Programming inductive proofs:
a new approach based on contextual types

Brigitte Pientka

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

Abstract. In this paper, we present an overview to programming with proofs in the reasoning framework, Beluga. Beluga supports the specification of formal systems given by axioms and inference rules within the logical framework LF. It also supports implementing proofs about formal systems as dependently typed recursive functions. What distinguishes Beluga from other frameworks is that it not only represents binders using higher-order abstract syntax, but directly supports reasoning with contexts and contextual objects. Contextual types allows us to characterize precisely hypothetical and parametric derivations, i.e. derivations which depend on variables and assumptions, and lead to a direct and elegant implementation of inductive proofs as recursive functions. Because of the intrinsic support for binders and contexts, one can think of the design of Beluga as the most advanced technology for specifying and prototyping formal systems together with their meta-theory.

1 Introduction

The POPLmark challenge [ABF$^+$05] triggered wide-spread interest in mechanizing the meta-theory of programming languages, and today, knowing how to formalize proofs in a proof assistant is an essential skill. The interest in mechanizing proofs has also gone beyond small toy examples and several large-scale formalizations to verify compilers using proof-assistants are on their way (see for example [Chl10,Chl07,LCH07,Ler09]).

Although mechanizing properties about programming language are nowadays common, this endeavor is unfortunately still plagued by the necessary overhead to manage bureaucratic details. T. Altenkirch [Alt93] remarked about his formalization of the strong normalization proof for system F in Lego in 1993:

> "When doing the formalization, I discovered that the core part of the proof...is fairly straightforward and only requires a good understanding of the paper version. However, in completing the proof I observed that in certain places I had to invest much more work than expected, e.g. proving lemmas about substitution and weakening" [Alt93]

Today, not much has changed. In this paper, we will identify key operations good meta-languages should support to easily and elegantly represent formal

systems and proofs about them and give a tutorial to programming proofs in the Beluga environment [Pie08,PD08,PD10]. Beluga is a two-level framework for programming with and reasoning about formal systems. The first layer supports the representation of formal systems within the logical framework LF[HHP93]. On top of LF, we provide a functional language that supports analyzing and manipulating LF data via pattern matching.

Beluga's strength and elegance come from supporting encodings based on higher-order abstract syntax (HOAS), in which binders in the object language are represented as binders in LF's meta-language. As a consequence, users can avoid implementing common and tricky routines dealing with variables, such as capture-avoiding substitution, renaming and fresh name generation. In addition, Beluga provides intrinsic support for contexts and contextual reasoning. This is particularly convenient when representing proofs, since in this setting a natural question arises: how to represent hypothetical and parametric derivations? One may of course represent such assumptions explicitly as lists, but as a consequence we would need to enforce properties such as well-formedness, scope, uniqueness of assumptions, weakening, etc. explicitly. In Beluga, contextual objects directly characterize hypothetical and parametric derivations, and hence the user can stay away from the bureaucracy of explicitly managing and reasoning about contexts. This allows a direct and elegant implementation of inductive proofs about hypothetical and parametric derivations as recursive functions over contextual objects. Because of the intrinsic support for binders and contexts, one can think of the design of Beluga as the most advanced technology for specifying and prototyping formal systems together with their meta-theory.

In this paper, we will first revisit the formalization of the simply-typed lambda-calculus together with its type uniqueness proof (Sec. 2). In particular, we will identify and highlight challenges any meta-language used for mechanizing formal systems and proofs must address. Second, we will give a tutorial to Beluga showing how induction proofs can be directly translated into Beluga functions (Sec. 3). Finally, we summarize the theoretical foundation of Beluga and discuss its implementation (Sec. 4) . In particular, we will touch on the issues of type reconstruction for Beluga. Last but not least, we compare Beluga to related frameworks which support higher-order abstract syntax encodings of formal systems and proofs about them (Sec. 5).

2 Revisiting the simply-typed lambda-calculus

We begin by revisiting the simply-typed lambda-calculus. We introduce the base type nat and function type $\mathsf{arr}\ T_1\ T_2$. The lambda-terms are either variables, abstractions or applications.

$$\begin{array}{lll} \text{Types } T, S ::= \mathsf{nat} & & \text{Terms } M, N ::= x \\ \quad \mid \mathsf{arr}\ T\ S & & \quad \mid \mathsf{lam}\ x{:}T.M \\ & & \quad \mid \mathsf{app}\ M\ N \end{array}$$

We label the input x in the lambda-abstraction $\mathsf{lam}\ x{:}T.M$ with its type T to ensure that every lambda-term has a unique type.

