Reasoning with Higher-Order Abstract Syntax and Contexts: A Comparison

Amy Felty[1] and Brigitte Pientka[2]

[1] SITE, University of Ottawa, Ottawa, Canada
afelty@site.uottawa.ca
[2] School of Computer Science, McGill University, Montreal, Canada
bpientka@cs.mcgill.ca

Abstract. A variety of logical frameworks support the use of higher-order abstract syntax (HOAS) in representing formal systems given via axioms and inference rules and reasoning about them. In such frameworks, object-level binding is encoded directly using meta-level binding. Although these systems seem superficially the same, they differ in a variety of ways; for example, in how they handle a context of assumptions and in what theorems about a given formal system can be expressed and proven. In this paper, we present several case studies which highlight a variety of different aspects of reasoning using HOAS, with the intention of providing a basis for qualitative comparison of different systems. We then carry out such a comparison among three systems: Twelf, Beluga, and Hybrid. We also develop a general set of criteria for comparing such systems. We hope that others will implement these challenge problems, apply these criteria, and further our understanding of the trade-offs involved in choosing one system over another for this kind of reasoning.

1 Introduction

In recent years, the POPLmark challenge [1] has stimulated considerable interest in mechanizing the meta-theory of programming languages, and the issued problems exercise many aspects that are known to be difficult to formalize. While several solutions have been submitted showing the diversity of possible approaches, it has been hard to compare them. Part of the reason is that while the proposed examples are typical for their domain, they do not highlight the differences between systems. We will bring a different view: As experts in designing and building logical frameworks, we propose a few challenge problems which highlight the differences between different meta-languages, and thereby hopefully provide a better understanding of what practitioners should be looking for.

Our focus in this paper is on encoding meta-theory of programming languages using higher-order abstract syntax (HOAS), where we encode object-level binders with meta-level binders. 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 programming language meta-theory which leads to very concise and elegant encodings and provides the most support for such an endeavor. However concentrating on encoding binders neglects one important aspect: the support for hypothetical and parametric reasoning. Even in systems supporting HOAS, there is not a clear answer to this. On one side of the spectrum, we find the logical framework Twelf [15] or the dependently-typed functional language Beluga [13,14]. Both systems provide direct support for contexts to keep track of hypotheses. In Twelf, contexts are implicit while in Beluga they are explicit. Supporting contexts directly has two advantages. First, it eliminates the need for building up a context and managing it explicitly via a first-order representation such as a list. More importantly, it eliminates the need to explicitly prove structural properties about contexts, such as weakening. Such built-in support for contexts allows for highly compact proofs. Second, using hypothetical and parametric reasoning provides us with direct meta-level support for applying substitution lemmas. Consequently, substitution lemmas come for free.

On the other side of the spectrum of systems supporting HOAS, we have, for instance, the two-level Hybrid system [10,5] as implemented in Coq [3] and Isabelle/HOL [11], Abella [6], and the Tac prover [2], where contexts are manually represented as lists. While the substitution lemma is still obtained for free because it is an application of the cut-rule, structural properties about contexts such as weakening must typically be proven separately as lemmas. These lemmas can be tedious and they may cause difficulties when automating induction proofs (see [2]). On the other hand, since these systems do not rely on specific built-in procedures for dealing with contexts, there is more flexibility in how they are handled and the necessary reasoning is more transparent to the user. Consequently, proofs in these systems are often easier to understand and to trust.

This paper presents three case-studies which highlight the different treatments of hypothetical reasoning. Along the way, we develop a set of questions which allow a qualitative evaluation and comparison of different reasoning systems. These questions also provide guidance for users and developers in understanding better the differences and limitations. Due to space restrictions, we concentrate on the logical framework Twelf, the functional dependently-typed language Beluga, and the interactive theorem proving environment Hybrid. However, we hope that these problems will subsequently also be implemented using related approaches and serve as a starting point to understand commonalities and differences. Details about the challenge problems and their mechanization can be found in an electronic appendix which is available at `http://complogic.cs.mcgill.ca/beluga/benchmarks`.

