Compiling Contextual Objects

Bringing Higher-Order Abstract Syntax to Programmers

Francisco Ferreira
McGill University
fferre8@cs.mcgill.ca

Stefan Monnier
Université de Montréal
monnier@iro.umontreal.ca

Brigitte Pientka
McGill University
bpientka@cs.mcgill.ca

Abstract

Binders in data-structures representing code or proofs can be represented in a variety of ways, from low-level first-order representations such as de Bruijn indices to higher-order abstract syntax (HOAS), with nominal logic somewhere in-between.

HOAS is arguably the cleanest and highest-level representation but comes with significant problems in expressiveness and efficiency. The Beluga language addresses this expressiveness problem by providing a powerful pattern matching facility as well as explicit control over contexts.

This work aims to solve one important efficiency concern by showing how to compile Beluga down to lower-level primitives. It does so by compiling Beluga's binders into an intermediate first-order representation that abstracts over the eventual low-level representation, and by adapting ML-style pattern compilation to the more general case of Beluga's patterns.

As an important side benefit, our work clarifies the connection between programming with HOAS in Beluga and programming with first-order approaches based on names.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords Contextual objects, compilation, pattern matching

1. Introduction

A key aspect when implementing compilers, code generators, interpreters, type inference engines, or theorem provers is the representation of data structures with names and binders. Historically, binders in data-structures such as abstract syntax trees have been represented in a variety of ways, mostly governed by software engineering concerns: is the representation efficient, reasonably simple to understand, and sufficiently easy to manipulate without introducing bugs by accident. Depending on the particular domain, different representations were used: e.g., de Bruijn indices when fast α-equivalence checking is important or unique small integers when the flexibility of moving code freely is a key concern.

Besides supporting clear and easy to maintain code, the growing interest in formal methods has added a new concern: how can we establish and track properties of binders and the code that manipulates them without obfuscating the development.

PLPV'13, January 22, 2013, Rome, Italy.

To this end, several systems have been developed with specialized support for binders, either via libraries or as first-class notions, to try and encapsulate as much as possible the common functionality of binders and prove their properties ideally once and for all, or at least more easily. Currently, there are two main theoretical approaches: supporting *nominals*, that is *names* as first-class notions, in the language ([Pitts 2003], [Pouillard and Pottier 2010]) and Higher-Order Abstract Syntax (HOAS) [Pfenning and Elliott 1988], which represents binders in the object level by reusing binders of the meta-language, implemented by systems like Twelf [Pfenning and Schürmann 1999], Delphin[Poswolsky and Schürmann 2009], and Beluga [Pientka and Dunfield 2010].

Over the past decade, we have made substantial progress in using sophisticated binder support and manipulating HOAS representations to effectively model proofs. However, in programming languages such support for binders is at best spotty. In the case of languages and libraries, techniques such as FreshML [Shinwell et al. 2003], folds or catamorphisms [Washburn and Weirich 2008], and Hobbits [Westbrook et al. 2011] all address the problem, but they either lack the formal guarantees, are complicated and inconvenient to use, or do not scale to dependent types.

Beluga [Pientka 2008; Pientka and Dunfield 2010] is a novel language specifically designed to manipulate data-structures with binders, providing all the usual operations we know and love, such as seamless α-renaming and capture-avoiding substitution, cleanly integrating them with dependent types. It starts with the logical framework LF [Harper et al. 1993] which allows for HOAS representations and from where it gets its high-level and clean semantics. LF objects together with their surrounding context in which they are meaningful are then embedded into computations thereby lifting the usual limitations of HOAS which typically prevent us from accessing and comparing variables. We call LF objects with their surrounding context *contextual objects* [Nanevski et al. 2008]. On the level of computations, Beluga provides recursion and a powerful pattern matching construct to analyze contextual objects. This makes it feasible to elegantly write code transformations such as for example closure conversion and hoisting.

While the theory behind Beluga is well developed, it remains unclear how to implement it efficiently and make its technology available in a realistic programming language. The current Beluga implementation uses an interpreter that largely follows the theoretical presentation. This allows us to explore the power of the language in small scale examples. In order to make it into a practical programming language however, it is important to be able to compile it into efficient code. Efficiency of the compiled code depends mostly on two different aspects:

- How to represent binders and contexts: Experience in compilers indicates that the best representation to use can depend on the particular binders. For example, SML/NJ's [Shao 1997] internal representation uses names for variables bound in the

computation code, but de Bruijn indices for variables bound within types.

- How to implement pattern matching: Beluga's patterns are powerful, and we use a specialization of higher-order pattern unification [Abel and Pientka 2011; Dowek et al. 1996], a decidable fragment of higher-order unification, for pattern matching. But of course, we do not want to compile Beluga's case to a sequence of calls to a generic higher-order pattern unification function.

In this paper, we describe how to compile contextual objects and contexts. The compilation is split into 3 parts:

1. We give a translation of code into an internal first-order representation that we call *Fresh-Style* , inspired by [Pouillard and Pottier 2010]. The Fresh-Style representation abstracts over the actual final representation and provides an abstract interface from which we can derive a concrete representation using names or de Bruijn indices. This constitutes our intermediate representation. Our translation between contextual objects and their corresponding Fresh-Style representation connects for the first time formally the gap between programming using the nominal approach and programming using contextual objects.
2. We describe pattern-matching compilation that turns the case statements in Beluga programs into a decision tree as is usually done with ML patterns. In particular, we extend ML-style pattern matching compilation to the more general case of Beluga's patterns, that can match on contexts, open terms, and λ-abstractions.
3. We show how to instantiate the abstract Fresh-Style representation to obtain a concrete representation of the binders using names and de Bruijn indices.

Our compiler framework is implemented in OCaml. The internal representation of the program in the prototype makes it easy to target a generic simply typed functional language. At this point, the prototype generates JavaScript code.

2. Overview

Beluga [Pientka 2008; Pientka and Dunfield 2010] is a dependently-typed programming and proof environment which consists of two levels: 1) We can specify formal systems such as intrinsically typed lambda-terms, type systems, or operational semantics in the logical framework LF [Harper et al. 1993]. This allows us to exploit the function space of LF (i.e. the meta-language) to model bindings in our object language and our object language encodings can be described using higher-order abstract syntax representations. As a consequence, the user inherits from LF (i.e. the meta-language) substitution and α-renaming. 2) On top of the logical framework LF, Beluga provides a dependently typed computational language which supports recursion and pattern matching. An LF object M of type A can be embedded into the computation language by packaging M with the context Ψ in which it is meaningful, to describe a contextual LF object $[\hat{\Psi}.M]$ of type $A[\Psi]$ [Nanevski et al. 2008] where $\hat{\Psi}$ can be obtained by dropping the type declarations in the typing context Ψ. To put it differently, $\hat{\Psi}$ describes the free variables occurring in M. What distinguishes Beluga's computation language from other functional languages is its supports for analyzing and manipulating contexts and contextual objects using pattern matching. The question we investigate in this paper is how to compile contexts and contextual objects. This is a key step in bringing the power of HOAS representation techniques to the ordinary programmer.

In our compiler, we use a technique derived from Pouillard and Pottier [2010] for the internal representation which we call the Fresh-Style representation. In this approach, we consider abstract names that inhabit worlds related by links. We interpret worlds as introducing scope and links as binders while keeping the notion of name abstract. This provides us with a common intermediate language to generate in the target language, binders with de Bruijn indices and names.

2.1 Example and challenges

Beluga is well adapted to reasoning about the meta-theory of programming languages. An inductive proof such as subject reduction corresponds to a recursive program about typing and evaluation derivations. More generally, Beluga is suitable for programs that manipulate data-structures with binders. Problems which require dealing with binders not only arise when we are modelling typing derivations and the reasoning about them, but already come up when we are implementing interpreters, type checkers, code generators, compilers, or decision procedures.

To illustrate the power of Beluga, we discuss translating intrinsically typed lambda-terms into typed de Bruijn representations. We begin by defining a type `tp` which has two constants: the type for individuals, `i : tp`, and function types `arr: tp →tp →tp`. Next, we define intrinsically typed lambda-terms by defining a type family `exp` which is indexed by `tp`:

```
datatype exp : tp → type =
| app : exp (arr A B) → exp A → exp B
| lam : (exp A → exp B) → exp (arr A B);
```

We represent the variable binding in the lambda-expression using the LF function `(exp A→exp B)`. For a thorough introduction to representing formal systems in LF we refer the reader to Pfenning [2000]. The definition for de Bruijn terms is straightforward.

```
datatype db : tp → type =
| one : db A
| shift: db A → db A
| lam' : db B → db (arr A B)
| app' : db (arr A B) → db A → db B;
```

While we do not enforce that all de Bruijn terms are well-typed, our definition is sufficiently strong to allow us to establish that the program `hoas2db` which translates a given object into its corresponding object preserves types. As we traverse an expression and translate the body of a lambda-expression, the object we are translating is not closed anymore. We hence translate an object of contextual type `[g.exp T]` where `g` is a context to a closed de Bruijn object `[.dB T]` of the same type. We use `.` to separate the context from the object. `hoas2db` hence has type `(g:ctx)[g.exp T]→[.db T]`. Implicit context quantification over the context `g` is modelled via `(g:ctx)` in our concrete syntax. Beluga overloads →: if → occurs in the type of a computation, then it denotes a function which can be defined via recursion and pattern matching. If it occurs in the LF type, it is a weak function space which purely models bindings.

Just as types classify terms, contexts are classified by schemas. In the example the schema `ctx` which is associated with the context `g` is declared as follows:

```
schema ctx = exp S;
```

It says that all declarations in a context of schema `ctx` must contain declarations `exp S` for some `S`. For example, `x:exp i,y:exp (arr i i)` is a valid context of schema `ctx`, but `a:tp,x:exp i` would not be a valid context of schema `ctx`.