2.1 Context-free formulation of typing rules

We give first a context-free formulation of the typing rules following Gentzen's original presentation of the natural deduction calculus [Gen35]. Let us first define the typing judgment more formally:

$$\text{Typing Judgment:} \quad \mathsf{oft}\ M\ T \qquad\qquad \text{read as "}M\text{ is of type }T\text{"}$$

Next, we give two typing rules, one for abstractions and one for applications and we label the inference rules with their names. Due to the context-free representation of the rules, we do not need a rule for variables, since whenever a variable is encountered when traversing an abstraction, we generate a new variable x together with the assumption $u\colon \mathsf{oft}\ x\ T$. To indicate the scope of the parameter x and the hypothesis $u\colon \mathsf{oft}\ x\ T$, we annotate the rule name $\mathsf{t\_lam}$ with super-scripts x, u.

$$\dfrac{\dfrac{}{\mathsf{oft}\ x\ T}\ u \\ \vdots \\ \mathsf{oft}\ M\ S}{\mathsf{oft}\ (\mathsf{lam}\, x{:}T.M)\ (\mathsf{arr}\ T\ S)}\ \mathsf{t\_lam}^{x,u} \qquad \dfrac{\mathsf{oft}\ M\ (\mathsf{arr}\ T\ S) \quad \mathsf{oft}\ N\ T}{\mathsf{oft}\ (\mathsf{app}\ M\ N)\ S}\ \mathsf{t\_app}$$

Since our goal is to prove, type uniqueness we also introduce an equality judgement which states that two types are equal if they are identical.

$$\text{Equality Judgment:} \quad \mathsf{eq}\ S\ T \qquad\qquad \text{read as "}S\text{ is equal to the type }T\text{"}$$

We only have the reflexivity axiom to define equality.

$$\dfrac{}{\mathsf{eq}\ \ T\ T}\ \mathsf{ref}$$

2.2 Formulation of typing rules with explicit contexts

While the context-free representation is sufficient and convenient for describing typing derivations, a formulation with explicit contexts to keep track of the assumptions is often used when actually implementing a type-checker based on these typing rules and also when reasoning about the given type system.

Before we give a formulation of the typing rules based on explicit contexts, we define the context more precisely. In particular, we specify that we are introducing the variable x together with the assumption $u\colon \mathsf{oft}\ x\ T$. As a consequence, we know that every variable x will be associated with a typing assumption $u\colon \mathsf{oft}\ x\ T$.

$$\text{Context } \Gamma ::= \cdot \mid \Gamma, x, u\colon \mathsf{oft}\ x\ T$$

We give the typing rules with explicit contexts next. While in the previous context-free formulation no variable rule was present, we now must have a rule which allows us to look up an assumption in the context. This look-up function relies on the fact that every variable in Γ has a typing assumption associated with it. We typically assume that it is associated with a unique typing assumption. This is silently enforced in the typing rule $\mathsf{t\_lam}$ where we implicitly rename the variable x, if x is already present in Γ, before extending the context Γ with the variable x together with the assumption $\mathsf{oft}\ x\ T$.

$$\frac{x, u :\ \mathsf{oft}\ x\ T \in \Gamma}{\Gamma \vdash\ \mathsf{oft}\ x\ T}\ u$$

$$\frac{\Gamma, x, u :\ \mathsf{oft}\ x\ T \vdash \mathsf{oft}\ M\ S}{\Gamma \vdash \mathsf{oft}\ (\mathsf{lam}\, x{:}T.M)\ (\mathsf{arr}\ T\ S)}\ \mathsf{t\_lam}^{x,u}\ \frac{\Gamma \vdash \mathsf{oft}\ M\ (\mathsf{arr}\ T\ S) \quad \Gamma \vdash \mathsf{oft}\ N\ T}{\Gamma \vdash \mathsf{oft}\ (\mathsf{app}\ M\ N)\ S}\ \mathsf{t\_app}$$

2.3 Type uniqueness

Finally, let us discuss the proof that every lambda-term has a unique type. While this theorem is quite simple, it is still interesting, since its proof relies on various properties of contexts and bound variables.

Theorem 1. *If* $\mathcal{D} :: \Gamma \vdash \mathit{oft}\ M\ T$ *and* $\mathcal{C} :: \Gamma \vdash \mathit{oft}\ M\ S$ *then* $\mathcal{E} :: \mathit{eq}\ T\ S$.

Proof. Induction on first typing derivation $\mathcal{D}$.

Case 1

$$\mathcal{D} = \frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \Gamma \vdash\ \mathsf{oft}\ M\ (\mathsf{arr}\ T\ S) & \Gamma \vdash\ \mathsf{oft}\ N\ T\end{array}}{\Gamma \vdash\ \mathsf{oft}\ (\mathsf{app}\ M\ N)\ S}\ \mathsf{t\_app}$$

$$\mathcal{C} = \frac{\begin{array}{cc}\mathcal{C}_1 & \mathcal{C}_2\\ \Gamma \vdash\ \mathsf{oft}\ M\ (\mathsf{arr}\ T'\ S') & \Gamma \vdash\ \mathsf{oft}\ N\ T'\end{array}}{\Gamma \vdash\ \mathsf{oft}\ (\mathsf{app}\ M\ N)\ S'}\ \mathsf{t\_app}$$

$\mathcal{E}$: $\mathsf{eq}\ \ (\mathsf{arr}\ T\ S)\ (\mathsf{arr}\ T'\ S')$	by i.h. using $\mathcal{D}_1$ and $\mathcal{C}_1$
$\mathcal{E}$: $\mathsf{eq}\ \ (\mathsf{arr}\ T\ S)\ (\mathsf{arr}\ T\ S)$ and $S = S'$ and $T = T'$	by inversion on reflexivity

Therefore there is a proof for $\mathsf{eq}\ \ S\ S'$ by reflexivity (since we know $S = S'$).