2 Examples

In this section, we give an informal presentation and proofs of various properties of the lambda-calculus. We discuss in detail the first example which is concerned with equality reasoning and then briefly sketch the other problems. Formal proofs will be discussed in later sections only for the first. All these examples are purposefully simple, so they can be easily understood and one can quickly appreciate the capabilities and trade-offs different systems offer. Yet we believe they are representative of the issues and problems arising when formalizing formal systems and proofs about them.

2.1 Equality Reasoning for Lambda-Terms

We begin by defining the syntax of the (untyped) lambda-calculus together with a declarative definition of equality which includes reflexivity and transitivity in addition to the structural rules. We then define the algorithmic version of equality, which concentrates only on the structural rules. We model the declarative definition of equality by the judgment $\Psi \vdash \mathsf{equal}\ M\ N$ and the algorithmic one by the judgment $\Phi \vdash \mathsf{eq}\ M\ N$ and carefully define the contexts Ψ and Φ. The goal is to prove these two versions of equality to be equivalent.

Term $M ::= y \mid \mathsf{lam}\, x.\, M \mid \mathsf{app}\ M_1\ M_2$ Context $\Phi ::= \cdot \mid \Phi, \mathsf{equal}\ x\ x$
Context $\Psi ::= \cdot \mid \Psi, \mathsf{eq}\ x\ x$

Algorithmic Equality

$$\frac{\mathsf{eq}\ x\ x \in \Psi}{\Psi \vdash \mathsf{eq}\ x\ x} \qquad \frac{\Psi, \mathsf{eq}\ x\ x \vdash \mathsf{eq}\ M\ N}{\Psi \vdash \mathsf{eq}\ (\mathsf{lam}\, x.\, M)\ (\mathsf{lam}\, x.\, N)} \qquad \frac{\Psi \vdash \mathsf{eq}\ M_1\ N_1 \quad \Psi \vdash \mathsf{eq}\ M_2\ N_2}{\Psi \vdash \mathsf{eq}\ (\mathsf{app}\ M_1\ M_2)\ (\mathsf{app}\ N_1\ N_2)}$$

Declarative Equality

$$\frac{\mathsf{equal}\ x\ x \in \Phi}{\Phi \vdash \mathsf{equal}\ x\ x} \qquad \frac{\Phi, \mathsf{equal}\ x\ x \vdash \mathsf{equal}\ M\ N}{\Phi \vdash \mathsf{equal}\ (\mathsf{lam}\, x.\, M)\ (\mathsf{lam}\, x.\, N)} \qquad \frac{}{\Phi \vdash \mathsf{equal}\ M\ M}$$

$$\frac{\Phi \vdash \mathsf{equal}\ M_1\ N_1 \quad \Phi \vdash \mathsf{equal}\ M_2\ N_2}{\Phi \vdash \mathsf{equal}\ (\mathsf{app}\ M_1\ M_2)\ (\mathsf{app}\ N_1\ N_2)} \qquad \frac{\Phi \vdash \mathsf{equal}\ M\ L \quad \Phi \vdash \mathsf{equal}\ L\ N}{\Phi \vdash \mathsf{equal}\ M\ N}$$

It may be slightly unusual to keep the fact that a variable is equal to itself as a declaration in the context in both formulations. It is only strictly necessary in the first. There are two main reasons. 1) Explicitly introducing the appropriate assumption about each variable is a general methodology which scales to more expressive assumptions. For example, when we specify typing rules, we must introduce a typing context that keeps track of the fact that a given variable has a certain type. 2) Choosing this formulation will also make our proofs more elegant and compact, while at the same time highlight the issues which arise when working with two formal systems each using different assumptions.

We begin by proving that reflexivity and transitivity are indeed admissible from the algorithmic definition of equality.

Theorem 1 (Admissibility of Reflexivity and Transitivity)

1. *If Ψ contains premises for all the free variables in M, then $\Psi \vdash \mathsf{eq}\ M\ M$.*
2. *If $\Psi \vdash \mathsf{eq}\ M\ L$ and $\Psi \vdash \mathsf{eq}\ L\ N$ then $\Psi \vdash \mathsf{eq}\ M\ N$.*

The first theorem can be proven by induction on M. The second can be proven by induction on the first derivation. We now state that when we have a proof for $\mathsf{equal}\ M\ N$ then we also have a proof using algorithmic equality.