The program `hoas2db` is written recursively by pattern matching on contextual object `e` of type `[g. exp T]`, i.e. `e` is an object of type `exp T` in the context `g`. There are several cases to consider: it could be an application, a lambda-expression or it could be a variable.

```
rec hoas2db : (g:ctx) [g. exp T] → [. db T] =
fn e ⇒ case e of
| [g, x:exp T . x] ⇒ [. one]
| [g, x:exp T . #p ..] ⇒
  let [. F] = hoas2db [g. #p ..] in
      [. shift F]
| [g. lam (λx. E .. x)] ⇒
  let [. F] = hoas2db [g,x:exp _. E .. x] in
    [. lam' F]
| [g. app (E1 .. ) (E2 .. )] ⇒
  let [. F1] = hoas2db [g. E1 ..] in
  let [. F2] = hoas2db [g. E2 ..] in
    [. app' F1 F2];
```

If `e` is `[g. app (E1 ..)(E2 ..)]`, then we recursively translate `[g. E1 ..]` and `[g.E2 ..]`. Since `E1` describes an object which may contain variables from the context `g`, we associate `E1` with the identity substitution written as `...` We call `E1` and `E2` meta-variables. They can be instantiated with an expression containing variables from `g`.

If `e` is `[g, lam λx. E .. x]`, we recursively translate the body `[g, x:exp _ . E .. x]` where we extend the context with a new declaration `x:exp _`. We write here an underscore for the type of `x`, since we do not have access to its concrete type, and let type reconstruction infer it.

Finally, we consider the case where `e` is a variable. Its corresponding de Bruijn index is determined by its position in the context `g`. Hence, we pattern patch on `g` and `e` simultanously. If `e` denotes the first variable in the context (i.e. `[g,x:exp T. x]`) then we return the de Bruijn index `[. one]`. If `e` denotes another variable from the context `g` (i.e. `[g,x: exp T. #p ..]`) then we recursively translate `[g.#p ..]` and shift its result. `#p` describes a parameter variable which can only be instantiated with a variable from `g`.

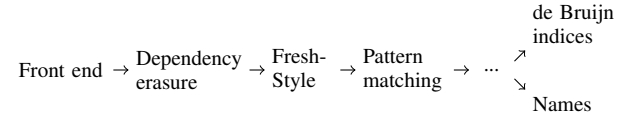
The following table summarizes the different kinds of patterns.

Matching on	Pattern	Matches
The shape of contexts	`g`	An arbitrary context.
	`g,x:exp T`	A context with at least one variable in it.
	␣ (i.e. nothing)	The empty context as in `[.F]`.
Constructors	`app` `lam`	
Variables in the context	`x`	The variable of that name in the context pattern `g, x:exp T`.
Parameter variables	`#p..`	Any variable from the context `g`. The `..` means it can depend only on variables from `g`.
Meta-variables	`E..x`	Any term that depends on the context variable plus the variable `x`.
λ-patterns	λx.	A term that begins with an abstraction.

To summarize the key aspect of Beluga: we can pattern match on contexts and contextual objects, i.e. objects which are meaningful within a context. In general, pattern matching may require higher-order pattern matching.

2.2 Compiler Pipeline

The compiler is structured in the usual pipeline of transformation phases. The part of the pipeline relevant here is as follows:



The front end consists of the parser, the type reconstruction and type checking, and the coverage checker. The output is a fully annotated program [Cave and Pientka 2012].

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 often the case when Beluga is used as a proof assistant. Nonetheless, many case studies in Beluga are interpreters, compilers, code generators and optimizers, and decision procedures. Such programs are where execution matters and in consequence are the focus of the compiler.

When we translate Beluga programs to their Fresh-Style representation, we also erase type dependencies *at the same time*. Erasing dependencies simplifies the type annotations by replacing the dependent types with a simple typing discipline. Since the program has already been fully type-checked, we could completely drop types. However, we decided to keep simple types, to be able to type check our translated code and detect compiler bugs early on. The choice of dropping the dependent types at this particular stage is largely arbitrary, since we could probably do it later, but likely no later than closure conversion.

The Fresh-Style conversion turns the bound variables in contextual LF objects into a first-order representation that abstracts over the particular concrete representation of those variables. We call this representation the Fresh-Style representation to relate it with Pouillard and Pottier [2010]. This representation is sufficiently high-level to allow various concrete representations, yet it is sufficiently low-level that we can express the decomposition of Beluga's higher-order pattern matching into a decision tree, as done in the next phase. The ellipsis in the above picture represents various classical optimization phases which can then be applied to clean up and simplify the code, although our prototype does not yet include any of those phases.

The treatment of binders ends finally with the last depicted phase which makes the variable representation concrete. We currently have implemented two concrete representations, one which uses de Bruijn indices for bound variables, and another that uses simple names for them.

3. Beluga's Theory

Beluga is a dependently-typed functional language which supports contexts and contextual objects as basic objects. It is designed to be parametric in the base domain. As Beluga's computation language is similar to functional languages such as ML and poses no particular challenges to compilation, we concentrate on compiling contexts and contextual LF objects.

3.1 Contextual LF

Contextual LF extends the logical framework LF [Harper et al. 1993] with the power of contextual objects $\hat{\Psi}.M$ of type $A[\Psi]$. M denotes an object which may refer to the bound variables listed in $\hat{\Psi}$ and has type A in the context Ψ (see also [Nanevski et al. 2008]). As usual, we assume that type constants **a** and term constants **c** are specified together with their kinds and types in a signature Σ.

Atomic types	P	::=	$\mathbf{a}\,\vec{M}$
Types	A, B	::=	$P \mid \Pi A.\, B$
Heads	H	::=	$x \mid \mathbf{c} \mid p[\sigma]$
Neutral Terms	R	::=	$H \mid R\, N \mid u[\sigma]$
Normal Terms	M, N	::=	$R \mid \lambda.M$
Context Shifts	c	::=	$0 \mid \psi \mid -\psi$
Substitutions	σ	::=	$\uparrow^{c,k} \mid \sigma, M \mid \sigma; H$
Contexts	Ψ	::=	$\cdot \mid \psi \mid \Psi, A$

We consider here a variant where we model bound variables via de Bruijn indices and explicit substitutions [Abadi et al. 1990]. We concentrate on normal forms in β-η-long form since these are the only meaningful objects in the logical framework. Furthermore, we concentrate here on characterizing well-typed terms(Figure 1), but omit kinds and kinding rules for types.

Normal objects may contain *ordinary bound variables* which are used to represent object-level binders and are bound by λ-abstraction or in a context Ψ. Such bound variables are modelled via de Bruijn indices and hence we omit their name in the λ-abstraction. Normal objects may also contain meta-variables $u[\sigma]$ and parameter variables $p[\sigma]$ which we call *contextual variables*. Contextual variables are associated with a postponed substitution σ. The meta-variable u stands for a contextual object $\hat{\Psi}.R$ where $\hat{\Psi}$ describes the ordinary bound variables which may occur in R. When concentrating on a de Bruijn representation, $\hat{\Psi}$ can be simply modelled by a context variable together with a number k which denotes the upper bound to the indices occurring in R. To put it differently, k describes the number of concrete declarations in the context Ψ. Without loss of generality we require that meta-variables have base type. The parameter variable p stands for a contextual object $\hat{\Psi}.R$ where R must be either an ordinary bound variable from $\hat{\Psi}$ or another parameter variable.

As is common, we rely on hereditary substitutions, written as $[N/x]_A(B)$ (or $[\sigma]_\Psi(B)$) to guarantee that when we substitute a term N which has type A for the variable x in the type B, we obtain a type B' which is in normal form. Hereditary substitutions continue to substitute, if a redex is created; for example, when replacing naively x by $\lambda y.c\ y$ in the object $x\ z$, we would obtain $(\lambda y.c\ y)\ z$ which is not in normal form and hence not a valid term in our grammar. Hereditary substitutions continue to substitute z for y in $c\ y$ to obtain $c\ z$ as a final result [Nanevski et al. 2008].

In the simultaneous substitutions σ, we do not make its domain explicit. Rather we think of a substitution together with its domain Ψ and the i-th element in σ corresponds to the i-th declaration in Ψ. We have two different ways of building a substitution: either by using a normal term M or a variable x. Note that a variable x is only a normal term M if it is of base type. However, as we push a substitution σ through a λ-abstraction $\lambda.M$, we need to extend σ with x whose index is 1. The resulting substitution σ, x may not be well-typed, since x may not be of base type and in fact we do not know its type. Hence, we allow substitutions not only to be extended with normal terms M but also with variables x. The presence of context variables, gives rise to a shift (denoted by $\uparrow$) which is annotated with two superscripts:a number k denoting by how many concrete declarations the result needs to be shifted and by flag c denoting the shift by a context ψ. We summarize the different variations needed below.

Range	Substitution	Domain
$x_1{:}A_1, \ldots, x_n{:}A_n$	$\uparrow^{0,n}$	$\cdot$
$\psi, x_1{:}A_1, \ldots, x_n{:}A_n$	$\uparrow^{0,n}$	ψ
$\psi, x_1{:}A_1, \ldots, x_n{:}A_n$	$\uparrow^{\psi,n}$	$\cdot$
$x_1{:}A_1, \ldots, x_n{:}A_n$	$\uparrow^{-\psi,n}$	ψ

A bound variable context Ψ contains bound variable declarations in addition to context variables. A context may only contain at most one context variable and it must occur at the left.

3.2 Meta-terms and Meta-types

We lift contextual LF objects to meta-types and meta-objects to treat meta-objects uniformly. Meta-objects are either contextual objects written as $\hat{\Psi}.R$ or contexts Ψ. These are the index objects which can be used to index computation-level types. There are three different meta-types: $P[\Psi]$ denotes the type of a meta-variable u and stands for a general contextual object $\hat{\Psi}.R$. $\#A[\Psi]$ denotes the type of a parameter variable p and it stands for a variable object, i.e. either $\hat{\Psi}.x$ or $\hat{\Psi}.p[\pi]$ where π is a variable substitution. A variable substitution π is a special case for general substitutions σ; however unlike $p[\sigma]$ which can produce a general LF object, $p[\pi]$ guarantees we are producing a variable. G describes the schema (i.e. type) of a context. The tag $\#$ on the type of parameter variables is a simple syntactic device to distinguish between the type of meta-variables and parameter variables. The meta-context in which an LF object appears uniquely determines if X denotes a meta-variable, parameter variable or context variable. We use the following convention: if X denotes a meta-variable we usually write u or v; for a parameter-variable, we write p and for context variables we use ψ.