Case 2

$$\mathcal{D} = \frac{\mathcal{D}_1}{\Gamma \vdash \mathsf{oft}\ (\mathsf{lam}\ x{:}T.M)\ (\mathsf{arr}\ T\ S)}\ \mathsf{t\_lam} \qquad \mathcal{C} = \frac{\mathcal{C}_1}{\Gamma \vdash \mathsf{oft}\ (\mathsf{lam}\ x{:}T.M)\ (\mathsf{arr}\ T\ S')}\ \mathsf{t\_lam}$$

$\mathcal{E} :\ \mathsf{eq}\ \ S\ S'$	by i.h. using $\mathcal{D}_1$ and $\mathcal{C}_1$
$\mathcal{E} :\ \mathsf{eq}\ \ S\ S$ and $S = S'$	by inversion using reflexivity

Therefore there is a proof for $\mathsf{eq}\ \ (\mathsf{arr}\ T\ S)\ (\mathsf{arr}\ T\ S')$ by reflexivity.

Case 3

$$\mathcal{D} = \frac{x, u\colon \mathsf{oft}\ x\ T \in \Gamma}{\Gamma \vdash\ \mathsf{oft}\ x\ T}\ u \qquad\qquad \mathcal{C} = \frac{x, v\colon \mathsf{oft}\ x\ S \in \Gamma}{\Gamma \vdash\ \mathsf{oft}\ x\ S}\ v$$

Every variable x is associated with a unique typing assumption (property of the context), hence $v = u$ and $S = T$.

2.4 Requirements for "good" meta-languages

"Good" meta-languages should free the user from dealing with tedious and boring bureaucratic details, so s/he is able to concentrate on the essence of a proof or algorithm. Ultimately, this means users can mechanize proofs quicker, since time is not wasted on cumbersome and tedious details, resulting proofs are easier to understand, since it captures the essential steps, and automation of such proofs is more feasible. In this section, we briefly review two important aspects which arise when mechanizing formal systems and their proofs.

Support for schematic variables and bound variables When inspecting the typing rules, we notice at least two different kinds of variables. In $\mathsf{lam}\ x.M$, we bind all occurrence of the variable x in the term M. Similarly, we rely on α-renaming of bound variables and also rely on substitution for bound variables, when we for example define the reduction semantics for the lambda-calculus. In addition to bound variables, we also have used schematic variables to describe the typing rules. For example, M, N, T, S are schematic variables. When we use the typing rules to construct a concrete typing derivation for some concrete lambda-term, we instantiate these schematic variables appropriately. We will subsequently call the schematic variables M, N, T, and S *meta-variables*. In addition, we also use *context variable* Γ to denote the sequence of assumptions. Finally, we revisit the variable rule

$$\frac{x, u :\ \mathsf{oft}\ x\ T \in \Gamma}{\Gamma \vdash\ \mathsf{oft}\ x\ T}\ u$$

In this rule, the variable x is not a bound variable, but is a special schematic variable; it represents any variable declared in the context Γ. This is in contrast to schematic meta-variables, which represent any concrete lambda-term. We will call x a *parameter variable* standing for concrete variables from a context Γ.

In addition to α-renaming and substitution for bound variables, we hence also have substitutions for context variables, meta-variables and parameter variables.

Support for contexts Providing intrinsic support for representing contexts and reasoning about them, will ease the mechanization of proofs. In the previous example, we silently assumed that every declaration occurs at most once. In addition, we often rely on weakening, strengthening and substitution lemmas. Managing and reasoning about contexts is an essential part of proofs about formal systems (see also the benchmarks we recently proposed together with A.

Felty in [FP]). Supporting such reasoning about contexts simplifies the proof developments, leads to quicker prototyping, and helps to identify mistakes more easily, since the user can concentrate on the main issues of the proof without being distracted by bookkeeping.

3 Beluga: a framework for programming proofs

3.1 Overview

Beluga is a two-level system. On the first level, it provides an implementation of the logical framework LF [HHP93] similar to the implementation of LF in the Twelf system [PS99]. This supports a compact representation of formal systems and derivations exploiting higher-order abstract syntax and dependent types.

One important characteristic is that our encodings of the object language are adequate, i.e. there is a one-to-one correspondence between the terms in the object language and the terms characterized in the meta-language, namely LF. Consequently, proofs about the typing rules and lambda-terms do not deal with proof obligations which establish that the given derivation is well-formed or that a given derivation is impossible.

On top of LF, we provide a functional language which supports writing recursive functions via pattern matching on LF objects. Taking a fresh look at the proofs-as-programs paradigm, we can identify the following correspondence between on paper proofs and Beluga functions.

On paper proof	Proofs as functions in Beluga
Case analysis	Case analysis and pattern matching
Inversion	Pattern matching using let-expression
Induction Hypothesis	Recursive call

Case analysis on a derivation in the on paper proof will correspond to a case-expression which pattern matches on contextual objects describing the derivation. An inversion step in the informal proof corresponds to a case-expression with one case which can be be written using a let-expression in Beluga. Finally, the appeal to the induction hypothesis corresponds to the recursive call in a Beluga program.

