Inductive Beluga: Programming Proofs

Brigitte Pientka and Andrew Cave

McGill University, Montreal, QC, Canada,
{bpientka,acave1}@cs.mcgill.ca

Abstract. BELUGA is a proof environment which provides a sophisticated infrastructure for implementing formal systems based on the logical framework LF together with a first-order reasoning language for implementing inductive proofs about them following the Curry-Howard isomorphism.
In this paper we describe four significant extensions to BELUGA: 1) we enrich our infrastructure for modelling formal systems with first-class simultaneous substitutions, a key and common concept when reasoning about formal systems 2) we support inductive definitions in our reasoning language which significantly increases BELUGA's expressive power 3) we provide a totality checker which guarantees that recursive programs are well-founded and correspond to inductive proofs 4) we describe an interactive program development environment. Taken together these extensions enable direct and compact mechanizations. To demonstrate BELUGA's strength and illustrate these new features we develop a weak normalization proof using logical relations.

Keywords: Logical frameworks, Dependent Types, Proof Assistant

1 Introduction

Mechanizing formal systems, given via axioms and inference rules, together with proofs about them plays an important role in establishing trust in formal developments. A key question in this endeavor is how to represent variables, (simultanous) substitution, assumptions, and derivations that depend on assumptions.

BELUGA is a proof environment which provides a sophisticated infrastructure for implementing formal systems based on the logical framework LF [11]. This allows programmers to uniformly specify syntax, inference rules, and derivation trees using higher-order abstract syntax (HOAS) and relieves users from having to build up common infrastructure to mange variable binding, renaming, and (single) substitution. BELUGA provides in addition support for first-class contexts [16] and simultaneous substitutions [4], two common key concepts that frequently arise in practice. Compared to existing approaches, we consider its infrastructure one of the most advanced for prototyping formal systems [6].

To reason about formal systems, BELUGA provides a standard first-order proof language with inductive definitions [3] and domain-specific induction principles [18]. It is a natural extension of how we reason inductively about simple domains such as natural numbers or lists, except that our domain is richer, since it allows us to represent and manipulate derivations trees that may depend on assumptions. Inductive BELUGA substantially extends our previous system [19]:

First-class Substitution Variables [4] We directly support simultaneous substitutions and substitution variables within our LF-infrastructure. Dealing with substitutions manually can lead to a substantial overhead. Being able to abstract over substitutions allows us to tackle challenging examples such as proofs using logical relations concisely without this overhead.

Datatype definitions [3] We extend our reasoning language with recursive type definitions. These allow us to express relationships between contexts and derivations (contextual objects). They are crucial in describing semantic properties such as logical relations. They are also important for applications such as type preserving code transformations (e.g. closure conversion and hoisting [1]) and normalization-by-evaluation. To maintain consistency while remaining sufficiently expressive, BELUGA supports two kinds of recursive type definitions: standard *inductive* definitions, and a form which we call *stratified* types.

Totality Checking [18] We implement a totality checker which guarantees that a given program is total, i.e. all cases are covering and all recursive calls are well-founded according to a structural subterm ordering. This is an essential step to check that the given recursive program constitutes an inductive proof.

Interactive Proof Development Similar to other interactive development modes (e.g. Agda [14] or Alf [12]), we support writing holes in programs (i.e. proofs) showing the user the available assumptions and the current goal, and we support automatic case-splitting based on BELUGA's coverage algorithm [5, 18] which users find useful in writing proofs as programs.

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

2 Inductive Proofs as Recursive Programs

We describe a weak normalization proof for the simply typed lambda-calculus using logical relations – a proof technique going back to Tait [20] and later refined by Girard [10]. The central idea of logical relations is to define a relation on terms recursively on the syntax of types instead of directly on the syntax of terms themselves; this enables us to reason about logically related terms rather than terms directly. Such proofs are especially challenging to mechanize: first, specifying logical relations themselves typically requires a logic which allows a complex nesting of implications; second, to establish soundness of a logical relation, one must prove a property of well-typed *open* terms under arbitrary instantiations of their free variables. This latter part is typically stated using some notion of simultaneous substitution, and requires various equational properties of these substitutions.

As we will see our mechanization directly mirrors the theoretical development that one would do on paper which we find a remarkably elegant solution.