Neutral Terms $\boxed{\Delta; \Psi \vdash R \Rightarrow A}$

$$\frac{\Sigma(\mathbf{c}) = A}{\Delta; \Psi \vdash \mathbf{c} \Rightarrow A} \qquad \frac{\Delta(p) = \#A[\Phi] \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta; \Psi \vdash p[\sigma] \Rightarrow [\sigma]_\Phi A}$$

$$\frac{\Psi(x) = A}{\Delta; \Psi \vdash x \Rightarrow A} \qquad \frac{\Delta(u) = P[\Phi] \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta; \Psi \vdash u[\sigma] \Rightarrow [\sigma]_\Phi P}$$

$$\frac{\Delta; \Psi \vdash R \Rightarrow \Pi A.B \quad \Delta; \Psi \vdash M \Leftarrow A}{\Delta; \Psi \vdash R\,M \Rightarrow [M/x]_A B}$$

Normal Terms $\boxed{\Delta; \Psi \vdash M \Leftarrow A}$

$$\frac{\Delta; \Psi \vdash R \Rightarrow P \quad \Delta; \Psi \vdash P = Q}{\Delta; \Psi \vdash R \Leftarrow Q} \qquad \frac{\Delta; \Psi, A \vdash M \Leftarrow B}{\Delta; \Psi \vdash \lambda.M \Leftarrow \Pi A.B}$$

Substitutions $\boxed{\Delta; \Psi \vdash \sigma \Leftarrow \Psi'}$

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

$$\frac{\Delta; \Psi \vdash \uparrow^{c,k} \Leftarrow \cdot}{\Delta; \Psi, A \vdash \uparrow^{c,k+1} \Leftarrow \cdot} \qquad \frac{\Delta; \Psi \vdash \uparrow^{c,k} \Leftarrow \psi}{\Delta; \Psi, A \vdash \uparrow^{c,k+1} \Leftarrow \psi}$$

$$\frac{\Delta; \Psi \vdash \sigma \Leftarrow \Psi' \quad \Delta; \Psi \vdash M \Leftarrow [\sigma]_{\Psi'} A}{\Delta; \Psi \vdash \sigma, M \Leftarrow \Psi', A}$$

$$\frac{\Delta; \Psi \vdash \sigma \Leftarrow \Psi' \quad \Delta; \Psi \vdash H \Rightarrow B \quad \Delta; \Psi \vdash [\sigma]_{\Psi'} A = B}{\Delta; \Psi \vdash \sigma; H \Leftarrow \Psi', A}$$

Figure 1. Typing Rules for Contextual LF

Context schemas	G	::=	$\exists(\overrightarrow{x:A}).B \mid G + \exists(\overrightarrow{x:A}).B$
Meta Terms	C	::=	$\hat{\Psi}.R \mid \Psi$
Meta Types	U	::=	$P[\Psi] \mid \#A[\Psi] \mid G$
Meta substitutions	θ	::=	$\cdot \mid \theta, C$
Meta-context	Δ	::=	$\cdot \mid \Delta, X{:}U$

Context schemas consist of different schema elements $\exists(\overrightarrow{x:A}).B$ which are built using $+$. Intuitively, this means a concrete declaration in a context must be an instance of one of the elements specified in the schema. For example, a context $x{:}\mathsf{exp\ nat}, y{:}\mathsf{exp\ bool}$ will check against the schema $\exists T{:}\mathsf{tp.exp}\ T$.

The uniform treatment of meta-terms, called C, and meta-types, called U, allows us to give a compact definition of meta-substitutions θ and meta-contexts Δ. This achieves a modular design of the computation language. We omit here the rules stating when meta-types and meta-contexts are well-formed and show only the typing rules for meta-terms and meta-substitutions in Figure 2.

3.3 Computation Language

We present in this section a dependently typed programming language Beluga along the lines of Mini-ML. Our language of computations includes recursion (written as $\mathsf{rec}\ f.E$), nameless functions (written as $\mathsf{fn}\ x.E$) and dependent functions (written as $\lambda X.\ E$). We have two different kinds of function applications, one for ap-

Meta Terms $\boxed{\Delta \vdash C \Leftarrow U}$

$$\frac{}{\Delta \vdash \cdot \Leftarrow G} \qquad \frac{\Delta(\psi) = G}{\Delta \vdash \psi \Leftarrow G}$$

$$\frac{\Delta \vdash \Psi \Leftarrow G \quad \exists(\overrightarrow{x : B'}).B \in G \quad \Delta; \Psi \vdash \sigma \Leftarrow (\overrightarrow{x{:}B'}) \quad A = [\sigma]_{\overrightarrow{(x:B')}}B}{\Delta \vdash \Psi, x{:}A \Leftarrow G}$$

$$\frac{\Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta \vdash \hat{\Psi}.\sigma \Leftarrow \Phi[\Psi]} \qquad \frac{\Delta; \Psi \vdash R \Leftarrow P}{\Delta \vdash \hat{\Psi}.R \Leftarrow P[\Psi]} \qquad \frac{\Psi(x) = A}{\Delta \vdash \hat{\Psi}.x \Leftarrow \#A[\Psi]}$$

$$\frac{\Delta(p) = \#A[\Phi] \qquad \Delta; \Psi \vdash \pi \Leftarrow \Phi \quad [\pi]_\Phi(A) = B}{\Delta \vdash \hat{\Psi}.p[\pi] \Leftarrow \#B[\Psi]}$$

Meta-Substitutions $\boxed{\Delta \vdash \theta \Leftarrow \Delta'}$

$$\frac{}{\Delta \vdash \cdot \Leftarrow \cdot} \qquad \frac{\Delta \vdash \theta \Leftarrow \Delta' \quad \Delta \vdash C \Leftarrow [\![\theta]\!]_{\Delta'}(U)}{\Delta \vdash \theta, C/X \Leftarrow \Delta', X{:}U}$$

Figure 2. Typing for meta-terms

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

$$\frac{y{:}T \in \Gamma}{\Delta; \Gamma \vdash y \Rightarrow T} \qquad \frac{\Delta; \Gamma \vdash I \Rightarrow T_2 \to T \qquad \Delta; \Gamma \vdash E \Leftarrow T_2}{\Delta; \Gamma \vdash I\ E \Rightarrow T}$$

$$\frac{\Delta; \Gamma \vdash I \Rightarrow \Pi X{:}U.T \qquad \Delta \vdash C \Leftarrow U}{\Delta; \Gamma \vdash I\ C \Rightarrow [\![C/X]\!]T} \qquad \frac{\Delta; \Gamma \vdash E \Leftarrow T}{\Delta; \Gamma \vdash (E : T) \Rightarrow T}$$

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

$$\frac{\Delta; \Gamma,\ f : T \vdash E \Leftarrow T}{\Delta; \Gamma \vdash \mathsf{rec}\ f.E \Leftarrow T} \qquad \frac{\Delta; \Gamma \vdash I \Rightarrow T \quad T = T'}{\Delta; \Gamma \vdash I \Leftarrow T'}$$

$$\frac{\Delta; \Gamma, y{:}T_1 \vdash E \Leftarrow T_2}{\Delta; \Gamma \vdash \mathsf{fn}\ y.E \Leftarrow T_1 \to T_2} \qquad \frac{\Delta, X{:}U; \Gamma \vdash E \Leftarrow T}{\Delta; \Gamma \vdash \lambda X.\, E \Leftarrow \Pi X{:}U.T}$$

$$\frac{\Delta \vdash C \Leftarrow U}{\Delta; \Gamma \vdash C \Leftarrow U} \qquad \frac{\Delta; \Gamma \vdash I \Rightarrow S \quad \text{for all } i\ \Delta; \Gamma \vdash B_i \Leftarrow S \to T}{\Delta; \Gamma \vdash \mathsf{case}\ I\ \mathsf{of}\ \vec{B} \Leftarrow T}$$

$\boxed{\Delta; \Gamma \vdash B \Leftarrow S \to T}$ Branch B with pattern of S checks against T

$$\frac{\Delta_i \vdash \theta_i \Leftarrow \Delta \quad \Delta_i \vdash C \Leftarrow [\![\theta_i]\!]U \quad \Delta_i; [\![\theta_i]\!]\Gamma \vdash E \Leftarrow [\![\theta_i]\!]T}{\Delta; \Gamma \vdash \Pi\Delta_i.\, C : \theta_i \mapsto E \Leftarrow U \to T}$$

Figure 3. Typing for computations

plying computation-level functions to an expression and the other to apply a dependent function to a meta-object C.

The type index objects are drawn from the domain of meta-objects presented in the previous section, but we emphasize that this language is parametric over the index domain, requiring only decidable equality to be able to compare two types.

The type language supports function types (written as $T_1 \to T_2$), dependent function types (written as $\Pi X{:}U.T$), and meta-types U. We only allow dependencies on meta-terms not on arbitrary computation-level expressions.

Our language is split into expressions for which we synthesize types and expressions which are checked against a type. This minimizes the necessary type annotations and provides a syntax directed recipe for a type checker. Intuitively, the expressions which introduce a type are expressions which are checked and expressions which eliminate a type are in the synthesis category.

Types	$T ::= U \mid T_1 \to T_2 \mid \Pi X{:}U.T$
Expressions (synth.)	$I ::= y \mid I\ E \mid I\ C \mid (E{:}T)$
Expressions (checked)	$E ::= I \mid C \mid \mathsf{fn}\ y.E \mid \lambda X.\, E \mid \mathsf{rec}\ f.E \mid \mathsf{case}\ I\ \mathsf{of}\ \vec{B}$
Branch	$B ::= \Pi\Delta.C : \theta \mapsto E$
Branches	$\vec{B} ::= \cdot \mid (B \mid \vec{B})$
Contexts	$\Gamma ::= \cdot \mid \Gamma, y{:}T$

We note that we can directly refer to meta-types and embed them in our computation-level types. Hence meta-objects can be directly analyzed and manipulated by our computation language.