As mentioned earlier, a key feature of Beluga is its support for contextual types to characterize contextual objects. Contextual types characterize contextual objects and thereby directly ensure we are only working with well-scoped derivations. Moreover, we can parameterize programs over contexts using context variables. This is essential when we want to model cases where the context grows as in the proof for type uniqueness when we considered the case which concluded with the typing rule for lambda-abstractions. It also allows us to express fine grained invariants and distinguish between different contexts.

Taken together, Beluga allows for a compact and elegant representation of proofs about formal system.

3.2 Representing simply-typed lambda-calculus in LF

To represent the simply-typed lambda-calculus in the logical framework LF, we define two LF types: the LF type `tp` for describing the types of our simply-typed lambda-calculus, and the LF type `tm` for characterizing the terms of the lambda-calculus.

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

```
tm: type .
lam : tp → (tm→tm) → tm.
app : tm → tm → tm.
```

The LF type `tp` has two constructors, `nat` and `arr`, corresponding to the types nat and arr T S respectively. Since `arr` is a constructor which takes in two arguments, its type is `tp → tp → tp`.

The LF type `tm` also has two constructors. The constructor `app` takes as input two objects of type `tm` and allows us to construct an object of type `tm`. The constructor for lambda-terms also takes two arguments as input; it first takes an object of type `tp` for the type annotation and the body of the abstraction is second. We use higher-order abstract syntax to represent the object-level binding of the variable x in the body M. Accordingly, the body of the abstraction is represented by the type `(tm → tm)`. For example, lam x:(arr nat nat) . lam y:nat . app x y is represented by `lam (arr nat nat)  λx.lam nat λy.app x y`. This encoding has several well-known advantages: First, the encoding naturally supports α-renaming of bound variables, which is inherited from the logical framework. Second, the encoding elegantly supports substitution for bound variables which reduces to β-reduction in the logical framework LF.

Next, we represent the context-free typing rules given earlier. Following the judgments-as-types principle, we define the type family `oft` which is indexed by terms `tm` and types `tp`. Each inference rule is then represented as a constant of the type `oft M T`. The rule `t_app` encodes the typing rule for applications: from derivations of `oft M (arr T S)` and `oft N T` we obtain a derivation for `oft (app M N) S`. The rule `t_lam` encodes directly the parametric hypothetical derivation "for all x assuming oft x T we can derive oft M S" using the dependent function type {`x:tm`} `oft x T → oft (M x) S`. While in the on-paper formulation of the rule, we silently assumed that we renamed x appropriately to ensure that x is new, we explicitly rename the bound variables in the representation of this rule in LF. This is achieved by the LF application `M x`.

```
oft: tm → tp → type .

t_app: oft M1 (arr T S) → oft N T
        → oft (app M N) S.
t_lam: ({x:tm} oft x T → oft (M x) S)
        → oft (lam T M) (arr T S).
```

```
eq: tp → tp → type .
ref: eq T T.
```

Finally, we represent the equality judgement as the type family `eq` which is indexed by two objects `tp`. Reflexivity is the only constant inhabiting the type `eq T S`. For a longer introduction on how to represent formal systems in the logical framework LF, we refer the reader to Pfenning's course notes [Pfe97].

3.3 Representing theorems as types in Beluga

Due to its support for dependent types and binders, Beluga is an ideal meta-language for representing theorems and proofs. Let us recall the theorem for type uniqueness from the previous section.

Theorem 2. *If* $\mathcal{D} : \Gamma \vdash \mathit{oft}\ M\ T$ *and* $\mathcal{C} : \Gamma \vdash \mathit{oft}\ M\ S$ *then* $\mathcal{E} :\ \mathit{eq}\ T\ S$.

This statement makes explicit the context Γ containing variables together with their typing assumptions. Before showing how to implement it, we describe more precisely the shape of contexts Γ, using a context schema declaration:

```
schema tctx = some [t:tp] block  x:tm. oft x t;
```

The schema `tctx` describes a context containing assumptions `x:tm`, each associated with a typing assumption `oft x t` for some type `t`. Formally, we are using a dependent product Σ (used only in contexts) to tie `x` to `oft x t`. We thus do not need to establish separately that for every variable there is a unique typing assumption: this is inherent in the definition of `tctx`. The schema classifies well-formed contexts and checking whether a context satisfies a schema will be part of type checking. As a consequence, type checking will ensure that we are manipulating only well-formed contexts, that later declarations overshadow previous declarations, and that all declarations are of the specified form.

To illustrate, we show some well-formed and some ill-formed contexts.

Context	Is of schema `tctx`?
`b1:block  x:tm.oft x (arr nat nat),` `b2:block  y:tm.oft y nat`	yes
`x:tm, u:oft x (arr nat nat)`	no (not grouped in blocks)
`y:tm`	no (typing assumption for `y` is missing)
`b:block  x:tm.oft y nat`	no (`y` is free)
`b1:block  x:tm.oft x (arr nat nat),` `b2:block  y:tm.oft x nat`	no (wrong binding structure)

Let us now show the type of a recursive function in Beluga which corresponds to the type uniqueness theorem.

```
{g:tctx} (oft (M ..) T)[g] → (oft (M ..) S)[g] → (eq T S)[ ]
```

We can read this type as follows: For all context `g` of schema `tctx`, given a derivation for `oft (M ..) T` in the context `g` and a derivation for `oft (M ..) S` in the context `g`, we return a derivation showing that `eq T S` in the empty context. Although we quantified over the context `g` at the outside, it need not be passed explicitely to a function of this type, but Beluga will be able to reconstruct it.

