Beluga: programming with dependent types, contextual data, and contexts

Brigitte Pientka[1]

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

Abstract. The logical framework LF provides an elegant foundation for specifying formal systems and proofs and it is used successfully in a wide range of applications such as certifying code and mechanizing meta-theory of programming languages. However, incorporating LF technology into functional programming to allow programmers to specify and reason about formal guarantees of their programs from within the programming language itself has been a major challenge.
In this paper, we present an overview of Beluga, a framework for programming and reasoning with formal systems. It supports specifying formal systems in LF and it also provides a dependently typed functional language that supports analyzing and manipulating LF data via pattern matching. A distinct feature of Beluga is its direct support for reasoning with contexts and contextual objects. Taken together these features lead to powerful language which supports writing compact and elegant proofs.

1 Introduction

Formal systems given via axioms and inference rules play a central role in describing and verifying guarantees about the runtime behavior of programs. While we have made a lot of progress in statically checking a variety of formal guarantees such as type or memory safety, programmers typically cannot define their own safety policy and reason about it within the programming language itself.

This paper presents an overview of a novel programming and reasoning framework, called Beluga [Pie08,PD08]. Beluga uses a two-level approach: on the data-level, it supports specifications of formal systems within the logical framework LF [HHP93]. The strength and elegance of LF 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. As a consequence, users can avoid implementing common and tricky routines dealing with variables, such as capture-avoiding substitution, renaming and fresh name generation. Because of this, one can think of HOAS encodings as the most advanced technology for specifying and prototyping formal systems which leads to very concise and elegant encodings and provides the most support for such an endeavor.

On top of LF, we provide a dependently typed functional language that supports analyzing and manipulating LF data via pattern matching. A distinct feature of Beluga is its explicit support for contexts to keep track of hypothesis, and contextual objects to describe objects which may depend on them. Contextual objects are characterized by contextual types. For example, $A[\Psi]$ describes a contextual object $\Psi.M$ where M has type A in the context Ψ and hence may refer to the variables declared in the context Ψ. These contextual objects are analyzed and manipulated naturally by pattern matching.

Furthermore, Beluga supports context variables which allow us to write 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 at the moment is to prototype formal systems together with their meta-theory. Formal systems given via axioms and inference rules are common in the design and implementation of programming languages, type systems, authorization and security logics, etc. Contextual objects concisely characterize hypothetical and parametric derivations. Inductive proofs about a given formal system can be implemented as recursive functions that case-analyze some given (possibly hypothetical) derivation. Hence, Beluga serves as a proof checking framework. At the same time, Beluga provides an experimental framework for programming with proof objects. Due to its powerful type system, the programmer can not only enforce strong invariants about programs statically, but also to create, manipulate, and analyze certificates (=proofs) which guarantee that a program satisfies a user-defined safety property. Therefore, Beluga is ideally suited for applications such as certified programming and proof-carrying code [Nec97].

Beluga is an implementation in OCaml based on our earlier work [Pie08,PD08]. It provides an re-implementation of LF [HHP93] including type reconstruction, constraint-based higher-order unification and type checking. On top of LF, we designed and implemented a dependently typed functional language that supports explicit contexts and pattern matching over contextual objects. To support reasoning with contexts, we support context weakening and subsumptions. A key step towards a palatable, practical source-level language was the design and implementation of a bidirectional type reconstruction algorithm for dependently typed Beluga functions. While type reconstruction for LF and Beluga is in general undecidable, in practice, the performance is competitive. Beluga also provides an interpreter to execute programs using an environment-based semantics.

Our test suite includes many specifications from the Twelf repository [PS99]. 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 a translation from natural deduction to Hilbert style proofs. To illustrate the expressive power of Beluga, our test suite also includes simple theorems about structural relationships between expressions and proofs about the paths in expressions. These latter theorems are interesting since they require nested quantifiers and implications, placing them outside the fragment of propositions expressible in systems such as Twelf. The Beluga system, including source code, examples, and a tutorial discussing key features of Beluga, is available from

`http://complogic.cs.mcgill.ca/beluga/`.