Branches are modelled by $\Pi\Delta.C : \theta \mapsto E$ where Δ describes the meta-variables occurring in the pattern which are often left implicit in the surface language. The refinement substitution θ describes how the type of the scrutinee is instantiated so the given branch is applicable.

Next, we summarize the bi-directional typing rules for computations in Figure 3. We distinguish between typing of expressions and branches. In the typing judgment, we will distinguish between the context Δ for contextual variables from our index domain and the context Γ which includes declarations of computation-level variables. Contextual variables will be introduced via $\lambda X.E$. The contextual variables in Δ are also introduced in the branch of a case-expression. Computation-level variables in Γ are introduced by recursion or functions. We use the following judgments:

$\Delta; \Gamma \vdash E \Leftarrow T$	Expression E checks against type T
$\Delta; \Gamma \vdash I \Rightarrow T$	Expression I synthesizes type T
$\Delta; \Gamma \vdash B \Leftarrow S \to T$	Branch B with pattern of type S checks against T

We will tacitly rename bound variables, and maintain that contexts declare no variable more than once. Moreover, we require the usual conditions on bound variables. For example in the rule for λ-abstraction the contextual variable X must be new and cannot already occur in the context Δ. This can be always achieved via α-renaming. Similarly, in the rule for recursion and function abstraction, the variable x must be new.

4. Translation to Fresh-Style

In this section, we describe how to translate Beluga expressions into their corresponding simply typed Fresh-Style representation. The key challenge is to represent contextual objects and contexts. We begin with a review of the "*Fresh-Look* " approach by Pouillard and Pottier [2010] upon which our approach is based.

4.1 Fresh Binders

"Fresh-Style ". The main objective of the Fresh-Look approach is to be able to represent names in a sound way. We can informally describe "soundness" of names by the following three *informal slogans*:

- *"name abstractions cannot be violated"*; or: *"the representations of two α-equivalent terms cannot be distinguished"*;
- *"names do not escape their scope"*;
- *"names with different scopes cannot be mixed"*

It is important to note, that low-level representations offer no intrinsic protection from most forms of misuse of binders, such as binders escaping their scopes or name capture. In contrast, high level representations like HOAS and Fresh-Look satisfy at least some of these requirements. In this paper, we translate contextual LF objects which use HOAS encodings to a generic representation which uses the Fresh-Look idea. We call this intermediate representation "Fresh-Style ". The Fresh-Style representation makes it easy

to convert to names or de Bruijn indices. Moreover, it allows us to relate contextual object to Fresh-Look encodings thereby clarifying the relationship between HOAS and nominal encodings.

The Fresh-Look representation's central idea is that variables are represented by *names*. *Names* are abstract entities that we can compare for equality, and each name inhabits a world. Terms and variables using this technique are indexed by a world to indicate in which world they are meaningful. The key is that each term cannot mix in the same scope sub-terms or variables from different worlds. The last concept we need is the notion of a *link*. *Links* relate one world to another, and each link introduces a name in the destination world. Our work differs in two important aspects: First we support an abstract world (in addition to just an empty world) to support the idea of contextual variables. Second, the Fresh-Look approach features 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. In our compiler (i.e., Fresh-Style) we only use hard-links and drop the idea of soft-links, since this is enough for our purposes. 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. Before referring to a name, it must be imported to the bigger inner world, because we cannot mix names from different worlds in the same scope. Let us summarize the key concepts:

- *Names* are the abstract representation of variables. There is an infinite amount of them, and each name inhabits a world.
- *Worlds* contain names, and are to be related to other worlds by a link. In fact, all the worlds are related to an empty world by a chain of links.
- A *link* $\alpha \leftharpoonup \beta$ relates two worlds α and β and introduces a new name β. This is the key concept of the model because a new bigger world contains one extra name that represents a newly bound variables.

In its original presentation [Pouillard and Pottier 2010], the Fresh-Look representation focuses on presenting an implementation in the dependently-typed language Agda and proving its soundness. We will focus on using the representation as an intermediate language for relating contextual LF to nominal and de Bruijn representations. A key advantage of using the Fresh-Style representation is that it allows us to delay the choice of concrete implementation of binders, but it is also sufficiently low-level to be a good target for contextual objects and allows us to describe important phases such as pattern matching compilation and other optimizations generically, i.e. independent of the concrete representation chosen for variables.

We implemented the Fresh-Style binders in Ocaml. The main module shown below uses three abstract types `world`, `name` and `link` and defines appropriate functions to manipulate them.

```
val empty : 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
```

`empty` returns a world with no names in it; `fresh` returns the link to a bigger world when given a world; `name_of` returns the name it introduces to the bigger world when given a link. `import` allows us to bring all names from the outer world to the inner world.

The final two functions `name_to_db` and `name_to_name` allow us to extract de Bruijn indices and plain-old names from the abstract `names` in the Fresh-Style .

Neutral Terms $\boxed{\delta; \Psi_\alpha \vdash R_\alpha \Rightarrow a}$

$$\frac{\Sigma(\mathbf{c}) = a}{\delta; \Psi_\alpha \vdash \mathbf{c} \Rightarrow a} \qquad \frac{\delta; \Psi_\alpha \vdash R_\alpha \Rightarrow a \to b \quad \delta; \Psi_\alpha \vdash M_\alpha \Leftarrow a}{\delta; \Psi_\alpha \vdash R_\alpha\, M_\alpha \Rightarrow b}$$

$$\frac{\Psi_\alpha(n_\alpha) = a}{\delta; \Psi_\alpha \vdash n_\alpha \Rightarrow a} \qquad \frac{\delta(p) = \#a[\Phi_\gamma] \quad \delta; \Psi_\alpha \vdash {}^{\gamma}\vec{\sigma}^{\alpha} \Leftarrow \Phi_\gamma}{\delta; \Psi_\alpha \vdash p[{}^{\gamma}\vec{\sigma}^{\alpha}] \Rightarrow a}$$

$$\frac{\delta(p) = a[\Phi_\gamma] \quad \delta; \Psi_\alpha \vdash {}^{\gamma}\vec{\sigma}^{\alpha} \Leftarrow \Phi_\gamma}{\delta; \Psi_\alpha \vdash u[{}^{\gamma}\vec{\sigma}^{\alpha}] \Rightarrow a}$$

Normal Terms $\boxed{\delta; \Psi_\alpha \vdash M_\alpha \Leftarrow a}$

$$\frac{\delta; \Psi_\alpha \vdash R_\alpha \Rightarrow \mathbf{a}}{\delta; \Psi_\alpha \vdash R_\alpha \Leftarrow \mathbf{a}} \qquad \frac{\delta; \Psi_\alpha, \alpha \leftharpoonup \beta{:}a \vdash M_\beta \Leftarrow b}{\delta; \Psi_\alpha \vdash \lambda\alpha \leftharpoonup \beta.M_\beta \Leftarrow a \to b}$$

Substitutions $\boxed{\delta; \Psi_\gamma \vdash {}^{\alpha}\vec{\sigma}^{\gamma} \Leftarrow \Psi'_\alpha}$

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

$$\frac{\delta; \Psi_\beta \vdash \uparrow^{c,k} \Leftarrow \psi_{[\cdot]}}{\delta; \Psi_\beta, \beta \leftharpoonup \alpha : a \vdash \uparrow^{c,k+1} \Leftarrow \psi_{[\cdot]}} \qquad \frac{\delta; \Psi_\beta \vdash \uparrow^{c,k} \Leftarrow \cdot}{\delta; \Psi_\beta, \beta \leftharpoonup \alpha : a \vdash \uparrow^{c,k+1} \Leftarrow \cdot}$$

$$\frac{\delta; \Psi'_\gamma \vdash {}^{\beta}\vec{\sigma}^{\gamma} \Leftarrow \Psi_\beta \quad \delta; \Psi'_\gamma \vdash M_\gamma \Leftarrow a}{\delta; \Psi'_\gamma \vdash \underbrace{{}^{\beta}\vec{\sigma}^{\gamma}, M_\gamma}_{{}^{\alpha}\vec{\sigma}^{\gamma}} \Leftarrow \Psi_\beta, \beta \leftharpoonup \alpha{:}a}$$

$$\frac{\delta; \Psi'_\gamma \vdash {}^{\beta}\vec{\sigma}^{\gamma} \Leftarrow \Psi_\beta \quad \delta; \Psi'_\gamma \vdash H_\gamma \Rightarrow a}{\delta; \Psi'_\gamma \vdash \underbrace{{}^{\beta}\vec{\sigma}^{\gamma}; H_\gamma}_{{}^{\alpha}\vec{\sigma}^{\gamma}} \Leftarrow \Psi_\beta, \beta \leftharpoonup \alpha{:}a}$$

Figure 4. Typing rules for Fresh-Style contextual LF

4.2 Beluga using the Fresh-Style representation

The compilation process starts with erasing dependencies and translating a simply typed Beluga program. Since we know that the original Beluga program type checks, it would be possible to erase all type information, and not only dependencies, but the compiler keeps the simple types to allow us to type check each transformation of the program.

In the intermediate representation we use Fresh-Style variables for bound variables occurring in contextual objects. The resulting object is simply typed, i.e. if the original object has type A in the context Ψ, then the Fresh-Style object has the same type A in the Fresh-Style context Ψ_α where Ψ_α is obtained by translating the context Ψ. Below is the grammar for Fresh-Style contextual LF.