We call the type `(oft (M ..) T)[g]` a contextual type and the object inhabiting it a contextual object. Since the term `M` can depend on the variables declared in the context `g`, we write `(M ..)`. Formally, `M` itself is a contextual object of type `tm[g]` and `..` is the identity substitution which α-renames the bound variables. On the other hand, `T` and `S` stand for closed objects of type `tp` and they cannot refer to declarations from the context `g`. Note that these subtleties were not captured in our original informal statement of the type uniqueness theorem.

3.4 Representing inductive proofs as recursive programs

We now show the program which corresponds to the inductive proof given in Section 2.3. The proof of type uniqueness proceeds by case analysis on the first derivation. Accordingly, the recursive function pattern-matches on the first derivation `d` which has type `(oft (M ..) T)[g]`.

```
rec unique : {g:tctx}
             (oft (M ..) T)[g] → (oft (M ..) S)[g] → (eq T S)[ ] =
fn d ⇒ fn f ⇒ case d of
| [g] t_app (D1 ..) (D2 ..) ⇒                    % Application case
  let [g] t_app (F1 ..) (F2 ..) = f in
  let [ ] ref = unique ([g] D1 ..) ([g] F1 ..) in
    [ ] ref

| [g] t_lam (λx.λu. D .. x u) ⇒                   % Abstraction case
  let [g] t_lam (λx.λu. F .. x u) = f in
  let [ ] ref = unique ([g,b:block x:tm.oft x _ ] D .. b.1 b.2)
                       ([g,b] F .. b.1 b.2) in
   [ ] ref

| [g] #q.2 .. ⇒              % d : oft #q.1 T      % Assumption case
  let [g] #r.2 .. = f  in    % f : oft #q.1 S
    [ ] ref ;
```

We consider each case individually. Each case in the proof on page 4 will correspond to one case in the case-expression.

Application case: If the first derivation `d` concludes with `t_app`, it matches the pattern `[g] t_app (D1 ..) (D2 ..)`, and is a contextual object in the context `g` of type `oft (app (M ..) (N ..)) S`. `D1` corresponds to the first premise of the typing rule for applications and has the contextual type `(oft (M ..) (arr T S))[g]`.

Using a let-binding, we invert the second argument, the derivation `f` which must have type `(oft (app (M ..) (N ..)) S')[g]`. `F1` corresponds to the first premise of the typing rule for applications and has the contextual type `(oft (M ..) (arr T' S'))[g]`. The appeal to the induction hypothesis using `D1` and `F1` in the on-paper proof corresponds to the recursive call `unique ([g] D1 ..) ([g] F1 ..)`. Note that while `unique`'s type says it takes a context variable `{g:tctx}`, we do not pass it explicitly; Beluga infers it from the context in the first argument passed. The result of the recursive call is a contextual object of type `(eq (arr T S) (arr T' S'))[ ]`. The only rule that could derive such an object is `ref`, and pattern matching establishes that `arr T S` = `arr T' S'` and hence `T` = `T'` and `S` = `S'`. Therefore, there is a proof of `[ ] eq S S'` using the rule `ref`.

Abstraction case: If the first derivation `d` concludes with `t_lam`, it matches the pattern `[g] t_lam (λx.λu. D .. x u)`, and is a contextual object in the context `g` of type `oft (lam T (λx. M .. x)) (arr T S)`. Pattern matching—through a let-binding—serves to invert the second derivation `f`, which must have been by `t_lam` with a subderivation `F1 ..x u` deriving `oft (M ..x) S'` that can use `x`, `u:oft x T`, and assumptions from `g`.

The use of the induction hypothesis on `D` and `F` in a paper proof corresponds to the recursive call to `unique`. To appeal to the induction hypothesis, we need to extend the context by pairing up `x` and the typing assumption `oft x T`. This is accomplished by creating the declaration `b:block x:tm. oft x T`. In the code, we wrote an underscore `_` instead of `T`, which tells Beluga to reconstruct it. (We cannot write `T` there without binding it by explicitly giving the type of `D`, so it is easier to write `_`.) To retrieve `x` we take the first projection `b.1`, and to retrieve `x`'s typing assumption we take the second projection `b.2`.

Now we can appeal to the induction hypothesis using `D1 ..b.1 b.2` and `F1 ..b.1 b.2` in the context `g,b:block x:tm. oft x T1`. From the i.h. we get a contextual object, a closed derivation of `(equal (arr T S) (arr T S'))[ ]`. The only rule that could derive this is `ref`, and pattern matching establishes that `S` must equal `S'`, since we must have `arr T S` = `arr T1 S'`. Therefore, there is a proof of `[] equal S S'`, and we can finish with the reflexivity rule `ref`.