Attempt 1 (Completeness). *If* $\Phi \vdash \mathsf{equal}\ M\ N$ *then* $\Psi \vdash \mathsf{eq}\ M\ N$*.*

However, we note that this statement does not contain enough information about how the two contexts Φ and Ψ are related. In the base case, where we have that $\Phi \vdash \mathsf{equal}\ x\ x$, we must know that for every variable x in Φ there exists a corresponding assumption such that $\mathsf{eq}\ x\ x$ in Ψ. There are two solutions to this problem. 1) We state how two contexts are related and then assume that if this relation holds the theorem holds. 2) We generalize the context used in the theorem such that it contains both assumptions as follows:

$$\text{Generalized context } \Gamma ::= \cdot \mid \Gamma, \mathsf{eq}\ x\ x, \mathsf{equal}\ x\ x$$

where we deliberately state that the assumption $\mathsf{eq}\ x\ x$ always occurs together with the assumption $\mathsf{equal}\ x\ x$, and then apply weakening and strengthening as needed to apply the equality inference rules. Both approaches can be mechanized and we discuss some of the trade-offs later. For now we will concentrate on the latter approach and state the revised generalized theorem.

Theorem 2 (Completeness). *If* $\Gamma \vdash \mathsf{equal}\ M\ N$ *then* $\Gamma \vdash \mathsf{eq}\ M\ N$*.*

Proof. Proof by induction on the first derivation. We show three cases which highlight the use of weakening and strengthening.

Case 1: Assumption from context
We know $\Gamma \vdash \mathsf{equal}\ x\ x$ where $\mathsf{equal}\ x\ x \in \Gamma$ by assumption. Because of the definition of Γ, we know that whenever we have an assumption $\mathsf{equal}\ x\ x$, we also must have an assumption $\mathsf{eq}\ x\ x$.

Case 2: Reflexivity rule
If the last step applied in the proof was the reflexivity rule $\Gamma \vdash \mathsf{equal}\ M\ M$, then we must show that $\Gamma \vdash \mathsf{eq}\ M\ M$. By the reflexivity lemma, we know that $\Psi \vdash \mathsf{eq}\ M\ M$. By weakening the context Ψ, we obtain the proof for $\Gamma \vdash \mathsf{eq}\ M\ M$.

Case 3: Equality rule for lambda-abstractions

$\Gamma \vdash \mathsf{equal}\ (\mathsf{lam}\, x.\, M)\ (\mathsf{lam}\, x.\, N)$	by assumption
$\Gamma, \mathsf{equal}\ x\ x \vdash \mathsf{equal}\ M\ N$	by decl. equality rule for lambda-abstraction
$\Gamma, \mathsf{eq}\ x\ x, \mathsf{equal}\ x\ x \vdash \mathsf{equal}\ M\ N$	by weakening
$\Gamma, \mathsf{eq}\ x\ x, \mathsf{equal}\ x\ x \vdash \mathsf{eq}\ M\ N$	by i.h.
$\Gamma, \mathsf{eq}\ x\ x \vdash \mathsf{eq}\ M\ N$	by strengthening
$\Gamma \vdash \mathsf{eq}\ (\mathsf{lam}\, x.\, M)\ (\mathsf{lam}\, x.\, N)$	by alg. equality rule for lambda-abstraction

This proof demonstrates many issues related to the treatment of bound variables and the treatment of contexts. First, we need to be able to apply a lemma which was proven in a context Ψ in a different context Γ. Second, we need to apply weakening and strengthening in the proof. Third, we need to be able to know the structure of the context and we need to be able to take advantage of it. We focus here on these structural properties of contexts, but of course many proofs also need the substitution lemma.