2.1 Representing Well-typed Terms and Evaluation in LF

For our example, we consider simply-typed lambda-terms. While we often define the grammar and typing separately, here we work directly with intrinsically typed terms, since it is more succinct. Their definition in the logical framework LF is straightforward. Below, `tm` defines our family of simply-typed lambda terms indexed by their type. In typical higher-order abstract syntax (HOAS) fashion, lambda abstraction takes a *function* representing the abstraction of a term over a variable. There is no case for variables, as they are treated implicitly. We remind the reader that this is a weak function space – there is no case analysis or recursion, and hence only genuine lambda terms can be represented.

```
LF tp : type =
| b   :  tp
| arr : tp → tp → tp;
```

```
LF tm : tp → type =
| app : tm (arr T S) → tm T → tm S
| lam : (tm T → tm S) → tm (arr T S)
| c   : tm b;
```

Our goal is to prove that evaluation of well-typed terms halts. For simplicity, we consider here weak-head reduction that does not evaluate inside abstractions, although our development smoothly extends. We encode the relation `step` stating that a term steps to another term either by reducing a redex (`beta` rule) or by finding a redex in the head (`stepapp` rule). In defining the `beta` rule, we fall back to LF-application to model substitution. In addition, we define a multi-step relation, called `mstep`, on top of the single step relation.

```
LF step : tm A → tm A → type =
| beta   : step (app (lam M) N) (M N)
| stepapp: step M M' → step (app M N) (app M' N);
LF mstep : tm A → tm A → type =
| refl   : mstep M M
| onestep: step M N → mstep N M'' → mstep M M';
```

Evaluation of a term halts if there is a value, i.e. either a constant or a lambda-abstraction which it steps to.

```
LF val : tm A → type =
| val/c   : val c
| val/lam : val (lam M);
```

```
LF halts : tm A → type =
| halts/m : mstep M M' → val M'
         → halts M;
```

2.2 Representing Reducibility using Indexed Types

Proving that evaluation of well-typed terms halts cannot be done directly, as the size of our terms may grow when we are using the `beta` rule. Instead, we define a predicate `Reduce` on well-typed terms inductively on the syntax of types, often called a reducibility predicate. This enables us to reason about logically related terms rather than terms directly.

– A term `M` of base type `b` is reducible if `halts M`.
– A term `M` of function type `(arr a B)` is reducible, if `halts M` and moreover, for every reducible `N` of type `A`, the application `app M N` is reducible.

Reducibility cannot be directly encoded at the LF layer, since it involves strong implications. We will use an indexed recursive type [3], which allows us to state *properties about well-typed terms* and define the reducibility relation recursively. In our case, it indeed suffices to state reducibility about closed terms; however, in general we may want to state properties about open terms, i.e. terms that may refer to assumptions. In BELUGA, we pair a term `M` together with the context Ψ in which it is meaningful, written as `[`Ψ` ⊢M]`. These are called contextual LF objects [13]. We can then embed contextual objects and types into the reasoning level; in particular, we can state inductive properties about contexts, contextual objects and contextual types.

```
stratified   Reduce : {A:[⊢tp]}{M:[⊢tm A]} ctype =
| I   : [ ⊢halts M] → Reduce [ ⊢b] [ ⊢M]
| Arr: [ ⊢halts M] →
        ({N:[ ⊢tm A]} Reduce [ ⊢A] [ ⊢N] → Reduce [ ⊢B] [ ⊢app M N])
     → Reduce [ ⊢arr A B ] [ ⊢M];
```

Here we state the relation `Reduce` about the closed term `M:[⊢tm A]` using the keyword **stratified**. The constructor `I` defines that `M` is reducible at base type, if `[⊢halts M]`. The constructor `Arr` defines that a closed term `M` of type `arr A B` is reducible if it halts, and moreover for every reducible `N` of type `A`, the application `app M N` is reducible. We write `{N:[⊢tm A]}` for explicit universal quantification over `N`, a closed term of type `A`. To the left of ⊢ in `[⊢tm A]` is where one writes the context the term is defined in – in this case, it is empty.

In the definition of `Reduce`, the arrows correspond to usual implications in first-order logic and denote a standard function space, not the weak function space of LF. Contextual LF types and objects are always enclosed with `[ ]` when they are embedded into recursive data-type definitions in the reasoning language. We note that the definition of `Reduce` is not (strictly) positive, and hence not inductive, since `Reduce` appears to the left of an arrow in the `Arr` case. However, there is a different criterion by which this definition is justified, namely *stratification*. We discuss this point further in Sec. 2.5

To prove that evaluation of well-typed terms halts, we now prove two lemmas:

1. All closed terms `M:[⊢tm A]` are reducible, i.e. `Reduce [⊢ A] [⊢ M]`.
2. If `Reduce [⊢ A] [⊢ M]` then evaluation of `M` halts, i.e. `[⊢ halts M]`.

