Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)

Brigitte Pientka and Joshua Dunfield

McGill University, Montréal, Canada
{bpientka,joshua}@cs.mcgill.ca

Abstract. Beluga is an environment for programming and reasoning about formal systems given by axioms and inference rules. It implements the logical framework LF for specifying and prototyping formal systems via higher-order abstract syntax. It also supports reasoning: the user implements inductive proofs about formal systems as dependently typed recursive functions. A distinctive feature of Beluga is that it not only represents binders using higher-order abstract syntax, but directly supports reasoning with contexts. Contextual objects represent hypothetical and parametric derivations, leading to compact and elegant proofs. Our test suite includes standard examples such as the Church-Rosser theorem, type uniqueness, proofs about compiler transformations, and preservation and progress for various ML-like languages. We also implemented proofs of structural properties of expressions and paths in expressions. Stating these properties requires nesting of quantifiers and implications, demonstrating the expressive power of Beluga.

1 Introduction

Beluga is an environment for programming with and reasoning about deductive systems. It uses a two-level approach. The data level implements the logical framework LF [3], which has been successfully used to define logics and represent derivations and proofs. Its strength and elegance comes 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.

On top of LF, we provide a computation level that supports analyzing and manipulating LF data via pattern matching. A distinctive feature of Beluga is explicit support for contexts and contextual objects, which concisely characterize hypothetical and parametric derivations (proof objects). These contextual objects are analyzed and manipulated naturally by pattern matching.

The Beluga system is unique in having context variables, allowing generic functions that abstract over contexts. As types classify terms, context schemas classify contexts. Contexts whose schemas are superficially incompatible can be reasoned with via context weakening and context subsumption.

The main application of Beluga is to prototype deductive systems together with their meta-theory. Deductive systems given via axioms and inference rules are common in the design and implementation of programming languages, type systems, authorization and security logics, and so on. In Beluga, inductive proofs about deductive systems are directly implemented as recursive functions that case-analyze some given (possibly hypothetical) derivations. At the same time, Beluga serves as an experimental framework for programming with proof objects, useful for certified programming and proof-carrying code [6].

Beluga is implemented in OCaml. It provides a completely new implementation of LF [3] together with type reconstruction, constraint-based higher-order unification and type checking. In addition, it provides a computation language that supports writing dependently-typed recursive functions over contextual objects and explicit contexts. Building on our earlier work [10,11], we designed a palatable source language. To achieve a practical system, we implemented bidirectional type reconstruction for dependently typed functions.

We tested our implementation of LF type reconstruction on many examples from the Twelf repository [7] and found its performance competitive. We also implemented a broad range of proofs as recursive Beluga functions, including proofs of the Church-Rosser theorem, proofs about compiler transformations, subject reduction, and translation from natural deduction to Hilbert style. To illustrate the expressive power of Beluga, our test suite includes simple theorems about structural relationships between expressions and proofs about the paths in expressions. These latter theorems have nested quantifiers and implications, placing them outside the fragment of propositions expressible in systems such as Twelf. Type reconstruction of these proofs takes less than a second. Finally, Beluga provides an interpreter, based on a lazy environment-based semantics, to execute computation-level programs.

The Beluga system, including source code, examples, and an Emacs mode, is available from `http://complogic.cs.mcgill.ca/beluga/`.

To provide an intuition for what Beluga accomplishes and how it is used, we first present a type uniqueness proof in Beluga. Section 3 compares Beluga to systems with similar applications. Section 4 discusses Beluga's implementation.

2 Example: Type Uniqueness

To illustrate the core ideas behind Beluga, we implement a proof of type uniqueness for the simply-typed lambda-calculus (STLC). First, we briefly review how to represent the STLC and its typing rules in LF.

```
tp: type .                                  exp: type .
nat: tp.                                    lam : tp → (exp→exp) → exp.
arr: tp → tp → tp.                          app : exp → exp → exp.

oft: exp → tp → type .                      equal: tp → tp → type .
                                            e_ref: equal T T.
t_app: oft E1 (arr T2 T) → oft E2 T2
       → oft (app E1 E2) T.

t_lam: ({x:exp} oft x T1 → oft (E x) T2)
        → oft (lam T1 E) (arr T1 T2).
```

The first part states that `nat` is a data-level type `tp` and that `arr` takes two arguments of type `tp` and constructs a `tp`. To represent λ-term constructors, we use higher-order abstract syntax: The constructor `lam` takes a `tp` and the body of the abstraction, of type `(exp → exp)`. For example, $\mathsf{lam}\, x{:}nat\,.\, x$ is represented by `lam nat λx.x`. In the second part, we represent the typing judgment $M : T$ in our object language by an LF type `oft`, and the typing rules are represented by LF type constants `t_app` and `t_lam`.

