Cheney and C. Urban. Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In Bart Demoen and Vladimir Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004)*, LNCS (3132), pages 269–283, 2004.

[16] Adam J. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 143–156. ACM, 2008.

[17] Thierry Coquand. Pattern matching with dependent types. In *Informal Proceedings of Workshop on Types for Proofs and Programs*, pages 71–84. Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., 1992. URL http://citeseer.ist.psu.edu/310402.html.

[18] Haskell B. Curry. *Combinatory logic / [by] Haskell B. Curry [and] Robert Feys. With two sections by William Craig*. North-Holland Pub. Co., Amsterdam, :, 1958. ISBN 0720422086.

[19] N. G. de Bruijn. The mathematical language automath, its usage, and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer Berlin / Heidelberg, 1970.

[20] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[21] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. In D. Kozen, editor, *10th Annual Symposium on Logic in Computer Science*, pages 366–374, San Diego, California, June 1995. IEEE Computer Society Press.

[22] Amy Felty and Alberto Momigliano. Hybrid. *Journal of Automated Reasoning*, 48:43–105, 2012.

[23] Murdoch Gabbay and Andrew 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.

[24] Murdoch J. Gabbay. A study of substitution, using nominal techniques and Fraenkel-Mostowksi sets. *Theoretical Computer Science*, 410(12-13):1159–1189, March 2009. ISSN 0304-3975.

[25] Louis-Julien Guillemette and Stefan Monnier. A type-preserving closure conversion in Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, Haskell '07, pages 83–92, 2007.

[26] Louis-Julien Guillemette and Stefan Monnier. A type-preserving compiler in Haskell. pages 75–86, 2008.

[27] Robert Harper. *Practical Foundation of Programming Languages*. Working draft. Version 1.32 of 05.15.2012, 2012.

[28] Robert Harper and Frank Pfenning. On equivalence and canonical form1s in the LF type theory. *ACM Transactions on Computational Logic*, 6(1):61–101, January 2005. ISSN 1529-3785.

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

[30] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[31] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 26–37, New York, NY, USA, 2001. ACM.

[32] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[33] Pierre Letouzey. A new extraction for coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *Lecture Notes in Computer Science*, pages 617–617. Springer Berlin / Heidelberg, 2003.

[34] Pierre Letouzey. Extraction in coq: An overview. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer Berlin / Heidelberg, 2008.

[35] Daniel R. Licata and Robert Harper. A universe of binding and computation. In Graham Hutton and Andrew P. Tolmach, editors, *14th ACM SIGPLAN International Conference on Functional Programming*, pages 123–134. ACM Press, 2009.

[36] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 35–46, New York, NY, USA, 2008. ACM.

[37] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990. ISBN 0-262-63132-6.

[38] Stefan Monnier and David Haguenauer. Singleton types here, singleton types there, singleton types everywhere. In *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages Meets Program Verification*, PLPV '10, pages 1–8, New York, NY, USA, 2010. ACM.

[39] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

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

[41] Nicolas Oury and Wouter Swierstra. The power of pi. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 39–50, New York, NY, USA, 2008. ACM.

[42] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[43] Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*. Springer-Verlag Lecture Notes Computer Science 1657, 1998.

[44] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206. Springer, 1999.

[45] Brigitte 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.

[46] Brigitte Pientka. Programming inductive proofs - a new approach based on contextual types. In Simon Siegler and Nathan Wasser, editors, *Verification, Induction, Termination Analysis*, pages 1–16. Springer-Verlag, 2010.

[47] Brigitte Pientka. An insider's look at LF type reconstruction: Everything you (n)ever wanted to know. *Journal of Functional Programming*, 2011. revised version submitted in June 2011.

[48] Brigitte Pientka and Joshua Dunfield. Covering all bases: design and implementation of case analysis for contextual objects. Technical report, McGill University, 2010.

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

[50] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, November 2003. ISSN 0890-5401.

[51] Adam Poswolsky and Carsten 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.

[52] Adam Brett Poswolsky. *Functional Programming with Logical Frameworks: The Delphin Project*. CreateSpace, Paramount, CA, 2008. ISBN 1440474923, 9781440474927.

[53] Nicolas Pouillard. Nameless, painless. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 320–332, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6.

[54] Nicolas Pouillard and François Pottier. A fresh look at programming with names and binders. In *15th ACM SIGPLAN International Conference on Functional Programming (ICFP'10)*, pages 217–228, 2010.

[55] Kristoffer H. Rose. Explicit substitution – tutorial & survey. Technical Report LS-96-3, September 1996. v+150 pp.

[56] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03)*, pages 120–135. Springer, 2003.

[57] Tim Sheard. Languages of the future. *SIGPLAN Notices*, 39(12):119–132, 2004.

[58] Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: programming with binders made simple. In *8th International Conference on Functional Programming (ICFP'03)*, pages 263–274. ACM Press, 2003.

[59] Christian Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reason.*, 40(4): 327–356, May 2008. ISSN 0168-7433.

[60] Geoff Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Journal of Functional Programming*, to appear, 2006.

[61] Edwin Westbrook, Nicolas Frisby, and Paul Brauner. Hobbits for haskell: a library for higher-order encodings in functional programming languages. In *Proceedings*

of the 4th ACM symposium on Haskell, Haskell '11, pages 35–46, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0860-1.