Overview To provide an intuition for what Beluga accomplishes and how it is used, we concentrate on implementing normalization for the simply-typed lambda-calculus where lambda-terms are indexed with their types (Section 2). In Section 3, we discuss the implementation of Beluga and focus in particular on issues surrounding type reconstruction. Finally, in Section 4 we compare Beluga to related systems with similar ambition and outline future work in Section 5.

2 Example: Normalizing lambda-terms

To illustrate the core ideas behind Beluga, we show how to implement a normalizer for the simply-typed lambda-calculus. We begin by introducing the simply-typed lambda calculus. We will write T for types which consist of the base type nat and function types $T_1 \to T_2$. For lambda-terms, we use M and N. We begin by introducing it.

$$\begin{array}{lll} \text{Types } T & ::= & \mathsf{nat} \mid T_1 \to T_2 \\ \text{Term } M, N & ::= & y \mid \mathsf{lam}\, x\,.\, M \mid \mathsf{app}\ M_1\ M_2 \end{array}$$

Next, we will define normalization of simply-typed lambda-terms using the judgment $\Gamma \vdash M \longrightarrow N$ which states that given the term M we can compute its normalform N in the context Γ. The context Γ here is simply keeping track of the list of bound variables in M and N and can be defined as follows:

$$\text{Context } \Gamma ::= \cdot \mid \Gamma, x$$

We now define a normalization algorithm for the lambda-calculus where we impose a call-by-name strategy.

$$\frac{x \in \Gamma}{\Gamma \vdash x \longrightarrow x} \qquad \frac{\Gamma, x \vdash M \longrightarrow N}{\Gamma \vdash \mathsf{lam}\, x\,.\, M \longrightarrow \mathsf{lam}\, x\,.\, N}$$

$$\frac{\Gamma \vdash M_1 \longrightarrow \mathsf{lam}\, x\,.\, M' \quad \Gamma \vdash [M_2/x]M' \longrightarrow N}{\Gamma \vdash \mathsf{app}\ M_1\ M_2 \longrightarrow N}$$

$$\frac{\Gamma \vdash M_1 \longrightarrow N_1 \quad \Gamma \vdash M_2 \longrightarrow N_2 \quad N_1 \neq \mathsf{lam}\, x\,.\, M'}{\Gamma \vdash \mathsf{app}\ M_1\ M_2 \longrightarrow \mathsf{app}\ N_1\ N_2}$$

Finally, we show how to represent terms and types in the logical framework LF, and implement the normalization algorithm as a recursive program in Beluga.

Representation of simply-typed lambda-terms in LF We will represent types and terms in such a way in the logical framework LF that we will only characterize well-typed terms. The definition for types in LF is straightforward and since several excellent tutorials and notes exist already [Pfe97,Twe09], we will keep this short.

We introduce an LF type `tp` and define type constructors `nat` and `arr`. Next, we represent terms with the goal to only characterize well-typed lamda-terms. We will achieve this by indexing the type of expressions with their type using dependent type. In addition, we will employ higher-order abstract syntax to encode the binder in the object-language by binders in the meta-language, namely LF. Hence, the constructor `lam` takes in a meta-level abstraction of type `(exp T1 → exp T2)`. To illustrate, consider the object-level lambda-term $\mathsf{lam}\, x\,.\,\mathsf{lam}\, y\,.\, x\ y$. It is represented as `lam λx. lam λy. app x y` in LF.

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

Implementation of normalization in Beluga The specification of simply-typed lambda-terms in LF is standard up to this point. We now concentrate on implementing the normalization algorithm described by the judgement $\Gamma \vdash M \longrightarrow N$. Intuitively, we will implement a function which when given a lambda-term M in a context Γ, it produces a lambda-term N in the same context Γ which will be in normal form.

The statement relies on a generic context Γ since the context of variables which we will encounter when we traverse a lambda-abstraction grows.

Defining contexts using context schemas We begin by defining the shape of contexts using *context schemas* in Beluga as follows:

```
schema ctx =  exp T;
```

Schemas classify contexts just as types classify terms. The schema `ctx` describes a context which contains assumptions `x:exp T` for some type `T`. In other words, all declarations occurring in a context of schema `ctx` are instances of `exp T` for some `T`.