The second lemma follows trivially from our definition. The first part is more difficult. It requires a generalization, which says that any well-typed term `M` under a closing substitution σ is in the relation, i.e. `Reduce [⊢ A] [⊢ M[`σ`]]`. To be able to prove this, we need that σ provides reducible instantiations for the free variables in `M`.

2.3 First-class Contexts and Simultaneous Substitutions

In BELUGA, we support first-class contexts and simultaneous substitutions. We first define the structure of the context in which a term `M` is meaningful by defining a context schema: `schema ctx = tm T;`

A context γ of schema `ctx` stands for any context that contains only declarations `x:tm T` for some `T`. Hence, `x1:tm b, x2:tm (arr b b)` is a valid context, while `a:tp,x:tm a` is not. We can then describe not only closed well-typed terms, but also a term `M` that is well-typed in a context γ as `[`$\gamma \vdash$ `tm A]` where γ has schema `ctx` [16].

To express that the substitution σ provides reducible instantiations for variables in γ, we again use an indexed recursive type.

```
inductive RedSub : {γ:ctx}{σ:[ ⊢γ]} ctype =
| Nil : RedSub [ ] [ ⊢ ^ ]
| Dot : RedSub [γ] [ ⊢ σ ] → Reduce [ ⊢A] [ ⊢M]
      → RedSub [γ, x:tm A[^]] [ ⊢ σ , M ];
```

In BELUGA, substitution variables are written as σ. Its type is written `[`$\vdash\gamma$`]`, meaning that it has domain γ and empty range, i.e. it takes variables in γ to closed terms of the same type. In the base case, the empty substitution, written as `^`, is reducible. In the `Dot` case, we read this as saying: if σ is a reducible substitution (implicitly at type `[`$\vdash\gamma$`]`) and `M` is a reducible term at type `A`, then σ with `M` appended is a reducible substitution at type `[`$\vdash\gamma$`,x:tm A[^]]` – the domain has been extended with a variable of type `A`; as the type `A` is closed, we need to weaken it by applying the empty substitution to ensure it is meaningful in the context γ. For better readability, we subsequently omit the weakening substitution.

2.4 Developing Proofs Interactively

We now have all the definitions in place to prove that any well-typed term `M` under a closing simultaneous substitution σ is reducible.

Lemma *For all* `M:[`$\gamma \vdash$ `tm A]` *if* `RedSub [`γ`] [`σ`]` *then* `Reduce [`$\vdash$`A] [`$\vdash$ `M[`σ`]]`.

This statement can be directly translated into a type in BELUGA.

```
rec main:{γ:ctx}{M:[γ⊢ tm A]}RedSub [γ] [⊢ σ] → Reduce [⊢ A] [⊢ M[σ]] = ?;
```

Logically, the type corresponds to a first-order logic formula which quantifies over the context γ, the type `A`, terms `M`, and substitutions σ. We only quantified over γ and `M` explicitly and left σ and `A` free. BELUGA's reconstruction engine [7, 17] will infer their types and abstract over them. The type says: for all γ and terms `M` that have type `A` in γ, if σ is reducible (i.e. `RedSub [`γ`] [ `$\vdash\sigma$`]`) then `M[`σ`]` is reducible at type `A` (i.e. `Reduce [`$\vdash$`A] [`$\vdash$`M[`σ`]]`).

We now develop the proof of our main theorem interactively following ideas first developed in the Alf proof editor [12] and later incorporated into Agda [14]. Traditionally, proof assistants such as Coq [2] build a proof by giving commands to a proof engine refining the current proof state. The (partial) proof object corresponding to the proof state is hidden. It is often only checked after the proof has been fully constructed. In BELUGA, as in Alf and Agda, the proof object is the primary focus. We are building (partial) proof objects (i.e. programs) directly. By doing so, we indirectly refine the proof state. Let us illustrate.