The rule `t_app` encodes the typing rule for applications: from derivations of `oft E1 (arr T2 T)` and `oft E2 T2` we get `oft (app E1 E2) T`. The rule `t_lam` uses a parametric hypothetical derivation "for all `x` assuming `oft x T1` we can derive `oft (E x) T2`", represented as a function type `{x:exp} oft x T → oft (E x) T2`. Finally, we define the equality judgment, which simply encodes reflexivity.

The above is standard in LF. We now state type uniqueness:

Theorem. *If* $\Gamma \vdash$ `oft E T` *and* $\Gamma \vdash$ `oft E T'` *then* `equal T T'`.

This statement makes explicit the context Γ containing variable typing assumptions. Note that while terms `E` can depend on variables declared in Γ, no variables can occur in the types `T` and `T'`, though this is not captured by the statement above.

The theorem corresponds to a type of a recursive function in Beluga. 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:exp. oft x t;
```

The schema `tctx` describes a context containing assumptions `x:exp`, 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`.

We can now state the Beluga type corresponding to the statement above:

```
{g:tctx} (oft (E ..) T)[g] → (oft (E ..) T')[g] → (equal T T')[ ]
```

Read it as follows: for all contexts `g` of schema `tctx`, given derivations of `(oft (E ..) T)[g]` and of `(oft (E ..) T')[g]` we can construct a derivation of `(equal T T')[ ]`. The `[ ]` means the result is closed. As we remarked, only the term `E` can contain variables; the type `T` is closed. Although we did not state this dependency in the on-paper statement, Beluga distinguishes closed objects from objects depending on assumptions. To describe the dependency of `E` on the context `g`, we write `(E ..)` associating .. with the variable `E`. (Technically, .. is an identity substitution mapping variables from `g` to themselves.) In contrast, `T` by itself denotes a closed `tp` that cannot depend on hypotheses in `g`.

The proof of type uniqueness is by case analysis on the first derivation. Accordingly, the recursive function in Figure 1 pattern-matches on the first derivation `d`, of type `(oft (E ..) T)[g]`. The first two cases correspond to `d` concluding with `t_app` or `t_lam`. The third case corresponds to when `d` derives `(oft (E ..) T)[g]` by using a declaration from the context `g`. If the context were

```
rec  unique : {g:tctx} (oft (E ..) T)[g]
                  → (oft (E ..) T')[g]
                  → (equal T T')[ ]
= fn  d ⇒ fn  f ⇒ case  d of
| [g] t_app (D1 ..) (D2 ..) ⇒                         % Application case
  let  [g] t_app (F1 ..) (F2 ..) = f in
  let  [ ] e_ref = unique ([g] D1 ..) ([g] F1 ..) in
    [ ] e_ref