Defining a recursive function for normalizing lambda-terms Next, we will represent the judgment $\Gamma \vdash M \longrightarrow N$ as a computation-level type in Beluga. Since we index expressions with their types, our statement will naturally enforce that types are preserved. The type will state that "for all contexts Γ, given an expression M of type T in the context Γ, we return an expression N of type T in the context Γ". In Beluga, this is written as follows:

```
{g:(ctx)*} (exp T)[g] → (exp T)[g]
```

Writing `{g:(ctx)*}` in concrete syntax corresponds to quantifying over the context variable `g` which has schema `ctx`. We annotate the schema name `ctx` with `*` to indicate that declarations matching the given schema may be repeated. The contextual type `(exp T)[g]` directly describes an expression M with type `T` in the context `g`. The element inhabiting the computation-level type `(exp T)[g]` is called a contextual object, since they may refer to variables listed in the context `g` and hence only make sense within the context `g`. For example, the contextual object `[x:exp] lam λy. app x y` has type `exp [x:exp]` and describes an expression which may refer to the bound variable `x`.

The variable `T` which is free in the specified computation-level type is implicitly quantified at the outside and has type `tp[ ]` denoting a closed object of type `tp`. Type reconstruction will infer the type of `T` and abstract over it.

We will now show the recursive function which implements the normalization algorithm given earlier. The function proceeds by pattern matching on elements of type `(exp T)[g]` and every inference rule will correspond to one branch in the case-expression.

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

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

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

The Beluga syntax follows ideas from ML-like languages with a few extension. For example, Λ`g` ⇒ . . . introduces abstraction over the context variable `g` corresponding to quantification over the context variable `g` in the type of `norm`. We then split on the object `e` which has contextual type `(exp T)[g]`. As in the definition we gave earlier, there are three cases to consider for `e`: either it is a variable from the context, it is a lambda-abstraction, or it is an application. Each pattern is written as a contextual object, i.e. the object itself together with its context. For the variable case, we use a *parameter variable*, written as `#p ...` and write `[g] #p ...` . Operationally, it will match any declaration from the context `g` once `g` is concrete. The parameter variable `#p` is associated with the identity substitution (written in concrete syntax with ...) to explicitly state its dependency on the context `g`.

The pattern `[g] lam λx. M ... x` describes the case where the object `e` is a lambda-abstraction. We write `M ... x` for the body of the lambda-abstraction which may refer to all the variables from the context `g` (written as ...) and the variable `x`. Technically,... `x` describes the identity substitution which maps all the variables from `g, x:exp T` to themselves. We now recursively normalize the contextual object `[g,x] M ... x`. To accomplish this, we pass to the recursive call the extended context `g, x:exp _` together with the contextual object `[g,x] M ... x`. We write an underscore for the type of `x` in the context `g, x:exp _` and let type reconstruction determine it. Note, that we cannot write `x:exp T1` since `T1` would be free. Hence, supporting holes is crucial to be able to write the program compactly and avoid unnecessary type annotations. The result of the recursive call is a contextual object `[g,x] N … x` which we will use to assemble the result. In the case for applications, we recursively normalize the contextual object `[g] M1 …` and then pattern match on its result. If it returned a lambda-abstraction `lam` λ`x. M' … x`, we simply replace `x` with `M2 …` . Substitution is inherently supported in Beluga and… `(M2 … )` describes the substitution which maps all variables in `g` to themselves (written as …) and `x` is mapped to `M2 …` . In the case where normalizing `[g] M1 …` does not return a lambda-abstraction, we continue normalizing `[g] M2 …` and reassemble the final result. In conclusion, our implementation yields a natural, elegant, and very direct encoding of the formal description of normalization.

2.1 Summary of the main ideas

Beluga supports a two-level approach for programming with and reasoning about HOAS encodings. The data-level supports specifications of formal systems in the logical framework LF. On top of it, we provide an expressive computation language which supports dependent types and recursion over HOAS encodings. A key challenge is that we must traverse a λ-abstractions and manipulate objects which may contain bound variables. In Beluga, we solve this problem by using contextual types which characterize contextual objects and by introducing context variables to abstract over concrete contexts and parameterize computation with them. By design, variables occurring in contextual objects can never escape their scope, a problem which often arises in other approaches. While in our previous example, all contextual types and objects shared the same context variable, our framework allows the introduction of different context variables, if we wish to do so.