Working backwards, we use the introduction rules for universal quantification and implications; `mlam`-abstraction corresponds to the proof term for universal quantifier introduction and `fn`-abstraction corresponds to implication introduction. We write `?` for the incomplete parts of the proof object.

```
rec main:{γ:ctx}{M:[γ ⊢tm A ]}RedSub [γ] [ ⊢σ] → Reduce [ ⊢A] [ ⊢M[σ] ] =
 mlam γ, M ⇒ fn rs ⇒ ?;
```

Type checking the above program succeeds, but returns the type of the hole:

```
- Meta-Context:
  {γ : ctx}
  {M : [γ ⊢ tm A]}
-----------------------------------------------------------------------
 - Context:
  main: {γ:ctx}{M:[γ ⊢tm A]} RedSub [γ] [ ⊢σ] → Reduce [ ⊢A] [ ⊢M[σ]]
  rs: RedSub [γ] [ ⊢ σ]
=======================================================================
 - Goal Type: Reduce [ ⊢ A] [ ⊢ M[σ]]
```

The meta-context contains assumptions coming from universal quantification, while the context contains assumptions coming from implications. The programmer can refine the current hole by splitting on variables that occur either in the meta-context or in the context using the splitting tactic that reuses our coverage implementation [5, 18] to generate all possible cases.

To split on `M`, our splitting tactic inspects the type of `M`, namely `[γ ⊢ tm A]` and automatically generates possible cases using all the constructors that can be used to build a term, i.e. `[γ ⊢lam λy.M]` and `[γ ⊢app M N]`, and possible variables that match a declaration in the context γ, written here as `[γ ⊢#p]`. Intuitively writing `[γ ⊢ M]` stands for a pattern where `M` stands for a term that may contain variables from the context γ.

```
rec main:{γ:ctx}{M:[γ ⊢tm A]} RedSub [γ] [ ⊢σ] → Reduce [ ⊢A] [ ⊢M[σ] ] =
 mlam γ, M ⇒ fn rs ⇒ (case [γ ⊢ M] of
 |[γ ⊢ #p] ⇒  ?
 |[γ ⊢ app M N] ⇒  ?
 |[γ ⊢ lam λy. M] ⇒  ?
 |[γ ⊢ c] ⇒  ? );
```

Variable Case We need to construct the goal `Reduce [ ⊢ T] [ ⊢ #p[σ]]` given a parameter variable `#p` of type `[γ ⊢ tm T]`. We use the auxiliary function `lookup` to retrieve the corresponding reducible term from σ. Note that applying the substitution σ to `[γ ⊢ #p]` gives us `[ ⊢ #p[σ]]`.

```
rec lookup:{γ:ctx}{#p:[γ ⊢tm A]}RedSub [γ] [⊢σ] →Reduce [⊢ A] [⊢ #p[σ]] = ?;
```

This function is defined inductively on the context γ. The case where γ is empty is impossible, since no variable `#p` exists. If `γ = γ', x:tm T`, then there are two cases to consider: either `#p` stands for `x`, then we retrieve the last element in the substitution σ together with the proof that it is reducible; if `#p` stands for another variable in `γ'`, then we recurse. All splits can be done through the splitting tactic.

Application Case Inspecting the hole tells us that we must construct a proof for `Reduce [⊢S] [⊢app M[σ] N[σ]]`. BELUGA turned `[⊢(app M N)[σ]]` silently into `[⊢app M[σ] N[σ]]` pushing the substitution σ inside. This is one typical example where our equational theory about simultaneous substitution that we support intrinsically in our system comes into play.

Appealing to IH on `N`, written as the recursive call `main [γ] [γ⊢N] rs`, returns `rN: Reduce [⊢A] [⊢N[σ]]`. Appealing to IH on `M`, written as the recursive call `main [γ] [γ ⊢ M] rs`, gives us `Reduce [⊢arr A B] [⊢M[σ]]`. By inversion on the definition of `Reduce`, we get to the state where we must build a proof for `Reduce [⊢B] [⊢app M[σ] N[σ]]` given the assumptions

```
   rN: Reduce [ ⊢ A] [ ⊢ N[σ]]
   ha: [ ⊢ halts (arr A B) (M[σ])]
   f: {N:[ ⊢tm A]} Reduce [ ⊢ A] [ ⊢ N] → Reduce [ ⊢ B] [ ⊢ app (M[σ]) N]
```

Using `f` and passing to it `N` together with `rN`, we can finish this case. Our partial proof object has evolved to:

```
rec main:{γ:ctx}{M:[γ ⊢tm A]} RedSub [γ] [ ⊢σ] → Reduce [ ⊢A] [ ⊢M[σ]] =
 mlam γ, M ⇒ fn rs ⇒ (case [γ ⊢ M] of
 |[γ ⊢ #p] ⇒  lookup [γ] [γ ⊢ #p] rs
 |[γ ⊢ app M N] ⇒
    let rN       = main [γ] [γ ⊢N ] rs in
    let Arr ha f = main [γ] [γ ⊢M ] rs in f [⊢ _ ] rN
 |[γ ⊢ lam λy. M] ⇒  ?
 |[γ ⊢ c] ⇒  ? );
```