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

| [g] #q.2 .. ⇒               % d : oft #q.1 T         % Assumption case
  let  [g] #r.2 .. = f   in    % f  : oft #q.1 T'
    [ ] e_ref ;
```

Fig. 1. Implementation of type uniqueness in Beluga

concrete, we could simply refer to the concrete variable names listed, but our function is generic for any context `g`. So we use a *parameter variable* `#q` that stands for *some* declaration in `g`.

The first (application) case is essentially a simpler version of the second (abstraction) case, so we omit the application case.

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 T1 (λx. E0 .. x)) (arr T1 T2)`. Thus, `E .. = lam T1 (λx. E0 .. x)` and `T = arr T1 T2`. 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 (E0 .. x) T2'` that can use `x`, `u:oft x T1`, and assumptions from `g`. Hence, after pattern matching on `d` and `f`, we know that `E = lam  T1 (λx. E0 .. x)` and `T = arr  T1  T2` and `T' = arr  T1  T2'`.

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 its typing assumption: `g, b:block x:exp. oft x T1`. In the code, we wrote an underscore `_` instead of `T1`, which tells Beluga to reconstruct it. (We cannot write `T1` there without binding it by explicitly giving the type of `D`, so it is much 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`. 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 `g, b:block x:exp. oft x _` in the first argument passed.

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:exp. oft x T1`. We pass three arguments: the context `g` and two contextual objects. From the i.h. we get a contextual object, a closed derivation of `(equal (arr T1 T2) (arr T1 T2'))[ ]`. The only rule that could derive this is `e_ref`, and pattern matching establishes that `T2` must equal `T2'`, and hence `arr T1 T2` = `arr T1 T2'`, i.e. `T = T'`. Hence, there is a proof of `[] equal T T'`, and we can finish with the reflexivity rule `e_ref`.

Assumption case: Here, we must have used an assumption from the context `g` to construct the derivation `d`. Parameter variables `#q` allow a generic case that matches a declaration `block x:exp.oft x S` for any `S` in `g`. Since our pattern match proceeds on typing derivations, we want the second component, written `#q.2`. The pattern match on `d` also establishes that `E = #q.1` and `S = T`. Next, we pattern match on `f`, which has type `oft (#q.1 ..) T'` 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:exp. 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 both types are equal and that `S' = T'` and `#r.1 = #q.1`. Finally, we observe that `#r.1 = #q.1` only if `#r` is equal to `#q`. Consequently, both parameters 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.)

3 Related Work

There are many approaches for specifying and reasoning about formal systems. Our work builds on the experience with Twelf [7], which provides a meta-language for specifying, implementing and reasoning about formal systems using higher-order abstract syntax. However, proofs in Twelf are relations; one needs to prove separately that the relation constitutes a total function and Twelf supports both termination and coverage checking.

A second key difference is that Twelf does not explicitly support contexts and contextual data; contexts (worlds) in Twelf are implicit. Consequently, it is not possible to distinguish between different contexts within the same statement and base cases are scattered.

The third key difference is its expressiveness. In Twelf, we can only encode forall-exists statements, while Beluga directly handles a larger class of statements. An example is the statement that if, for all paths P through a lambda-term M, we know that P also characterizes all paths through another term N, then M and N must be equal.

Delphin [12] is closest to Beluga. Its implementation uses much of the Twelf infrastructure, but proofs are implemented as functions (like Beluga) rather than relations. As in Twelf, contexts are implicit with similar consequences. To have more fine-grained control over assumptions which we typically track in a context, Delphin users can use a continuation-based approach where the continuation plays the role of a context and must be explicitly managed by the programmer.

Abella [2] 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.

Finally, the Hybrid system [5] 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.

4 Implementation

Beluga is implemented in OCaml. It provides a complete reimplementation of the logical framework LF. Similarly to the Twelf core, Beluga supports type reconstruction for LF signatures based on higher-order pattern unification with constraints. In addition, we designed and implemented a type reconstruction algorithm for dependently-typed functions on contextual data.

Type reconstruction is, in general, undecidable for the data level (that is, LF) and for the computation level. For LF, our algorithm reports a principal type, a type error, or that the source term needs more type information. For our computation language, we check functions against a given type and either succeed, report a type error, or fail by asking for more type information. It is always possible to make typing unambiguous by adding more annotations.

An efficient implementation of higher-order unification is crucial to this. For higher-order patterns [4], we implemented a unification algorithm [8] and, similarly to Twelf, extended it with constraints. We also extended the algorithm to handle parameter variables and Σ-types for variables.

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.

Finally, Beluga includes an interpreter with a lazy environment-based operational semantics. This allows us to execute Beluga programs, producing concrete derivations and other LF data.

In the future, we plan to address two significant issues.

Totality. Type-checking guarantees local consistency and partial correctness, but does not guarantee that functions are total. Thus, while we can implement, partially verify, and execute functions about derivations in deductive systems, Beluga does not currently guarantee the validity of a meta-proof. The two missing pieces are coverage and termination. We formulated an algorithm [1] to ensure that all cases are covered, and plan to implement it over the next few months. Verifying termination will follow ideas in Twelf [13,9] for checking that arguments in recursive calls are indeed smaller.

Automation. 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

1. J. Dunfield and B. Pientka. Case analysis of higher-order data. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08)*, volume 228 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 69–84. Elsevier, June 2009.
2. A. Gacek. The Abella interactive theorem prover (system description). In *4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 154–161. Springer, Aug. 2008.
3. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.
4. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
5. 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.
6. G. C. Necula. Proof-carrying code. In *24th Annual Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119. ACM Press, Jan. 1997.
7. 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)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206. Springer, 1999.
8. B. Pientka. *Tabled higher-order logic programming.* PhD thesis, Department of Computer Science, Carnegie Mellon University, 2003. CMU-CS-03-185.
9. B. Pientka. Verifying termination and reduction properties about higher-order logic programs. *Journal of Automated Reasoning*, 34(2):179–207, 2005.
10. 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.
11. 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.
12. 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, June 2009.
13. E. Rohwedder and F. Pfenning. Mode and termination checking for higher-order logic programs. In H. R. Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, Apr. 1996. Springer-Verlag Lecture Notes in Computer Science (LNCS) 1058.