2.2 Reasoning about Variable Occurrences

In this example, we reason about the shape of terms instead of equality of terms. The idea is to compare terms up to variables. For example $\mathsf{lam}\, x.\, \mathsf{lam}\, y.\, \mathsf{app}\ x\ y$ would have the same shape as $\mathsf{lam}\, x.\, \mathsf{lam}\, y.\, \mathsf{app}\ y\ x$ but these two terms are obviously not equal. We use the judgment $\Phi \vdash \mathsf{shape}\ M_1\ M_2$ to describe that the term M_1 and the term M_2 have the same shape or structure. Thinking of the lambda-terms being described by a syntax tree, comparing the shape of two terms corresponds to comparing two syntax trees where we do not care about specific variable names which are at the leaves of it. The definition for $\mathsf{shape}\ M_1\ M_2$ can be found in the electronic appendix.

We now prove that if M_1 and M_2 have the same shape, then they must have the same number of variables using the judgment $\Phi \vdash \mathsf{var{-}occ}\ M\ I$ where I describes the total number of variable occurrences in the term M. So for example, the total number of variable occurrences in the term $\mathsf{lam}\, x.\, \mathsf{lam}\, y.\, \mathsf{app}\ (\mathsf{app}\ y\ x)\ x$ is 3. If we think of the lambda-term as a syntax tree, then I describes the number of leaves in the syntax tree described by the term M. We give three different variations, intended to show differences among systems.

Theorem 3

1. *If $\Phi \vdash \mathsf{shape}\ M_1\ M_2$ then there exists an I such that $\Phi \vdash \mathsf{var{-}occ}\ M_1\ I$ and $\Phi \vdash \mathsf{var{-}occ}\ M_2\ I$. Furthermore I is unique.*
2. *If $\Phi \vdash \mathsf{shape}\ M_1\ M_2$ then for all I. $\Phi \vdash \mathsf{var{-}occ}\ M_1\ I$ implies $\Phi \vdash \mathsf{var{-}occ}\ M_2\ I$.*
3. *If $\Phi \vdash \mathsf{shape}\ M_1\ M_2$ and $\Phi \vdash \mathsf{var{-}occ}\ M_1\ I$ then $\Phi \vdash \mathsf{var{-}occ}\ M_2\ I$.*

2.3 Reasoning about Subterms in Lambda-Terms

For the next example, we define when a given lambda-term M is a subterm of another lambda-term N and hence we consider M to be structurally smaller than (or equal to) N using the following judgment: $\Psi \vdash \mathsf{le}\ M\ N$. Rules for this judgment are given in the electronic appendix. We concentrate here on stating a very simple intuitive theorem that says that if for all terms N, if N is smaller than K implies that N is also smaller than L, then clearly K is smaller than L.

Theorem 4. *If for all N. $\Psi \vdash \mathsf{le}\ N\ K$ implies $\Psi \vdash \mathsf{le}\ N\ L$ then $\Psi \vdash \mathsf{le}\ K\ L$.*

This theorem is interesting because in order to state it, we nest quantification and implications placing them outside the fragment of propositions directly expressible in systems such as Twelf.

3 Mechanization in Twelf and Beluga

In this section, we discuss how the previous examples are implemented in Twelf and Beluga. Both systems share an encoding of expressions and inference rules for declarative and algorithmic equality in the logical framework LF [7]. There are several excellent tutorials on how to represent inference rules in the logical framework LF, and hence we keep this very short.

Formalization of Lambda-Terms and Declarative and Algorithmic Equality. Using HOAS, we represent binders in the object-language (see for example $\mathsf{lam}\,x.\,M$) using binders in the meta-language, i.e., the logical framework LF. Hence the constructor `lam` takes in a function of type `exp → exp`. For example, the object-language term $\mathsf{lam}\,x.\,\mathsf{lam}\,y.\,\mathsf{app}\ x\ y$ will be represented in LF as `lam (λx. lam (λy. app x y) )`. Bound variables found in the object language, are not explicitly represented in the meta-language.