Abstraction Case We must find a proof for `Reduce [⊢arr T S] [⊢lam λy.M[σ,y]]`. We note again that the substitution σ has been pushed silently inside the abstraction. By definition of `Reduce` (see the constructor `Arr`), we need to prove two things: 1) `[⊢halts (lam λy.M[σ,y])]` and 2) assuming `N:[⊢tm T]` and `rN:Reduce [⊢T] [⊢N]` we need to show that `Reduce [⊢S] [⊢app (lam λy.M[σ,y]) N]`. For part 1), we simply construct the witness `[⊢halts/m refl val/lam]`. For part 2), we rely on a lemma stating that `Reduce` is backwards closed under reduction.

```
rec bwd_closed:{S:[⊢step M M']} Reduce [⊢A] [⊢M'] → Reduce [⊢A] [⊢M] = ?;
```

Using the fact that `N` provides a reducible term for `x`, we appeal to IH on `M` by recursively calling `main [γ,x:tm _] [γ,x⊢M] (Dot rs rN)`. As a result we obtain `rM:Reduce [⊢S] [⊢M[σ,N]]`. Now, we argue by the lemma `bwd_closed` and using the `beta` rule, that `Reduce [⊢S] [⊢app (lam λy.M[σ,y]) N]`. While this looks simple, there is in fact some hidden equational reasoning about substitutions. From the `beta` rule we get `[⊢ (λy.M[σ,y]) N]` which is not in normal form. To replace `y` with `N`, we need to compose the single substitution that replaces `y` with `N` with the simultaneous substitution `[σ,y]`. Again, our equational theory about simultaneous substitutions comes into play.

The complete proof object including the case for constants is given in Fig. 1. Underscores that occur are inferred by BELUGA's type reconstruction.

2.5 Totality Checking

For our programs to be considered proofs, we need to know: 1) Our programs cover all cases 2) They terminate and 3) all datatype definitions are acceptable.

```
rec main:{γ:ctx}{M:[γ⊢tm A]}RedSub [γ] [⊢σ] → Reduce [⊢A] [⊢M[σ]] =
/ total  m (main γ a s m) /
mlam γ, M ⇒ fn rs ⇒ case [γ ⊢ M] of
| [γ ⊢ #p] ⇒  lookup [γ] [γ ⊢ #p] rs
| [γ ⊢ lam λx. M] ⇒
  Arr [ ⊢ halts/m refl val/lam]
      (mlam N ⇒ fn rN ⇒
          let rM = main [γ,x:tm _] [γ,x⊢M] (Dot rs rN) in
          bwd_closed [⊢beta] rM)
| [γ ⊢ app M N] ⇒
    let rN       = main [γ] [γ ⊢N ] rs in
    let Arr ha f = main [γ] [γ ⊢M ] rs in   f [ ⊢ _ ] rN
| [γ ⊢  c] ⇒  I [ ⊢ halts/m refl val/c];
```

Fig. 1. Weak Normalization Proof for the Simply-Typed Lambda-Calculus

We verify coverage following [5, 18]. If the program was developed interactively it is covering by construction. To verify the program terminates, we verify that the recursive calls are well-founded. We use a totality declaration to specify the argument that is decreasing for a given function. In the given example, the totality declaration tells BELUGA that `main` is terminating in the fourth position; the type of `main` specifies first explicitly the context γ, followed by two implicit arguments for the type `A` and the substitution σ, that are reconstructed, and then the term `M:[γ⊢tm A]`. Following [18], we generate valid recursive calls when splitting on `M` and then subsequently verify that only valid calls are made.

Recursive datatype definitions can be justified in one of two possible ways: by declaring a definition with **inductive**, BELUGA verifies that the definition adheres to a standard strict positivity condition, i.e. there are no recursive occurrences to the left of an arrow. Positive definitions are interpreted inductively, which enables them to be used as a termination argument in recursive functions.

Alternatively, by declaring a definition with **stratified**, BELUGA verifies that there is an index argument which decreases in each recursive occurrence of the definition. This is how our definition of `Reduce` is justified: it is stratified by its `tp` index. Such types are not inductive, but rather can be thought of as being constructed in stages, or defined by a special form of large elimination. Consequently, BELUGA does not allow stratified types to be used as a termination argument in recursive functions; instead one may use its index.