Assumption case: Here, we must have used an assumption from the context `g` to construct the derivation `d`. Parameter variables allow a generic case that matches a declaration `block x:tm.oft x T` for any `T` in `g`. Since our pattern match proceeds on typing derivations, we want the second component of the parameter `#q`, written as `#q.2`. The pattern match on `d` also establishes that `M = #q.1`. Next, we pattern match on `f`, which has type `oft (#q.1 ..) S` in the context `g`. Clearly, the only possible way to derive `f` is by using an assumption from `g`. We call this assumption `#r`, standing for a declaration `block y:tm. oft y S`, so `#r.2` refers to the second component `oft (#r.1 ..) S`. Pattern matching between `#r.2` and `f` also establishes that `#r.1 = #q.1`. Finally, we observe that `#r.1 = #q.1` only if `#r` is equal to `#q`. We can only instantiate the parameter variables `#r` and `#q` with bound variables from the context or other parameter variables. Consequently, the only solution to establish that `#r.1 = #q.1` is the one where both the parameter variable `#r` and the parameter variable `#q` refer to the same bound variable in the context `g`. Hence, we must have `#r = #q`, and both parameters must have equal types, and `S = S' = T = T'`. (In general, unification in the presence of Σ-types does not yield a unique unifier, but in Beluga only parameter variables and variables from the context can be of Σ type, yielding a unique solution.)

4 Revisiting the design of Beluga

4.1 Theoretical foundation

Beluga's foundation rests on the idea of contextual modal type theory which was introduced in detail in [NPP08]. A contextual object $[\Psi]M$ has contextual type $A[\Psi]$ if M has type A in the context Ψ. In the setting of Beluga, we use a contextual type to describe an LF object within a context. By design, variables occuring in M can never extrude their scope. Generalizing ideas in [DPS97] data of type $A[\Psi]$ may be embedded into computations and analyzed via pattern matching. Consequently, different arguments to a computation may have

different local contexts and we can distinguish between data of type $A[\,]$ which is closed and open data of type $A[\Psi]$ giving us fine-grained control. Since we want to allow recursion over open data objects and the local context Ψ may grow as we analyze the object M, our foundation supports context variables.

In [Pie08], we presented a simply-typed foundation for Beluga which included a bi-directional type system together with type preservation and progress proofs. Subsequently in [PD08], we extended this work to account for dependent types.

The design of Beluga distunguishes cleanly between bound variables on the LF-level and schematic variables, such as meta-variables, parameter variables and context variables.

4.2 Implementation

Beluga is implemented in OCaml. It provides a re-implementation of the logical framework LF together with LF type reconstruction and LF type checking based on explicit substitutions [ACCL90]. In addition, we designed a palatable source language for writing recursive functions about contextual objects. We list some of the challenges we addressed below.

Type reconstruction for LF Our LF type reconstruction algorithm is designed around the ideas in [Pfe91] and closely resembles the implementation of LF type reconstruction in the Twelf system [PS99]. The essential principle can be summarized as follows: Process every declaration one at a time. Given a constant declaration, we infer the type of the free variables and any omitted arguments η-expanding variables when necessary. The free variables and the variables occurring in omitted arguments together constitute the implicit arguments of the constant. When subsequently using this constant we must omit passing implicit arguments. To ilustrate, let us briefly revisit the declaration of `t_lam`.

```
t_lam: ({x:tm} oft x T → oft (M x) S)
         → oft (lam T M) (arr T S).
```

Type reconstruction will produce the following type:

```
t_lam: {T:tp}{M:tm -> tm}{S:tp}
       ({x:tm} oft x T → oft (M x) S)
         → oft (lam T (λx. M x)) (arr T S).
```

The variables `T:tp`, `S:tp`, and `M:tm → tm` are called implicit arguments. Note, we also η-expanded `M`, where it was necessary. When we subsequently use the constant `t_lam`, for example within the program `unique` where we pattern match on the shape of the objects of type `(oft (M ..) (T ..))[g]`, we simply write `[g ] t_lam (λx.λu. D .. x u)` and omit passing the arguments for `T:tp`, `S:tp`, and `M:tm → tm`.

However there are a few subtle differences between our implementation and the one found in the Twelf system: Our implementation of the constraint-based unification algorithm [EP91,Pfe91] is more conservative and addresses some known shortcomings [Ree09]. Our surface language is also slightly more restrictive than Twelf's surface language, since we only handle η-expansion and require

that the user writes LF objects in β-normal form. This makes the implementation and the theoretical foundation for LF type reconstruction more streamlined. Type reconstruction is, in general, undecidable for LF and our algorithm reports a principal type, a type error, or that the source term needs more type information.

Type reconstruction for Beluga programs In Beluga, we write recursive programs over contextual LF objects which are embedded within computations. Hence, we first generalized LF type reconstruction so it can be used for contextual LF objects. This is necessary, since these objects may contain context-variables, meta-variables and parameter variables which are absent from the pure logical framework LF. We also extended the unification algorithm to handle parameter variables and Σ-types for variables.