Beluga's theoretical basis is contextual modal type theory which has been described in [NPP08]. We later extended contextual modal type theory with context variables which allow us to abstract over concrete contexts and parameter variables which allow us to talk abstractly about elements of contexts. The foundation for programming with contextual data objects and contexts was first described in [Pie08] and subsequently extended with dependent types in [PD08].

3 Implementation

The Beluga system is implemented in OCaml. It consists of a re-implementation of the logical framework LF [HHP93] which supports in general specifying formal systems given via axioms and inference rules. Similar to the core of the Twelf system [PS99], we support type reconstruction for LF signatures based on higher-order pattern unification with constraints.

On top of the logical framework LF, we provide a dependently-typed functional language which allows the user to declare context schemas, write recursive functions using pattern matching on contextual data and abstract over concrete contexts using context variables and context abstraction. Designing a palatable, usable source language for Beluga has been challenging. Subsequently, we will list some of the challenges we addressed:

3.1 Higher-order unification with constraints

Higher-order unification is in general undecidable [Gol81], however a decidable fragment, called higher-order patterns [Mil91,Pfe91b] exist. A higher-order pattern is a meta-variable which is applied to distinct bound variables. In our implementation, we associate meta-variables with explicit substitutions which represents the distinct bound variables which may occur in the instantiation of the meta-variable and employ de Bruijn indices [DHKP96]. Our implementation of higher-order pattern unification is based on the development in [Pie03].

Since concentrating on higher-order patterns only is too restrictive, we adopt ideas from the Twelf system and solve higher-order pattern problems eagerly and delay non-pattern cases using constraints which are periodically revisited. Beluga also supports parameter variables which can be instantiated with either bound variables or other parameter variables and we extended unification to account for them. The main difficulty in handling parameter variables lies in the fact that their type may be a Σ-type (dependent product). In general, higher-order unification for Σ-types is undecidable. Fortunately, if we restrict Σ-types to only parameter variables or bound variables, then unique solution exists.

3.2 Type reconstruction for Beluga

Dependently typed systems can be extremely verbose since dependently typed objects must carry type information. For this reason, languages supporting dependent types such as Twelf [PS99], Delphin [PS08], Coq [BC04] , Agda [Nor07], or Epigram [MM04] all support some form of type reconstruction. However, there are hardly any concise formal description on how this is accomplished, what issues arise in practice, and what requirements the user-level syntax should satisfy. Formal foundations and correctness guarantees are even harder to find. This is a major obstacle if we want this technology to spread and dependent types are to reach mainstream programmers and implementors of programming languages.

In the setting of Beluga, we must consider two forms of type reconstruction: (1) type reconstruction for the logical framework LF, which allows us to specify formal systems, and (2) type reconstruction for the computation language which support writing recursive programs using pattern matching on LF objects.

Type reconstruction for LF We illustrate briefly the problem. Consider the expression $\mathsf{lam}\, x\, .\, \mathsf{lam}\, y\, .\, \mathsf{app}\ x\ y$ which is represented as `lam λx. lam λy. app x y` in LF. However, since expressions are indexed by types to ensure that we only represent well-typed expressions, constructors `lam` and `app` take in also two index arguments describing the type of the expression. Type reconstruction needs to infer them. The reconstructed, fully explicit representation is

```
lam (arr T S) (arr T S) λx. lam S T λy. app T S x y
```

We adapted the general principle also found in [Pfe91a]: We may omit an index argument when a constant is used, if it was implicit when the type of the constant was declared. An argument is implicit to a type if it either occurs as a free variable in the type or it is an index argument in the type. Following the ideas in the Twelf system, we do not allow the user to supply an implicit argument explicitly.

Type reconstruction is, in general, undecidable for the LF. Our algorithm similar to its implementation in the Twelf system reports a principal type, a type error, or that the source term needs more type information.