3 Related Work and Conclusion

There are several approaches to specifying and reasoning about formal systems using higher-order abstract syntax. The Twelf system [15] also provides an implementation of the logical framework LF. However, unlike proofs in BELUGA where we implement proofs as recursive functions, proofs in Twelf are implemented as relations. Twelf does not support the ability to reason about contexts, contextual LF objects and first-class simultaneous substitutions. More importantly, it can only encode forall-exists statements and does not support recursive data type definitions about LF objects.

The Abella system[8] provides an interactive theorem prover for reasoning about specifications using higher-order abstract syntax. Its theoretical basis is different and its reasoning logic extends first-order logic with ∇-quantifier[9] which can be used to express properties about variables. Contexts and simultaneous substitutions can be expressed as inductive definitions, but since they are not first-class we must establish properties such as composition of simultaneous substitution, well-formedness of contexts, etc. separately. This is in contrast to our framework where our reasoning logic remains first-order logic, but all reasoning about variables, contexts, simultaneous substitution is encapsulated in our domain, the contextual logical framework. Abella's interactive proof development approach follows the traditional model: we manipulate the proof state by a few tactics such as our splitting tactic and there is no proof object produced that witnesses the proof.

Inductive BELUGA allows programmers to develop proofs interactively by relying on holes. Its expressive power comes on the one hand from indexed recursive datatype definitions on the reasoning logic side and on the other hand from the rich infrastructure contextual LF provides. This allows compact and elegant mechanizations of challenging problems such as proofs by logical relations. In addition to the proof shown here other examples include the mechanization of a binary logical relation for proving completeness of an algorithm for $\beta\eta$-equality and a normalization proof allowing reductions under abstractions, both for the simply-typed lambda calculus.

Acknowledgements Over the past 6 years several undergraduate and graduate students have contributed to the implementation: A. Marchildon, O. Savary Belanger, M. Boespflug, S. Cooper, F. Ferreira, D. Thibodeau, T. Xue.

Bibliography

[1] O. Savary Belanger, S. Monnier, and B. Pientka. Programming type-safe transformations using higher-order abstract syntax. In *Third International Conference on Certified Programs and Proofs (CPP'13)*, Lecture Notes in Computer Science (LNCS 8307), pages 243–258. Springer, 2013.

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

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

[4] A. Cave and B. Pientka. First-class substitutions in contextual type theory. In *8th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13)*, pages 15–24. ACM Press, 2013.

[5] 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)*, *Electronic Notes in Theoretical Computer Science (ENTCS 228)*, pages 69–84. Elsevier, 2009.

[6] A. P. Felty, A. Momigliano, and B. Pientka. The next 700 Challenge Problems for Reasoning with Higher-order Abstract Syntax Representations: Part 2 - a Survey. *Journal of Automated Reasoning*, 2015.

[7] F. Ferreira and B. Pientka. Bidirectional elaboration of dependently typed languages. In *16th International Symposium on Principles and Practice of Declarative Programming (PPDP'14)*. ACM, 2014.

[8] A. Gacek. The Abella interactive theorem prover (system description). In *4th International Joint Conference on Automated Reasoning*, *Lecture Notes in Artificial Intelligence (LNAI 5195)*, pages 154–161. Springer, 2008.

[9] A. Gacek, D. Miller, and G. Nadathur. Combining generic judgments with recursive definitions. In *23rd Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2008.

[10] J.-Y. Girard, Y. Lafont, and P. Tayor. *Proofs and types*. Cambridge University Press, 1990.

[11] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[12] L. Magnusson and B. Nordström. The Alf proof editor and its proof engine. In *TYPES:Types for Proofs and Programs*, *Lecture Notes in Computer Science (LNCS 806)*, pages 213–237. Springer, 1994.

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

[14] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007. Technical Report 33D.

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

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

[17] B. Pientka. An insider's look at LF type reconstruction: Everything you (n)ever wanted to know. *Journal of Functional Programming*, 1(1–37), 2013.

[18] B. Pientka and A. Abel. Structural recursion over contextual objects,. In *13th Typed Lambda Calculi and Applications (TLCA'15)*, LIPIcs-Leibniz International Proceedings in Informatics, 2015

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

[20] W. Tait. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.*, 32(2):198–212, 1967.