In addition, we extended the general principle behind LF reconstruction to support type reconstruction for computations: given a computation-level type such as for example

```
{g:tctx} (oft (M ..) T)[g] → (oft (M ..) S)[g] → (eq T S)[ ]
```

we first infer the contextual type of `M`, `T` and `S`.

```
{g:tctx}{M::tm[g]}{T::tp[ ]}{S::tp[ ]}
(oft (M ..) T)[g] → (oft (M ..) S)[g] → (eq T S)[ ]
```

In general, the free variables and the variables occurring in omitted arguments together with the context variable constitute the implicit arguments of the function and must be omitted when using the function. In the type of `unique`, we have no omitted arguments and hence the implicit arguments are the context variable `g` and the free variables `T`, `S`, and `M`. When we make a recursive call to `unique` in for example the `t_app` case, we simply write `unique ([g] D1 ..) ([g] F1 ..)` omitting the implicit arguments.

We provide special syntax for declaring that a context must be passed explicitely and will not be reconstructed. In this case, the schema of the context variable we quantify over is wrapped in `( )*`.

Finally, case-expressions pose unique challenges in the presence of dependent types, since pattern matching on an object may refine the type of the object. In our implementation, we first reconstruct the types of free variables occurring in the pattern itself and insert any omitted arguments. Next, we reconstruct a refinement substitution which is then stored together with the pattern. For example, when we pattern match on `[g] t_lam λx.λu. D .. x u` the scrutinee had type `(oft (M ..) T)[g]` but the pattern has the contextual type `(oft (lam λx. N .. x ) (arr T1 T2))[g]`. Hence, we synthesize the refinement `(lam λx.N .. x) = M ..` and `(arr T1 T2) = T`.

Context subsumption Beluga also supports context subsumption, so one can provide a contextual object in a context Ψ in place of a contextual object in some other context Φ, provided Ψ can be obtained by weakening Φ. This mechanism, similar to world subsumption in Twelf, is crucial when assembling larger proofs.

For example, if we require a context that contains only declarations `tm`, then we can supply a context which contains declarations `block x:tm. oft x nat`.

Totality Type-checking guarantees local consistency and partial correctness, but does not guarantee that functions are total. For verifying that the implemented function is total and constitutes a valid proof, we need to verify that all cases are covered and that the function is terminating, i.e. all recursive calls are on smaller arguments. Building on the algorithm described in [DP09], Joshua Dunfield recently added a coverage checker to Beluga. The final missing piece to verifying totality is a termination checker which we envision will follow ideas used in the Twelf system [RP96,Pie05] for checking that arguments in recursive calls are indeed smaller.

5 Comparison with other systems supporting HOAS

Encodings based on higher-order abstract syntax represent binders in the object language via binders in the meta-language. As a consequence, they inherit all the properties from the meta-language such as renaming of bound variables and substitution for bound variables. This means the user can avoid implementing tedious and sometimes tricky operations, such as capture-avoiding substitution. However, even in systems supporting HOAS we find different approaches to supporting contexts and the properties about them.

The Hybrid system [MMF08] tries to exploit the advantages of HOAS within the well-understood setting of higher-order logic as implemented by systems such as Isabelle and Coq. Hybrid provides a definitional layer where higher-order abstract syntax representations are compiled to de Bruijn representations, with tools for reasoning about them using tactical theorem proving and principles of (co)induction. This is a flexible approach, but contexts must be defined explicitly and properties about them must be established separately [FM09].

Abella [Gac08] is an interactive theorem prover for reasoning about specifications of formal systems. Its theoretical basis is different, but it supports encodings based on higher-order abstract syntax. However, contexts are not first-class and must be managed explicitly. For example, type uniqueness requires a lemma that each variable has a unique typing assumption, which comes for free in Beluga.

On the other side of spectrum, we find systems such as Twelf, Delphin and Beluga. Twelf is the most mature system and it provides a uniform meta-language for specifying formal systems together with their proofs using HOAS. Proofs are implemented as relations, and one establishes separately that the relation constitutes a total function and Twelf supports both termination and coverage checking. Delphin [PS09] is closest to Beluga. Proofs are implemented as functions (like Beluga) rather than relations, and its implementation uses much of the Twelf infrastructure.

In Twelf and Delphin, contexts are implicitely supported and we can reason about the contexts using world checking. However, the user does not have fine-grained control over the context. In particular we cannot state that a given object

is closed while some other object is not. In the statement of the type uniqueness theorem for example, we cannot distinguish between the fact that the typing derivation depends on assumptions from the context while the proof that two types are equal does not depend on the context.

Another difference to Twelf and Delphin lies in the type reconstruction and coverage algorithms for Beluga programs. In the type uniqueness proof for example, Beluga crucially relies on the fact that the type constructor `arr` is injective; given an object of type `eq (arr T S) (arr T S')[ ]` we reason by inversion that the only possible way we could have derived an object of this type is by the rule `ref`. Therefore, type reconstruction will synthesize `S = S'` which is then used to finish the proof. In Twelf and Delphin, we need to prove a lemma stating "if `eq (arr T S) (arr T S')` then `eq S S'`, since the coverage checker will otherwise not accept the proof.