Object-language	Representation in LF
Term $M ::= y$	`exp : type`
$\mid \mathsf{lam}\,x.\,M$	`lam :(exp → exp) → exp.`
$\mid \mathsf{app}\ M_1\ M_2$	`app : exp → exp → exp.`

We give the implementation of the declarative and algorithmic equality rules next using the two type families `eq` and `equal` respectively. Each inference rule is then represented as a type. Hypothetical derivations (as in the rule for lambda-abstraction) are represented as higher-order functions.

```
eq: exp → exp → type.
eq_lam :  (Πx : exp. eq x x → eq (E x) (F x))
          → eq (lam (λx. E x)) (lam (λx. F x)).
eq_app : eq E1 F1 → eq E2 F2 → eq (app E1 E2) (app F1 F2).

equal: exp → exp → type.
e_l: (Πx:exp. equal x x → equal (T x) (T' x))
     → equal (lam (λx. T x)) (lam (λx. T' x)).
e_a: equal T2 S2  → equal T1 S1  → equal (app T1 T2) (app S1 S2).
e_r: equal T T.
e_t: equal T R → equal R S → equal T S.
```

Proofs as Recursive Functions. Beluga is a functional language where (hypothetical) derivations are characterized by contextual objects and an inductive proof about derivations is written as a recursive function using pattern matching on them. Each case of the proof corresponds to one branch in the function. First, we define the context schema for the context Ψ which was used in defining algorithmic equality to track assumptions of the form $\mathsf{eq}\ x\ x$ (see page 230). Context schemas classify contexts just as types classify terms. It can be defined as follows: `schema eqCtx = block x:exp . eq x x;` This states that our context consists of blocks of assumptions, containing `x:exp` and `eq x x`. More formally, the block-construct introduces a Σ-type grouping the two declarations together.

The reflexivity theorem which stated that for all M there exists a proof for $\mathsf{eq}\ M\ M$ can then be implemented as a recursive function called `ref` which will have the following type: `rec ref : {ψ:(eqCtx)*} {M::exp[ψ]} (eq (M..)(M..))[ψ]`

This can be read as follows: for all contexts ψ which have schema `(eqCtx)*`, for all terms `M`, we have a proof that `(eq (M..) (M..))[ψ]`. Explicit quantification over the context variable ψ is written using curly brackets in `{ψ:(eqCtx)*}`. The schema is annotated with `*` to denote that declarations of the specified schema may be repeated and the context must be passed explicitly by the user. For universally quantifying over `M`, we use curly brackets in `{M::exp[ψ]}`. Central to Beluga is the idea of a contextual type. `M` for example has type `exp[ψ]` which describes an object `M` which has type `exp` in the context ψ. `M` is hence an expression which may refer to variables in the context ψ. When we use `M` it is associated with a substitution which maps all the variables in ψ to the correct target context. In the example, we use `M` within the contextual type `(eq (M..) (M..))[ψ]`. Hence, `M` is declared in the context ψ and because it is also used in the context ψ, it is associated with the identity substitution, which is written as `..` in our concrete syntax. Intuitively, it means `M` can depend on all the variables which occur in the context described by ψ. The derivation $\Psi \vdash \mathsf{eq}\ M\ M$ is directly captured by the contextual type `(eq (M..) (M..))[ψ]`.

Before we represent the completeness theorem as a recursive function `ceq`, we define the schema of the generalized context, following our previous informal development as follows: `schema eCtx = block x:exp,u:eq x x.equal x x ;`

Finally, we state the type and implementation of the function `ceq`. We indicate that the context γ is implicit in the actual implementation of the proof and will be reconstructed by omitting the `(...)*` when declaring the schema of γ.

```
 rec ceq: {γ:eCtx} (equal (T..) (S..))[γ] → (eq (T..) (S..))[γ] =
 fn e ⇒ case e of
| [γ] #p.3.. ⇒ [γ] #p.2..                               % Assumption from context
| [γ] e_r (T.. )⇒ ref [γ] <γ. _ >                       % Reflexivity
| [γ] e_t (D2..) (D1..) ⇒                               % Transitivity
  let  [γ] F2.. = ceq ([γ] D2..) in
  let  [γ] F1.. = ceq ([γ] D1..) in
    trans ([γ] F1..)  ([γ] F2..)
| [γ] e_l (λx. λu. D.. x u) ⇒                           % Abstraction
  let  [γ,b:block x:exp,u:eq x x . equal x x] F.. b.1 b.2 =
        ceq ([γ, b:block x:exp, u:eq x x . equal x x] D.. b.1 b.3)
  in
     [γ] eq_lam (λx.λv. F.. x v)
| [γ] e_a (D2..) (D1..) ⇒                                % Application
    let  [γ] F1.. = ceq ([γ] D1..) in
    let  [γ] F2.. = ceq ([γ] D2..) in
      [γ] eq_app (F1..) (F2..) ;
```