Types	a, b	::=	$\mathbf{a} \mid a \to b$
Heads	H_α	::=	$n_\alpha \mid \mathbf{c} \mid p[{}^{\alpha}\vec{\sigma}^{\beta}]$
Neutral Terms	R_α	::=	$H_\alpha \mid R_\alpha N_\alpha \mid u[{}^{\alpha}\vec{\sigma}^{\beta}]$
Normal Terms	M_α, N_α	::=	$R_\alpha \mid \lambda\alpha \leftharpoonup \beta.M_\beta$
Context Shifts	c	::=	$0 \mid \psi_{[\cdot]} \mid -\psi_{[\cdot]}$
Substitutions	${}^{\alpha}\vec{\sigma}^{\beta}$	::=	$\uparrow^{c,k} \mid {}^{\gamma}\vec{\sigma}^{\beta}, M_\beta \mid {}^{\gamma}\vec{\sigma}^{\beta}; H_\beta$
Contexts	Ψ_α	::=	$\cdot \mid \psi_\alpha \mid \Psi_\gamma, \gamma \leftharpoonup \alpha{:}a$

The most important aspect of this representation is that terms are annotated with the world in which they are meaningful. Terms must be "well-worlded", i.e. applications must involve terms of the same world, and abstractions living in world α must include a link from that world to an inner bigger world (e.g., $\alpha \leftharpoonup \beta$) and the body of

Meta Terms $\boxed{\delta \vdash \kappa \Leftarrow \upsilon}$

$$\frac{\delta; \Psi_\alpha \vdash R_\alpha \Leftarrow p}{\delta \vdash \hat{\Psi}_\alpha.R_\alpha \Leftarrow p[\Psi_\alpha]} \qquad \frac{\Psi_\alpha(n_\alpha) = a}{\delta \vdash \hat{\Psi}_\alpha.n_\alpha \Leftarrow \#a[\Psi_\alpha]}$$

$$\frac{\delta(p) = \#a[\Phi_\beta] \quad {}^{\beta\to\alpha}\pi \text{ is a pattern subst.} \quad \delta; \Psi_\alpha \vdash {}^{\beta\to\alpha}\pi \Leftarrow \Phi_\beta}{\delta \vdash \hat{\Psi}_\alpha.p[{}^{\beta\to\alpha}\pi] \Leftarrow \#a[\Psi_\alpha]}$$

$$\frac{a \in g \quad \delta \vdash \Psi_\gamma \Leftarrow g}{\delta \vdash \Psi_\gamma, \gamma \leftharpoonup \alpha{:}a \Leftarrow g} \qquad \frac{\delta(\psi) = g}{\delta \vdash \psi_{[\cdot]} \Leftarrow g} \qquad \frac{}{\delta \vdash \cdot \Leftarrow g}$$

Figure 5. Typing rules for simple meta terms and types

the abstraction must live in the inner world (e.g., world β). The typing rules (see Fig. 4) hence not only enforce that the given objects are well-typed but in addition that they are "well-worlded".

We again lift fresh-style LF objects to fresh-style meta-types and meta-objects (see Fig. 5). For the subsequent description, it is convenient to distinguish between meta types which stand for contextual LF types, described by w, and context schemas g.

Context schemas	g	::=	$\mathbf{a} \mid \mathbf{a} + g$
Meta Terms	κ	::=	$\hat{\Psi}_\alpha.R_\alpha \mid \Psi_\alpha$
Cont. LF Types	w	::=	$a[\Psi_\alpha] \mid \#a[\Psi_\alpha]$
Meta Types	υ	::=	$w \mid g$
Meta substitutions	ρ	::=	$\cdot \mid \rho, \kappa$
Meta-context	δ	::=	$\cdot \mid \delta, X{:}\upsilon$

As only bound variables in contextual objects use the Fresh-Style representation, the computational language is not very different from before, except the language is now simply typed. While we drop the dependencies on contextual LF objects, we retain the dependency on contexts. The dependent function space $\Pi X{:}U.T$ is hence split up into $w \overset{\square}{\to} \tau$ and $\Pi\phi_{[\cdot]}{:}g.\tau$ (see also the translation given in Sec. 4.3).

Types	τ	::=	$\tau_1 \to \tau_2 \mid w \overset{\square}{\to} \tau \mid \Pi\phi_{[\cdot]}{:}g.\tau \mid \upsilon$
Expr. (synth.)	i	::=	$y \mid i\, e \mid i\ \kappa \mid (e{:}\tau)$
Expr. (checked)	e	::=	$i \mid \kappa \mid \mathsf{fn}\ y.e \mid \lambda X.\, e \mid$ $\mathsf{rec}\ f.e \mid \mathsf{case}\ i\ \mathsf{of}\ \vec{\zeta}$
Branch	ζ	::=	$\Pi\delta.\kappa : \rho \mapsto e$
Branches	$\vec{\zeta}$	::=	$\cdot \mid (\zeta \mid \vec{\zeta})$
Contexts	γ	::=	$\cdot \mid \gamma, y{:}\tau$

The resulting typing rules for computations (see Fig. 6) are similar to the original rules for Beluga programs, however the rule for checking branches needs some explanation. As before, ρ_i is a refinement substitution which refines the type indices meta-types. Since we only deal with simple LF types and we have erased all indices, it may seem superflous to still keep ρ_i and even apply it to the context γ and the simple computation type τ - all of which are simply typed. The reason this is necessary is that ρ carries instantiations for context variables - and dependencies on context variables still persist. By matching on the shape of the context, we may learn that the given context must be empty for example; this knowledge must be propagated.

4.3 Translating Beluga to Fresh-Style Binders

As mentioned earlier, our intermediate representation is simply typed. Thus, we drop type dependencies when translating Beluga programs to their corresponding Fresh-Style representation. Erasing dependencies is common when compiling dependently typed programs as it is not clear how to scale existing type-preserving

$\boxed{\delta; \gamma \vdash i \Rightarrow \tau}$ Expression i synthesizes type τ

$$\frac{y{:}\tau \in \gamma}{\delta; \gamma \vdash y \Rightarrow \tau} \qquad \frac{\delta; \gamma \vdash i \Rightarrow \tau_2 \to \tau \quad \delta; \gamma \vdash e \Leftarrow \tau_2}{\delta; \gamma \vdash i\, e \Rightarrow \tau}$$

$$\frac{\delta; \gamma \vdash e \Leftarrow \tau}{\delta; \gamma \vdash (e : \tau) \Rightarrow \tau} \qquad \frac{\delta; \gamma \vdash i \Rightarrow w \overset{\square}{\to} \tau \quad \delta \vdash \kappa \Leftarrow q}{\delta; \gamma \vdash i\ \kappa \Rightarrow \tau}$$

$$\frac{\delta; \gamma \vdash i \Rightarrow \Pi\phi_{[\cdot]}{:}g.\tau \quad \delta \vdash \Psi_\alpha \Leftarrow g}{\delta; \gamma \vdash i\ \Psi_\alpha \Rightarrow [\![\Psi_\alpha/\phi_{[\cdot]}]\!]\tau}$$

$\boxed{\delta; \gamma \vdash e \Leftarrow \tau}$ Expression e checks against type τ

$$\frac{\delta; \gamma \vdash i \Rightarrow \tau}{\delta; \gamma \vdash i \Leftarrow \tau} \quad \frac{\delta \vdash \kappa \Leftarrow \upsilon}{\delta; \gamma \vdash \kappa \Leftarrow \upsilon} \quad \frac{\delta; \gamma, y{:}\tau_1 \vdash E \Leftarrow \tau_2}{\delta; \gamma \vdash \mathsf{fn}\ y.e \Leftarrow \tau_1 \to \tau_2}$$

$$\frac{\delta, X{:}w; \gamma \vdash e \Leftarrow \tau}{\delta; \gamma \vdash \lambda X.\, e \Leftarrow w \overset{\square}{\to} \tau} \qquad \frac{\delta, X{:}g; \gamma \vdash e \Leftarrow \tau}{\delta; \gamma \vdash \lambda X.\, e \Leftarrow \Pi\phi{:}g.\tau}$$

$$\frac{\delta; \gamma \vdash i \Rightarrow w \quad \text{for all } k,\ \delta; \gamma \vdash \zeta_k \Leftarrow w \to \tau}{\delta; \gamma \vdash \mathsf{case}\ i\ \mathsf{of}\ \zeta_1 \mid \ldots \mid \zeta_n \Leftarrow \tau}$$

$$\frac{\delta; \gamma,\ f : \tau \vdash e \Leftarrow \tau}{\delta; \gamma \vdash \mathsf{rec}\ f.e \Leftarrow \tau}$$

$\boxed{\delta; \gamma \vdash \zeta \Leftarrow w \to \tau}$ Branch ζ with pattern of type w checks against τ

$$\frac{\delta_i \vdash \kappa \Leftarrow [\![\rho_i]\!]_\delta(w) \quad \delta_i \vdash \rho_i \Leftarrow \delta \quad \delta_i; [\![\rho_i]\!]_\delta(\gamma) \vdash e \Leftarrow [\![\rho_i]\!]_\delta(\tau)}{\delta; \gamma \vdash \Pi\delta_i.\kappa : \rho_i \mapsto e \Leftarrow w \to \tau}$$

Figure 6. Typing rules for simple Fresh-Style computations

compilation techniques (such as Morrisett et al. [1999], Guillemette and Monnier [2008], and Monnier and Haguenauer [2010]) to dependent types. Dependency erasure is also commonly used in proof assistants that extract functional programs from dependently typed specifications. One salient example is *extraction* in Coq described by Letouzey [2003] and Letouzey [2008] and used to implement CompCert a certified C compiler as described by Leroy [2009].