Beluga may be thought of as the most advanced system for reasoning about formal systems, since it provides not only support for binders but also for contexts. Contexts are explicit in the system; we can distinguish between different contexts, reason with them using context subsumption, and even observe their shape by matching on them.

6 Conclusion

Beluga is a powerful programming environment for implementing formal systems together with their meta-theory. Besides the type uniqueness example, our test suite includes standard examples such as the Church-Rosser theorem, cut-admissibility, Natural Deduction to Hilbert-style proof translations, proofs about compiler transformations, and preservation and progress for various ML-like languages. Together with A. Felty, we have proposed a list of simple benchmarks [FP] which highlight the challenges due to representing and managing a context of assumptions. Recently, we also re-implemented part one of the POPLmark challenge [ABF+05], soundness and completeness of algorithmic subtying for System $F_{\texttt{sub}}$, following the proof pearl in [Pie07] where we exploit a higher-order representation of the assumptions.

Beluga is however not only a reasoning environment, but may also serve as an experimental framework for programming with dependent types and proof objects, useful for certified programming and proof-carrying code [Nec97]. We used Beluga to implement for example type-preserving CPS translations, translations between deBruijn and HOAS representation of terms, certifying type checking algorithms, and type-preserving interpreters.

In the future, we plan to concentrate on automating proofs. Currently, the recursive functions that implement induction proofs must be written by hand. We plan to explore how to enable the user to interactively develop functions in collaboration with theorem provers that can fill in parts of functions (that is, proofs) automatically.

References

ABF+05. B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In Joe Hurd and Thomas F. Melham, editors, *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS(3603), pages 50–65. Springer, 2005.

ACCL90. M. Abadi, L. Cardelli, P. Curien, and J. Lèvy. Explicit substitutions. In *17th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'90)*, pages 31–46. ACM Press, 1990.

Alt93. T. Altenkirch. A formalization of the strong normalization proof for system f in lego. In M. Bezem and J. F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications (TLCA '93)*, LNCS(664), pages 13–28. Springer, 1993.

Chl07. A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)*, pages 54–65. ACM, 2007.

Chl10. A. Chlipala. A verified compiler for an impure functional language. In Manuel V. Hermenegildo and Jens Palsberg, editors, *37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*, pages 93–106. ACM, 2010.

DP09. J. Dunfield and B. Pientka. Case analysis of higher-order data. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08)*, volume 228 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 69–84. Elsevier, 2009.

DPS97. J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher-order abstract syntax. In *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163. Springer, 1997.

EP91. C. Elliott and F. Pfenning. A semi-functional implementation of a higher-order logic programming language. In P. Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. MIT Press, 1991.

FM09. A. P. Felty and A. Momigliano. Reasoning with hypothetical judgments and open terms in hybrid. In *11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '09)*, pages 83–92, ACM, 2009.

FP. A. P. Felty and B. Pientka. Reasoning with higher-order abstract syntax and contexts: A comparison. In M. Kaufmann and L. C. Paulson, editors, *International Conference on Interactive Theorem Proving*, LNCS. Springer, 2010.

Gac08. A. Gacek. The Abella interactive theorem prover (system description). In *4th International Joint Conference on Automated Reasoning*, LNCS(5195), pages 154–161. Springer, 2008.

Gen35. G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 1935.

HHP93. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

LCH07. D. K. Lee, K. Crary, and R. Harper. Towards a Mechanized Metatheory of Standard ML. In M. Hofmann and M. Felleisen, editors, *34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*, pages 173–184, 2007. ACM Press.

Ler09. X. Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.

MMF08. A. Momigliano, A. J. Martin, and A. P. Felty. Two-Level Hybrid: A system for reasoning using higher-order abstract syntax. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'07)*, volume 196 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 85–93. Elsevier, 2008.

Nec97. G. C. Necula. Proof-carrying code. In *24th Annual Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119. ACM Press, 1997.

NPP08. A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49, 2008.

PD08. B. Pientka and J. Dunfield. Programming with proofs and explicit contexts. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 163–173. ACM Press, July 2008.

PD10. B. Pientka and J. 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)*, LNAI, 2010.

Pfe91. F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

Pfe97. F. Pfenning. Computation and Deduction, 1997.

Pie05. B. Pientka. Verifying termination and reduction properties about higher-order logic programs. *Journal of Automated Reasoning*, 34(2):179–207, 2005.

Pie07. B. Pientka. Proof pearl: The power of higher-order encodings in the logical framework LF. In K. Schneider and J. Brandt, editors, *20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'07)*,LNCS(4732), pages 246–261. Springer, 2007.

Pie08. B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM Press, 2008.

PS99. F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, LNCS(1632), pages 202–206. Springer, 1999.

PS09. A. Poswolsky and C. Schürmann. System description: Delphin—a functional programming language for deductive systems. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08)*, volume 228 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 135–141. Elsevier, 2009.

Ree09. J. Reed. Higher-order constraint simplification in dependent type theory. In A. Felty and J. Cheney, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'09)*, 2009.

RP96. E. Rohwedder and F. Pfenning. Mode and termination checking for higher-order logic programs. In H. R. Nielson, editor, *European Symposium on Programming (ESOP'96)*, pages 296–310, LNCS(1058), Springer, 1996.