We explain the three cases shown also in the proof on page 231. First, let us consider the case where we used an assumption from the context. It is modelled using parameter variables `#p` in Beluga. Operationally, `#p` can be instantiated with any bound variable from the context γ. Since the context γ consists of blocks with the following structure: `block x:exp,u:eq x x . equal x x`, we in fact want to match on the third element of such a block. This is written as `#p.3..`. The type of `#p.3` is `equal (#p.1..) (#p.1..)`. Since our context always contains a block and the parameter variable `#p..` describes such a block, we know that there exists a proof for `eq (#p.1..) (#p.1..)` which can be described by `#p.2..`.

Second, we consider the case where we applied the reflexivity rule `e_r` as a last step. In this case, we need to refer to the reflexivity lemma we proved about algorithmic equality. To use the function `ref` which implements the reflexivity lemma for algorithmic equality we however need a context of schema `eqCtx` but the context used in the proof for `ceq` is of schema `eCtx`. Since the schema `eCtx` in fact contains at least as much information as the schema `eqCtx`, we should be allowed to pass a context of schema `eCtx` when a context of schema `eqCtx` is required. This is achieved by incorporating context subsumption in Beluga (see [17,8] for an introduction to context subsumption).

Third, we consider the case for `e_lam`. In this case, we extend the context with the new declarations about variables and pass to the recursive call `ceq` the derivation `[`γ`, b:block x:exp,u:eq x x.equal x x] D.. b.1 b.3)`. Weakening is built-in. Although the derivation `D` only depends on the context ψ`,x:exp,u:equal x x`, we can use it in the context which also has the assumption `eq x x`. Applying the induction hypothesis corresponds to the recursive call. The result of recursive call is a derivation `F`, where `F` only depends on `x:exp` and `u:eq x x`. In the on-paper proof we employed strengthening. Finally, we use `F` to assemble the final result `eq_lam (λx.λv. F.. x v)`.

The cases where we applied the application rule `e_a` and the transitivity rule `e_t` as a last step are straightforward. In both cases, we simply appeal to the induction hypothesis on the subderivations `D1..` and `D2..`. This is implemented as a recursive call to `ceq` using the derivation `[`γ`] D1..` and the recursive call to `ceq` using the derivation `[`γ`] D2..`. Finally we assemble the result. In the case for applications we use the rule `eq_app` and in the case for transitivity we use the lemma `trans`.

Proofs as Relations. In Twelf, the proof is implemented as a relation between two derivations, and we separately check that it constitutes a total function. The mode declaration says how we must read the relation operationally. The theorem is represented as a type family, and each case of the proof is represented as one type (or clause). The proof is similar to the implementation in Beluga, with a few exceptions. In Twelf, the context in which we prove the theorem is implicit, and there is no generic variable case, but the variable case is folded into the case for lambda-abstraction. We begin by stating the reflexivity theorem as a relation in Twelf together with the corresponding world declaration. Similar to context schemas, world declarations allow us to describe the context in which the theorem is proven. However, unlike schemas, worlds also keep information about base cases. Since variable cases are handled implicitly, not explicitly, the context must not only list assumptions `x:exp` and `u:equal x x` but in addition a proof that reflexivity holds for `x`, i.e., `ref x u`.

```
ref: ΠT:tp.equal T T → type.              %mode ref +T -D.

%block r_block : block {x:term}{u:equal x x}{r_x: ref x u}.
%worlds (r_block) (ref T D).
```