Translating LF types We begin by translating contextual LF objects and contextual LF types. The dependency erasure, written as $()^-$, follows the LF literature [Harper and Pfenning 2005].

$$\begin{array}{lrcl} \text{Types Erasure} & (\Pi A_1.\, A_2)^- & = & (A_1)^- \to (A_2)^- \\ & (\mathbf{a} \cdot S)^- & = & \mathbf{a} \end{array}$$

Translating LF contexts Translating a context Ψ to its corresponding representation Ψ_α in Fresh-Style is slightly more complicated, since we need to recursively add links for each declaration starting from the empty world.

$$\begin{array}{lrcl} \mathsf{transCtx} & \cdot & = & \cdot \\ \mathsf{transCtx} & \psi & = & \psi_{[\cdot]} \\ \mathsf{transCtx} & (\Psi, A) & = & \mathsf{let}\ \Psi'_\beta = \mathsf{transCtx}\ \Psi\ \mathsf{in} \\ & & & (\Psi'_\beta, \mathsf{fresh}\ \beta{:}(A)^-) \end{array}$$

Translating LF objects The translation of LF terms to their Fresh-Style representation is done using two mutual recursive functions, `trans` and `transSub`, that respectively translate contextual LF terms and LF substitutions. We rely on two auxiliary functions: fresh generates a new world β and a link $\alpha \leftharpoonup \beta$ when given a world α and function $\mathsf{name\_of}(\alpha \leftharpoonup \beta)$ returns the fresh name introduced in world β. Furthermore, $\Psi_\alpha(x)$ retrieves the name of

the x-th declaration in Ψ_α.

$$
\begin{array}{llllll}
\mathsf{trans} & \Psi_\alpha & \lambda.M & = & \lambda(\mathsf{fresh}\ \alpha). & \\
 & & & & \mathsf{trans}\ (\Psi_\alpha, \mathsf{fresh}\ \alpha)(M)^- & \\
\mathsf{trans} & \Psi_\alpha & (R\ N) & = & (\mathsf{trans}\ \Psi_\alpha\ R)\ (\mathsf{trans}\ \Psi_\alpha\ N) & \\
\mathsf{trans} & \Psi_\alpha & x & = & \mathsf{name\_of}(\Psi_\alpha(x)) & \\
\mathsf{trans} & \Psi_\alpha & u[\sigma] & = & u[\overset{\gamma\to\alpha}{\sigma_1}] \text{ where} & \\
 & & & & \mathsf{transSub}\ \Psi_\alpha\ \sigma = (\overset{\gamma\to\alpha}{\sigma_1}, \_) & \\
\mathsf{trans} & \Psi_\alpha & p[\sigma] & = & p[\overset{\gamma\to\alpha}{\sigma_1}] \text{ where} & \\
 & & & & \mathsf{transSub}\ \Psi_\alpha\ \sigma = (\overset{\gamma\to\alpha}{\sigma_1}, \_) &
\end{array}
$$

The `trans` function is straight-forward and the only interesting detail is how the body of a λ-abstraction is translated inside the bigger context.

Translating 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 $\mathsf{transSub}$ takes the context Φ_α in the range world and a substitution σ and returns the context Φ_β in the domain world together with the translated substitution σ' s.t. $\Phi_\beta \vdash \sigma' \Leftarrow \Phi_\alpha$.

$$
\begin{array}{llllll}
\mathsf{transSub} & \Psi_\alpha & \uparrow^{0,0} & = & (\uparrow^{0,0}, \Psi_\alpha) & \\
\mathsf{transSub} & \Psi_\alpha & \uparrow^{0,k} & = & (\uparrow^{0,k}, \Psi_\delta) \text{ where} & \\
 & & & & (\_, \Psi_\delta) = & \\
 & & & & \mathsf{transSub}(\Psi_\alpha,\ \mathsf{fresh}\ \alpha) \uparrow^{0,k-1} & \\
\mathsf{transSub} & \psi_{[\cdot]} & \uparrow^{\psi,0} & = & (\uparrow^{\psi_{[\cdot]},0}, \cdot) & \\
\mathsf{transSub} & \cdot & \uparrow^{-\psi,0} & = & (\uparrow^{-\psi_{[\cdot]},0}, \psi_{[\cdot]}) & \\
\mathsf{transSub} & \psi_{[\cdot]} & \uparrow^{\psi,k} & = & (\uparrow^{\psi_{[\cdot]},k}, \Psi_0) \text{ where } |\Psi_0| = k & \\
\mathsf{transSub} & \cdot & \uparrow^{-\psi,k} & = & (\uparrow^{-\psi_{[\cdot]},k}, \psi_{[\cdot]}, \Psi_0) \text{ where} & \\
 & & & & |\Psi_0| = k & \\
\mathsf{transSub} & \Psi_\alpha & (\sigma, M) & = & ((\overset{\gamma\to\alpha}{\sigma'}, \mathsf{trans}\ \Psi_\alpha M), & \\
 & & & & (\Psi'_\gamma, \mathsf{fresh}\ \gamma)) \text{ where} & \\
 & & & & \mathsf{transSub}\ \Psi_\alpha\ \sigma\ = (\overset{\gamma\to\alpha}{\sigma'}, \Psi'_\gamma) & \\
\mathsf{transSub} & \Psi_\alpha & (\sigma; H) & = & ((\overset{\gamma\to\alpha}{\sigma'}, \mathsf{trans}\ \Psi_\alpha H); & \\
 & & & & (\Psi'_\gamma, \mathsf{fresh}\ \gamma)) \text{ where} & \\
 & & & & \mathsf{transSub}\ \Psi_\alpha\ \sigma\ = (\overset{\gamma\to\alpha}{\sigma'}, \Psi'_\gamma) &
\end{array}
$$

To translate a meta-object $[\Psi.R]$ we simply translate each part: $\mathsf{transCtx}\Psi\ =\ \Psi_\alpha$ and $\mathsf{trans}\ \Psi_\alpha\ R\ =\ R_\alpha$. However, this translation may lose information. Consider for example the objects `[g, x:exp i. #p ..]` and `[g, x:exp (arr i i).#p..]`. After the translation, both are mapped to `[g, x:exp. #p ..]`. This means while the original program was unambiguous, it is now no longer. Dependency erasure only works if type indices are irrelevant for the execution.

We omit the erasure and translation to meta-types and computation-level types which is a straightforward extension.

5. Pattern matching compilation

During pattern matching compilation we compute splitting trees recursively [Fessant and Maranget 2001; Maranget 2008]. To support the rich pattern matching of contextual objects we separate how the tree is generated at compile-time from 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 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. Our compilation schema supports using de Bruijn indices and unique names. A key element of our algorithm is that compilation is performed on contextual values using the Fresh-Style representation (without committing yet on a concrete representation for contextual objects), and the comparison of terms and patterns is deferred to the runtime when the internal representation of contextual values has already been made.

Pattern matching in Beluga can be seen as the regular ML-style pattern matching extended with support for contextual objects, i.e. support for matching against the shape of contexts, inside binders, and against specific variables in the context.

In Beluga, pattern matching is performed by the **case** expression of the computational language. **case** provides a term that is 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, in general, is 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. However, depending on the expression and the exact technique it may produce an exponential blowup of the size of program. The algorithm for computing the discrimination tree is also far from trivial, and different choices result in different performance and memory consumption. For our compiler we derive our technique from Maranget [2008] who uses a technique using trees for high-performance and some heuristics to try to minimize the size of the tree.

5.1 The Target Language for Pattern Matching Compilation

When computing the splitting tree, nested pattern matching is replaced by simple non-nested discrimination statements. In our case we replace **case** expressions with `switch` and `ctx_switch` expressions. These simpler expressions are lower level because they do not contain recursive patterns.

We assume that the given source program type checks and is covering (see [Dunfield and Pientka 2009; Schürmann and Pfenning 2003]). From the patterns in the source program, we build a splitting tree. We restrict patterns to be at most second-order, i.e. we do not allow bound variable or parameter variable occurrences of function type. This simplifies our description.

$$
\begin{array}{llll}
\text{Expr. Checked} & e & ::= & \ldots \mid \mathsf{switch}\ i\ \mathsf{of}\ \vec{S} \\
 & & & \mid \mathsf{ctx\_switch}\ i\ \mathsf{of}\ \vec{\zeta} \\
\text{Ctx. Switch Branches} & \vec{\zeta} & ::= & \cdot \mid (\zeta|\vec{\zeta}) \\
\text{Ctx. Switch Branch} & \zeta & ::= & \Pi\delta.[\![\hat{\Psi}_{\hat{\alpha}}]\!] : \rho \mapsto e \\
\text{Switch Branches} & \vec{S} & ::= & \cdot \mid (S|\vec{S}) \\
\text{Switch Branch} & S & ::= & \Pi\delta.[\![\hat{\Psi}_\alpha.Q_\alpha]\!] \mapsto e \\
\text{Simple Pattern} & Q_\alpha & ::= & \lambda\alpha \leftharpoonup \beta.V_\beta \mid \mathbf{c}\ \vec{V_\alpha} \mid \mathsf{n}_\alpha \mid \\
 & & & u[\overset{\gamma\to\alpha}{\sigma}] \mid p[\overset{\gamma\to\alpha}{\pi}] \\
\text{Pattern Variable} & V_\alpha & ::= & u[\uparrow^{0,0}] \mid p[\uparrow^{0,0}]
\end{array}
$$

Pattern variables in our compiled splitting tree are denoted by V_α; they are either meta-variables or parameter variables which are fully applied, i.e. they can depend on every variable in the context. Since we use explicit substitutions, we associate the identity substitution (i.e. $\uparrow^{(0,0)}$) with a fully applied meta-variable or parameter variable. This ensures that variable dependency checks are only carried out at the leafs of the splitting tree. Pattern variables, unlike meta-variables occurring in source level patterns, are not necessarily of atomic type. We disentangle the match of an argument to a constant and the further refinement that this argument stands for a lambda-abstraction. For example, matching on `lam λx. E .. x` is

Context Switch:

$$\frac{\delta;\gamma \vdash i \Rightarrow \mathbf{a}[\Psi_\alpha] \quad \text{for all } k\ \delta;\gamma \vdash \zeta_k \Leftarrow \tau}{\delta;\gamma \vdash \mathsf{ctx\_switch}\ i\ \mathsf{of}\ \zeta_1 | \ldots | \zeta_n \Leftarrow \tau}$$

Ctx. Switch Branches:

$$\frac{\delta';[\![\rho]\!]\gamma \vdash e \Leftarrow [\![\rho]\!]\tau}{\delta;\gamma \vdash \Pi\delta'.[\![\hat{\Psi}_{\hat{\alpha}}]\!] : \rho \mapsto e \Leftarrow \tau}$$

General Switch:

$$\frac{\delta;\gamma \vdash i \Rightarrow \mathbf{a}[\Psi_\alpha] \quad \text{for all } k\ \delta;\gamma \vdash S_k \Leftarrow \mathbf{a}[\Psi_\alpha] \to \tau}{\delta;\gamma \vdash \mathsf{switch}\ i\ \mathsf{of}\ S_1 | \ldots | S_n \Leftarrow \tau}$$

Switch Branches:

$$\frac{\delta';\Psi_\alpha \vdash Q_\alpha \Leftarrow \mathbf{a} \quad \delta,\delta';\gamma \vdash e \Leftarrow \tau}{\delta;\gamma \vdash \Pi\delta'.[\![\hat{\Psi}_\alpha.Q_\alpha]\!] \mapsto e \Leftarrow \mathbf{a}[\Psi_\alpha] \to \tau}$$

Figure 7. Typing rules for switch expressions

done in two steps; first we consider the simple pattern `lam` $\mathtt{V}_\alpha$ and then we consider the simple pattern $\lambda\alpha \leftharpoonup \beta.V'_\beta$.