Type reconstruction for dependently typed recursive functions Type reconstruction for Beluga functions begins by reconstructing their specified computation-level type. For example, the type of `norm` was declared as `{g:(ctx)*} (exp T)[g] →(exp T)[g]` where the variable `T` is free. Type reconstruction will infer its type as `tp[ ]` and yield `{g:(ctx)*}{T::tp[ ]}(exp T)[g] →(exp T)[g]`. Note, that we do not attempt to infer the schema of the context variable at this point. This could only be done by inspecting the actual program and performing multiple passes over it. Since the type of inferred variables may depend on the context variable `g`, we insert their abstractions just after the context variable has been introduced.

Beluga functions, as the function `norm`, may be dependently typed and we apply the same principle as before. An index argument is implicit to a computation-level type if it either occurs as a free meta-variable in the computation-level type or it is an index argument in the computation-level type. Hence, we may omit passing an index argument to a Beluga function, if it was implicit when the type of the function was declared. Considering the program `norm` again this means whenever we call `norm` recursively we may omit passing the concrete type for `T`.

Let us describe reconstruction of recursive function step-by-step. Reconstruction of the recursive function `norm` is guided by its type which is now fully known. For example, we know that after introducing the context with Λg ⇒`...` , we must introduce the meta-variable `T` and the beginning of the `norm` function will be: Λg ⇒mlam `T` ⇒fn `e` ⇒case `e` of `....`

Reconstruction of the function body may refer to `T` and cannot leave any free meta-variables. It must accomplish three tasks: (1) We must insert missing arguments to recursive calls to `norm`. For example in the application case, we have `norm [g] ([g] M1 ... )`, but `norm` must in fact also take as a second input the actual type of `(M1 ... )`. (2) We must infer the type of meta-variables and parameter variables occurring in patterns. (3) We must infer the overall type of the pattern and since patterns may refine dependent types, we must compute a refinement. For example, in the case for abstractions, the type of the scrutinee is `(exp T)[g]`, but the type of the pattern is `(exp (arr T1 T2))[g]`. Hence, we must infer the refinement which state `T = (arr T1 T2)`.

One major challenge is that omitted arguments, which we need to infer, may depend on context variables, bound variables, and other meta-variables. To accomplish this we extended our unification algorithm to support meta^2-variables which allow us to track dependencies on contexts and other bound meta-variables.

Type reconstruction for the computation level is undecidable. 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 if no ground instantiation can be found for an omitted argument or if we cannot infer the type of meta-variables occurring in patterns. It is always possible to make typing unambiguous by adding more annotations.

4 Related work

In our discussion on related work, we will concentrate on programming languages supporting HOAS specifications and reasoning about them. Most closely related to our work is the Twelf system [PS99], a proof checking environment based on the logical framework LF. Its design has strongly influenced the design of Beluga. While both Twelf and Beluga support specifying formal systems using HOAS in LF, Twelf supports implementing proofs as relations. To verify that the relation indeed constitutes a proof, one needs to prove separately that it is a total function. Twelf is a mature system providing termination checking as well as an implementation of coverage checking. Both features are under development for Beluga. One main difference between Twelf and Beluga lies in the treatment of contexts. In Twelf, the actual context of hypotheses remains implicit. As a consequence, instead of a generic base case, base cases in proofs are handled whenever an assumption is introduced. This may lead to scattering of base cases and adds some redundancy. World declarations, similar to context schema declarations, check that assumptions introduced are of the expected form and that appropriate base cases are indeed present. Because worlds in the Twelf system also carry information about base cases, manual weakening is required more often when assembling larger proofs using lemmas. Explicit contexts in Beluga, make the meta-theoretic reasoning about contexts, which is hidden in Twelf, explicit. We give a systematic comparison and discuss the trade-offs of this decision together with illustrative examples in [FP10].

Another important difference between Twelf and Beluga is its expressive power. To illustrate, consider the following simple statement about lambda-terms: If for all N, N is a subterm of K implies that N is a subterm of M, then K must be a subterm of M. Because this statement requires nested quantification and implication, especially in a negative position, it is outside Twelf's meta-logic which is used to verify that a given relation constitutes a proof. While this has been known, we hope that this simple theorem illustrates this point vividly.

More recently, we see a push towards incorporating logical framework technology into mainstream programming languages to support the tight integration of specifying program properties with proofs that these properties hold.