We now inspect the implementation of the proof of the completeness proof from page 231. It will be very similar to our proof in Beluga, except for the treatment of base cases and contexts.

```
ceq: eq T S → equal T S → type.           %mode ceq +E -D.
c_r: ref _ E
     → ceq eq_r E.

c_t: ceq D1 E1 → ceq D2 E2 → tr E1 E2 E
     → ceq (eq_t D2 D1) E.

c_l: (Πx:tm.Πu:equal x x.Πt_x:tr u u u.Πr_x: ref x u.Πv:eq x x.
      ceq v u → ceq (E x v) (D x u))
     → ceq (eq_l E) (eq_l D).
```

```
c_a: ceq F1 D1 → ceq F2 D2
     → ceq (eq_a F2 F1) (equ_a D2 D1).
%block cl:block {x:term}{u:equal x x}{t_x:tr u u u}{r_x:ref x u}{v:eq x x}
           {c_x: ceq v u}.
%worlds (cl) (ceq E D).
%total E (ceq E D).
```

We can read for example the case `c_a` for applications as follows: Given the relation `ceq F1 D1` (i.h. on the derivation `F1` and `D1`) and the relation `ceq F2 D2` (i.h. on the derivation `F2` and `D2`), we know `ceq (e_a F2 F1) (equ_a D2 D1)`. This case is closely related to the case in our functional program. The differences arise in the case for lambda-abstractions. Since Twelf supports contexts only implicitly, we must introduce a variable `x` not only together with the assumption `equal x x` and `eq x x`, as we do in Beluga, but we also must assume that the reflexivity and transitivity lemma hold for this variable and that indeed there is a proof that guarantees that whenever we have `equal x x` we must have a proof for `eq x x`.

Because there is no explicit context and no explicit variable case when reasoning about formal systems, the base cases are scattered and pollute our context. Consequently, it now is harder to compose lemmas and reason about the relationship between different contexts. For example, the world described by blocks `r_block` is not a prefix of the world described by blocks `cl`. In Twelf, this will lead to world subsumption failure and the user needs to weaken manually the proof for reflexivity to include assumptions `t_x:trans u u u`.[1] Apart from the issues around contexts, the Twelf allows a very compact representation of the completeness proof. Weakening and strengthening is handled automatically. For a more detailed explanation regarding the formalization of proofs in the Twelf system and context subsumption, we refer the reader to [8].

4 Mechanization in Two-Level Hybrid

The Hybrid approach [10] exploits the advantages of HOAS within general theorem proving systems. We use a pretty-printed version of Coq concrete syntax in this paper. *Prop* is the type of meta-level formulas and the usual symbols (e.g., $\rightarrow$, $\forall$) represent the meta-level connectives and quantifiers. $[\![\ A_1; A_2; \ldots; A_n\ ]\!] \rightarrow A$ abbreviates $A_1 \rightarrow (A_2 \rightarrow \cdots (A_n \rightarrow A) \cdots)$, or equivalently $(A_1 \wedge A_2 \wedge \cdots \wedge A_n) \rightarrow A$. The symbol $==$ denotes definitional equality. Free variables in inductive definitions and statements of theorems are implicitly universally quantified at the top-level of each clause or statement.

Hybrid provides a type *expr* and a set of operators on this type used to encode object-language syntax. It is built definitionally on the foundation of the meta-language of the underlying theorem prover; no axioms are introduced. The operators that are used in this paper, with their types are:

$$\mathsf{CON} : \mathit{con} \rightarrow \mathit{expr} \mid \mathsf{APP} : \mathit{expr} \rightarrow \mathit{expr} \rightarrow \mathit{expr} \mid \mathsf{LAM} : (\mathit{expr} \rightarrow \mathit{expr}) \rightarrow \mathit{expr}.$$

[1] Alternatively, we can also weaken the transitivity lemma and change the order of blocks.

We define the type *con* later to represent the constants of an object-language.