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 contextual variables $\psi_{[\cdot]}$ and the contextual substitution ρ provides an instantiation for the context variable ψ. On the other hand, the `switch` expression splits on all the other patterns. To keep `switch` low-level it does not support nested patterns.

5.2 Typing rules for the expanded language

The typing rules for the extensions (`switch` and `ctx_switch`) are given in Fig.7. The general switch rule is derived from the rule for the case expression. We infer the type of the scrutinee and then verify for each branch that the pattern has the same type as the scrutinee and check that the body of the branch has the same overall type as the case expression. We simply extend δ with the additional meta-variables described by δ' occurring in the pattern. Note that we have no refinement substitution ρ associated to switch branches, since we are working in the simply typed setting. Intuitively, the only refinement allowed is the one on context variables; this refinement is taken care of in the context switch rule. When type checking a context switch statement against a type τ, we ensure that the scrutinee is well-typed and check that each branch has type τ; for each branch we verify that the body has type $[\![\rho_i]\!]\tau$ in the context $[\![\rho_i]\!]\gamma$ where ρ_i is the refinement substitution in the i-th branch.

Our typing rules for simple patterns are special instances of the typing rules for Fresh-Style LF where all pattern variables are fully applied, i.e. they are associated with the identity substitution. As a consequence, all bound variable checks are pushed to the leafs.

5.3 Compiling Beluga's Patterns

5.3.1 Discriminating the shape of contexts

In Beluga each branch of a case statement starts with a context we 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 part of a pattern, and that they are not nested, so when compiling patterns, the splitting tree begins by discriminating on the contexts and then continues by recursively splitting on the rest of the pattern. The contextual discrimination is done by adding a ctx_switch expression that groups the branches with patterns that describe contexts of the same shape. Because there is no nesting this is done first and only once.

After we split on the shape of contexts, each branch is not necessarily mutually exclusive. For example, in the function `hoas2db` from Section 2.1 we distinguish cases based on contexts with at least one variable(i.e. `[g, x:exp T]`) and those with an arbitrary context (i.e. `[g]`). Hence, ctx_switch has two cases: `[g,x:exp]` and `[g]`. When matching against a term `[x:exp. lam λy. app y x]`, we will first match the context `[x:exp]` against `[g, x:exp]`. However, pursuing this branch further will subsequently fail, because the only patterns with context `[g,x:exp]` were variable cases. Since we assume that our patterns in the original program are covering, we know that we must continue to match `[x:exp]` against the second branch of ctx_switch where we have `[g]`. In general, the `ctx_switch` expression handles the pattern matching failure and continues the matching with the next `ctx_switch` branch.

5.3.2 Matching the terms of contextual objects

For the rest of the patterns the situation is similar to other languages [Maranget 2008]. The first step is building a clause matrix with all the patterns and the bodies of the branches in the case expression.

$$\vec{\mathcal{Q}} \to \mathcal{E} = \begin{pmatrix} Q_{1,1} & \cdots & Q_{1,n} & \to & e_1 \\ Q_{2,1} & \cdots & Q_{2,n} & \to & e_2 \\ \vdots & \ddots & \vdots & & \vdots \\ Q_{m,1} & \cdots & Q_{m,n} & \to & e_m \end{pmatrix}$$

We write $\mathcal{Q}$ for the matrix containing patterns and $\mathcal{E}$ for the column containing the bodies of the branches $e_1, \ldots e_m$. Initially $\vec{\mathcal{Q}} \to \mathcal{E}$ has only one pattern column, but as the splitting proceeds there could be more than one. We define two decomposition operations on a matrix using a tag q to facilitate the decomposition where $q \in \{\text{Const}(\mathbf{c}_1), \ldots, \text{Const}(\mathbf{c}_k), \text{Var}(p), \text{Var}(u_1), \ldots, \text{Var}(u_k)\}$: $S(q, \vec{\mathcal{Q}} \to \mathcal{E})$ *specializes* the matrix with respect to q by selecting only those patterns relevant to q. The new matrix contains only the rows whose first pattern $(Q_{i,1})$ is compatible with the tag q. Furthermore, it splits the first column according to the arity of the tag q. $D(\vec{\mathcal{Q}})$ is a *default* matrix. It retains all the rows of $\vec{\mathcal{Q}}$ whose first pattern admits all values that do not match any pattern in the first column. We then build the tree by branching on each of the specialized matrices and the default matrix for the rest of the cases.

We define both operations following Maranget [2008]. We define the compilation scheme $CC(\vec{M}, \vec{\mathcal{Q}} \to \mathcal{E})$ where $\vec{M}$ are the LF objects we pattern match on, and $\vec{\mathcal{Q}} \to \mathcal{E}$ is the clause matrix.

1. If the matrix $\vec{\mathcal{Q}}$ has no rows pattern matching fails.

$$CC(\vec{M}, \emptyset \to \mathcal{E}) \stackrel{def}{=} \mathsf{fail}$$

2. Otherwise, $\vec{\mathcal{Q}}$ is not empty. If $\vec{\mathcal{Q}}$ contains more than one column there is a significant choice to make, because splitting can proceed in any column, different choices lead to different performance and size of code, for simplicity in this description we will always split on the first column.

   (a) We calculate Σ_1 as the set of tags of the first column of $\vec{\mathcal{Q}}$.

   (b) We calculate:

$$E_k \stackrel{def}{=} CC((V_1, \ldots, V_n), S(q_k, \vec{\mathcal{Q}} \to \mathcal{E}))$$

   where $V_1, \ldots, V_n$ are obtained by splitting the term in the first column according to q_k. Intuitively, if q_k is Const(**c**) and **c** takes n arguments, then $V_1, \ldots, V_n$ stand for those arguments and are generated based on the type of **c**. We do

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

$$E_d \stackrel{def}{=} CC((M_2 \dots M_n), D(\vec{\mathcal{Q}} \to \mathcal{E}))$$

(c) We create a switch expression in the decision tree by associating with each $\mathbf{c}_i$ its corresponding action E_i and the default case with the action E_d.

Pattern matching compilation continues recursively with the remaining patterns.

In pattern languages with only constructors and variables, the specialization operation simply matches same constructors. To support contextual objects we define an extended specialization operation in the following way:

- Constructors: group together all the constructors of the same name.
- Bound variables from the context: group bound variables 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 (sometimes called *pronominal* variables [Licata and Harper 2009]).
- Parameter variables: group parameter variables with the same substitution.This is justified because different substitutions describe different behavior.
- Meta-variables: this case is similar to parameter variables. We group meta-variables with the same substitution.
- Lambda patterns: all lambdas in the head of a pattern are equivalent and grouped together in specialization. The name of the bound variable is irrelevant because comparison of names is pronominal [Licata and Harper 2009].

In conclusion, the pattern matching compilation contains three elements: 1) the language extension containing the simple patterns and `ctx_switch`, switch expressions, 2) the compilation algorithm with the parameterized specialization and 3) the new type-checking rules to be able to check the result of the compilation with the extended language.

5.4 Matching at Runtime

At runtime, there are two relevant matching operations: Context matching, $\Psi_\alpha \overset{ctx}{\approx} \Phi_\beta$, tests whether Φ_β is an instance of Ψ_α. Term matching, $\hat{\Psi}_\alpha.Q_\alpha \overset{M}{\approx} \hat{\Phi}_\beta.Q'_\beta$, succeeds if Q'_β is an instance of Q_α. We only define sucess and failure of matching operations; the instantiations for pattern variables is straightforward from the given rules, but not handled here to not clutter the presentation. We also note that the context $\hat{\Psi}_\alpha$ on the left and the context $\hat{\Phi}_\beta$ on the right hand side of $\overset{M}{\approx}$ are not syntactically the same. While we will have matched Φ_β against Ψ_α and this context match must have succeed prior to calling term matching, the contexts may differ in the names of the links and worlds employed.

Context Matching:

$$\frac{}{\cdot \overset{ctx}{\approx} \cdot} \qquad \frac{}{\psi_{[\cdot]} \overset{ctx}{\approx} \Psi_\beta} \qquad \frac{\Psi_{\alpha_1} \overset{ctx}{\approx} \Phi_{\beta_1}}{\Psi_{\alpha_1}, \alpha_1 \leftharpoonup \alpha_2 : a \overset{ctx}{\approx} \Phi_{\beta_1}, \beta_1 \leftharpoonup \beta_2 : a}$$

$\overset{ctx}{\approx}$ is inductively defined on the structure of contexts. A context variable matches any context. When we match $\Phi_{\beta_1}, \beta_1 \leftharpoonup \beta_2 : A$ against $\Psi_{\alpha_1}, \alpha_1 \leftharpoonup \alpha_2 : A$, there is no requirement that the worlds and links in each of the contexts are the same.