The Delphin language [PS08] is most closely related to Beluga. Both support writing recursive functions over LF specifications, but differ in the theoretical foundation. In particular, contexts to keep track of assumptions are implicit in Delphin. It hence lacks the ability to distinguish between closed objects and objects depending on bound variables on the type-level. Delphin's implementation utilizes as much of the Twelf infrastructure as possible.

Licata and Harper [LH09] developed a logical framework supporting datatypes that mix binding and computation, implemented in the programming language Agda [Nor07,Agd09]. Their system does not explicitly support context variables and abstraction over them, but interprets binders as pronouns which refer to a designated binding site. Structural properties such as weakening, contraction, and substitution are not directly supported by the underlying theoretical foundation, but implemented in a datatype-generic manner. Finally, the current implementation does not support dependent types.

A different more pragmatic approach to allow manipulation of binding structures is pursued in nominal type systems which serve as a foundation of FreshML [SPG03]. In this approach names and α-renaming are supported but implementing substitution is left to the user. Generation of a new name and binding names are separate operations which means it is possible to generate data which contains accidentally unbound names since fresh name generation is an observable side effect. To address this problem, Pottier [Pot07] describes pure FreshML where we can reason about the set of names occurring in an expression via a Hoare-style proof system. These approaches however lack dependent typing and hence are not suitable for programming with proofs.

5 Conclusion and future work

Over the past year, we designed an implemented Beluga based on our type-theoretic foundation described in [Pie08,PD08]. Our current prototype has been tested on a wide variety of examples, including proofs of the Church-Rosser theorem, proofs about compiler transformations, subject reduction, and translation from natural deduction to Hilbert style proofs. We also used Beluga to implement proofs for theorems about structural relationships between expressions and proofs about the paths in expressions. Both of these statements require nested quantifiers and implications, placing them outside the fragment of propositions expressible in systems such as Twelf. In the future, we plan to concentrate on the following two aspects:

Guaranteeing totality of functions While type-checking guarantees local consistency and partial correctness, it does not guarantee that the implemented function is total. Thus, while we can implement, partially verify, and execute the functions, at present Beluga cannot guarantee that these functions are total and that their implementation constitutes a valid proof. The two missing pieces are coverage and termination. In previous work [DP09], we have described an algorithm to ensure that all cases are covered and we are planning an implementation of coverage during the next few months. Verifying termination of a recursive function will essentially follow similar ideas from the Twelf system [RP96,Pie05] to ensure that arguments passed to the recursive call are indeed smaller.

Automating induction proofs Our framework currently supports the implementation of induction proofs as recursive functions. It however lacks automation. In the future, we plan to explore how to connect our framework to a theorem prover which can fill in parts of the function (= proof) automatically and where the user can interactively develop functions in collaboration with a theorem prover.

References

Agd09. Agda wiki, 2009. `http://wiki.portal.chalmers.se/agda/`.

BC04. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.

DHKP96. Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 259–273, Bonn, Germany, September 1996. MIT Press.

DP09. 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.

FP10. Amy P. Felty and Brigitte Pientka. Reasoning with higher-order abstract syntax and contexts: A comparison. Technical report, School of Computer Science, McGill University, Jan 2010.

Gol81. W. D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

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

LH09. 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.

Mil91. Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.

MM04. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

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

Nor07. Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, sep 2007. Technical Report 33D.

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

PD08. Brigitte Pientka and Joshua 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.

Pfe91a. Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

Pfe91b. Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

Pfe97. Frank Pfenning. Computation and deduction, 1997.

Pie03. Brigitte Pientka. *Tabled higher-order logic programming.* PhD thesis, Department of Computer Science, Carnegie Mellon University, 2003. CMU-CS-03-185.

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

Pie08. 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.

Pot07. François Pottier. Static name control for FreshML. In *22nd IEEE Symposium on Logic in Computer Science (LICS'07)*, pages 356–365. IEEE Computer Society, July 2007.

PS99. Frank Pfenning and Carsten 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.

PS08. Adam B. Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *Proceedings of the 17th European Symposium on Programming (ESOP '08)*, volume 4960, page 93. Springer, March 2008.

RP96. Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag Lecture Notes in Computer Science (LNCS) 1058.

SPG03. 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, New York, NY, USA, 2003. ACM Press.

Twe09. Twelf wiki, 2009. `http://twelf.plparty.org/wiki/Main_Page`.