$\overset{M}{\approx}$ is not inductively defined on the pattern because we are dealing with simple patterns which do not allow nesting. This operation compares simple patterns (Q_α) to a contextual LF object and is defined as follows

Term Matching:

$$\frac{}{\hat{\Psi}_\alpha.\lambda\alpha \leftharpoonup \beta.V_\beta \overset{M}{\approx} \hat{\Phi}_\gamma.\lambda\gamma \leftharpoonup \delta.\, M_\delta} \qquad \frac{\mathsf{c} = \mathsf{c}'}{\hat{\Psi}_\alpha.\mathsf{c}\ \vec{V_\alpha} \overset{M}{\approx} \hat{\Phi}_\gamma.\mathsf{c}' \cdot \vec{M_\gamma}}$$

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

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

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

There are five different cases to consider. A pattern and a term match when:

- The head of the term and the pattern are the same constructor. The instantiation we return is $V^i_\alpha = M^i_\gamma$, i.e. the i-th variable in $\vec{V_\alpha}$ is instantiated with the i-th term in $\vec{M_\gamma}$.
- The head and the pattern are λ-expressions. In this case we instantiate V_β with M_δ.
- The pattern is a meta-variable. In this case we ensure that applying the inverse substitution to the right hand side exists. Applying the inverse substitution is an operation defined by Dowek et al. [1996] and Abel and Pientka [2011]. It guarantees that the right hand side does not contain more free variables than allowed by the substitution associated with u. In this case, we instantiate u with $[\overset{\alpha \to \gamma}{\sigma}]^{-1}(M_\gamma)$.
- The pattern is a parameter variable, the inverse substitution is defined as in the meta-variable case, to ensure the variable dependencies imposed by $p[\overset{\alpha \to \gamma}{\pi}]$ are observed. In this case, we instantiate p with $[\overset{\alpha \to \gamma}{\pi}]^{-1}(n_\gamma)$.
- The pattern and the head of the term are LF variables declared in the same position in their contexts.

5.5 Names and Indices from Fresh-Style Variables

We use Fresh-Style bound variables because it is easy to generate the pattern matching tree and then convert to two representations without having to recalculate the tree. Additionally, many optimizations can be written using this representation; this fact simplifies supporting several back-ends each with a different binding strategy, as it minimizes code duplication.

However, for code generation we need to choose at compile-time a concrete representation of binders, in our case between plain-old names and de Bruijn indices.

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 $\alpha \leftharpoonup \beta$ 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).

5.6 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 as the resulting program is a form of the simply-typed λ-calculus.

In Section 5.4, we presented a general matching algorithm for the Fresh-Style language after pattern matching compilation. However, using this general algorithm during run-time is not necessarily efficient. We hence refine the matching algorithm subsequently for Fresh-Style representation using names and de Bruijn .

5.6.1 Matching with de Bruijn Indices

In the de Bruijn representation, contexts can be represented as a pair of a context variable and a number describing the number of concrete declarations. We write $(\cdot, k)$ for a context containing k declarations and (ψ, k) for a context which starts with a context variable ψ and is followed by k declarations. We use c to stand for either a context variable ψ or an empty context. For easier readability we write Ψ for a context whose concrete representation is (c, k). The matching operations are then defined as follows:

Context Matching:

$$\frac{}{(\cdot,k) \overset{ctx}{\approx} (\cdot,k)} \qquad \frac{}{(\psi,0) \overset{ctx}{\approx} (\cdot,k)} \qquad \frac{(c,n-1) \overset{ctx}{\approx} (\cdot,k-1)}{(c,n) \overset{ctx}{\approx} (\cdot,k)}$$

Term Matching:

$$\frac{}{\Psi.\lambda.V \overset{M}{\approx} \Phi.\lambda.M} \qquad \frac{\mathbf{c} = \mathbf{c}'}{\Psi.\mathbf{c}\ \vec{V} \overset{M}{\approx} \Phi.\mathbf{c}'\vec{M}} \qquad \frac{[\sigma]^{-1}M}{\Psi.u[\sigma] \overset{M}{\approx} \Phi.M}$$

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

Context matching succeeds as long as 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 only the lengths of the contexts are compared.

5.6.2 Matching with names

In the name representation, we model a context using a list of concrete names. The matching operations with names are similar to previous definitions of the function but account for the different structure of contexts and variable binders.

Context Matching:

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

Term Matching:

$$\frac{}{\Psi.\lambda\, x.V \overset{M}{\approx} \Phi.\lambda\, y.M} \qquad \frac{\mathbf{c} \overset{M}{\approx} \mathbf{c}'}{\Psi.\mathbf{c}\ \vec{V} \overset{M}{\approx} \Phi.\mathbf{c}'\vec{M}} \qquad \frac{[\sigma]^{-1}M}{\Psi.u[\sigma] \overset{M}{\approx} \Phi.M}$$

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

6. Related Work

Pouillard and Pottier [2010] present a high-level, abstract representation for names and contexts and then show how this abstract interface can simply be instantiate with a choice of various concrete representations. We reuse a lot of their work but start from an even higher-level representation.

Urban [2008] presents an Isabelle package that lets the user write proofs in terms of an abstract notion of names, which are under the hood represented by plain first order data. Contrary to our work he does not worry about efficiency and instead focuses on proving that the lower-level representation actually provides the expected semantic properties.

Along the same lines, Felty and Momigliano [2012] provide a package that lets users use HOAS in their specifications and proofs, and where the properties of those binders are proved by relating them to some lower-level first-order representation. But here as well, the focus is on writing proofs rather than programs, so efficiency of the code manipulated is a secondary concern.

Shinwell et al. [2003]; Washburn and Weirich [2008]; Westbrook et al. [2011] present various ways to provide support for programming with data-structures that contain binders within existing languages making them more practical than our work, but at some significant costs: they do not provide capture-avoiding substitution and do not support dependent types, for example. As the interface and the implementation are tightly integrated, it is more difficult to change the implementation strategy.

Chlipala [2008] shows a similar, though more light-weight, effort but within the context of a proof assistant. It can support dependent types, but the concrete representation of binders cannot be changed.

7. Conclusion

We have presented a generic framework for compiling contextual objects which can be instantiated to yield compiled code using names and code using de Bruijn indices. This opens up the possibility to choose the best representation depending on how binders are used. A key aspect of our framework is the pattern matching compilation for contextual objects which allows matching under the λ-binder. Our framework also provides for the first time a connection between programming with HOAS as done in Beluga and programming with nominal techniques. We believe our work is the first step in adding HOAS and contextual objects to existing programming languages and making this sophisticated technology available to the ordinary programmer.

In the future, we plan to prove that our compilation scheme is faithful to the theory, so that the meta-theoretical results developed for the theoretical foundation of Beluga carry over to the code generated by our compiler. Furthermore, we plan to explore the relation between the splitting trees used in pattern matching and those generated during coverage checking [Dunfield and Pientka 2009; Schürmann and Pfenning 2003].

We also plan to explore alternative compilation schemes such as targeting a functional language that supports pattern matching (e.g. OCaml). This can be done by translating contextual objects to linear higher-order patterns [Pientka and Pfenning 2003] and reusing the pattern matching algorithm of the target language together with guards enforcing higher-order constraints. This would decouple the aspects of compiling patterns with binders and efficiently compiling simple patterns.

Finally, we aim to improve the concrete low-level representations with the optimizations used in typical hand-implementations, such as explicit substitutions [Nadathur and Qi 2003] or hash-consing [Shao et al. 1998]. Many optimizations can also be done in the Fresh-Style representation, which makes them available to all the back-ends. Last but not least, we plan to investigate how to combine and choose various concrete representations, such that some binders can use one representations while others can use another depending on which operations are most often used on them.

References

Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 31–46. ACM Press, 1990.

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.

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

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.

Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Joint International Conference and Symposium on Logic Programming*, pages 259–273. MIT Press, September 1996.

Joshua Dunfield and Brigitte Pientka. Case analysis of higher-order data. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08)*, volume 228 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 69–84. Elsevier, June 2009.

Amy P. Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012.

Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 26–37. ACM, 2001.

Louis-Julien Guillemette and Stefan Monnier. A type-preserving compiler in Haskell. In James Hook and Peter Thiemann, editors, *13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 75–86. ACM, 2008.

Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic*, 6(1): 61–101, 2005.

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

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

Pierre Letouzey. A new extraction for coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs (TYPES'02)*, Lecture Notes in Computer Science (LNCS 2646), pages 200–219. Springer, 2003.

Pierre Letouzey. Extraction in coq: An overview. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *4th Conference on Computability in Europe: Logic and Theory of Algorithms*, Lecture Notes in Computer Science (LNCS 5028), pages 359–369. Springer, 2008.

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.

Luc Maranget. Compiling pattern matching to good decision trees. In *ACM SIGPLAN Workshop on ML (ML'08)*, pages 35–46. ACM, 2008.

Stefan Monnier and David Haguenauer. Singleton types here, singleton types there, singleton types everywhere. In *4th ACM SIGPLAN Workshop on Programming Languages Meets Program Verification (PLPV'10)*, pages 1–8. ACM, 2010.

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.

Gopalan Nadathur and Xiaochu Qi. Explicit substitutions in the reduction of lambda terms. In *5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 195–206. ACM, 2003.

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

Frank Pfenning. *Computation and Deduction*. Cambridge University Press, 2000. In preparation.

Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Symposium on Language Design and Implementation (PLDI'88)*, pages 199–208, June 1988.

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

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.

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.

Brigitte Pientka and Frank Pfenning. Optimizing higher-order pattern unification. In F. Baader, editor, *19th International Conference on Automated Deduction (CADE-19)*, Lecture Notes in Artificial Intelligence (LNAI) 2741, pages 473–487. Springer-Verlag, 2003.

Andrew Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, November 2003. ISSN 0890-5401.

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.

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

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.

Z. Shao. An overview of the FLINT/ML compiler. In *1997 Workshop on Types in Compilation*, 1997.

Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *International Conference on Functional Programming*, pages 313–323. ACM Press, September 1998.

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.

Christian Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

Geoff Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Journal of Functional Programming*, 18(01):87–140, 2008.

Edwin Westbrook, Nicolas Frisby, and Paul Brauner. Hobbits for Haskell: a library for higher-order encodings in functional programming languages. In *4th ACM Symposium on Haskell (Haskell'11)*, pages 35–46. ACM, 2011.