In the two-level approach used by Hybrid, a specification logic (SL) is defined inductively and used to encode inference rules of object-languages. Hypothetical and parametric judgments are encoded in the SL layer. In this paper, we use a simple SL, a sequent formulation of a fragment of second-order minimal logic with backchaining, adapted from [9] (and also used in [10]). Its syntax and inference rules can be encoded directly as follows:

$$\begin{aligned} \textit{inductive } oo :=\ & \mathsf{tt} : oo \mid \langle \_ \rangle : atm \to oo \mid \_ \,\mathsf{and}\, \_ : oo \to oo \to oo \\ & \mid \_ \,\mathsf{imp}\, \_ : atm \to oo \to oo \mid \mathsf{all} : (expr \to oo) \to oo \end{aligned}$$

$$\begin{array}{lrcl} \textit{inductive } \_ \rhd\_ \_ : atm\ list \to nat \to oo \to Prop := & & & \\ s\_tt : & & \to & \Gamma \rhd_n \mathsf{tt} \\ s\_and : & [\![\, \Gamma \rhd_n G_1;\ \Gamma \rhd_n G_2 \,]\!] & \to & \Gamma \rhd_{n+1} (G_1 \;\mathsf{and}\; G_2) \\ s\_all : & [\![\, (\forall x.\, \mathsf{proper}\, x \to \Gamma \rhd_n G\ x) \,]\!] & \to & \Gamma \rhd_{n+1} (\mathsf{all}\, x.\, G\ x) \\ s\_imp : & [\![\, A, \Gamma \rhd_n G \,]\!] & \to & \Gamma \rhd_{n+1} (A \;\mathsf{imp}\; G) \\ s\_init : & [\![\, A \in \Gamma \,]\!] & \to & \Gamma \rhd_n \langle A \rangle \\ s\_bc : & [\![\, A \longleftarrow G;\ \Gamma \rhd_n G \,]\!] & \to & \Gamma \rhd_{n+1} \langle A \rangle \end{array}$$

In the inductive definition of *oo*, *atm* is a parameter used to represent atomic predicates of the object-language and $\langle \_ \rangle$ coerces atoms into propositions of type *oo*. In the definition of the SL, we use the symbol $\rhd$ for the sequent arrow and decorate it with natural numbers to allow reasoning by (complete) induction on the *height* of a proof. For convenience we write $\Gamma \rhd G$ if there exists an n such that $\Gamma \rhd_n G$, and furthermore we simply write $\rhd G$ when $\emptyset \rhd G$. The first four clauses of the definition directly encode the introduction rules of a sequent calculus for this logic. Terms of type *expr* are built on an underlying de Bruijn syntax. The use of the proper annotation rules out terms that have occurrences of bound variables that do not have a corresponding binder (*dangling indices*).[2] In the last two rules, atoms are provable either by assumption or via *backchaining* over a set of Prolog-like rules, which encode the properties of the object-language. The notation $A \longleftarrow G$ represents an instance of one of the clauses of this definition. The sequent calculus is parametric in those clauses.

A small set of structural rules of the SL is proved, and used to reason about object-languages. We prefix theorems formalized in Hybrid with "H-."

H-Theorem 5 (Structural Properties)
(*a*) *Height weakening:* $[\![\, \Gamma \rhd_n G;\ n < m \,]\!] \to \Gamma \rhd_m G$
(*b*) *Context weakening:* $[\![\, \Gamma \rhd_n G;\ \Gamma \subseteq \Gamma' \,]\!] \to \Gamma' \rhd_n G$
(*c*) *Atomic cut:* $[\![\, A, \Gamma \rhd G;\ \Gamma \rhd \langle A \rangle \,]\!] \to \Gamma \rhd G$

Formalization of Lambda-Terms and Declarative and Algorithmic Equality. To represent the object-language, we fill in the definition of *con*, define new operators app and lam using the operators defined earlier for *expr*, and fill in the definition of *atm*, which includes the is_tm relation for well-formedness of terms as well as eq and equal. The inference rules are inductively defined using $(\_ \longleftarrow \_)$.

[2] Hybrid 0.2 described in [10] includes an improvement that doesn't require the proper predicate, but the proofs in this paper are not yet ported to the new version.

$$\begin{array}{rcl}
\textit{inductive con} & := & cAPP : con \mid cLAM : con \\
\mathsf{app}\ M_1\ M_2 & == & (\mathsf{APP}\ (\mathsf{APP}\ (\mathsf{CON}\ cAPP)\ M_1)\ M_2) \\
\mathsf{lam}\, x.\, M\ x & == & (\mathsf{APP}\ (\mathsf{CON}\ cLAM)\ (\mathsf{LAM}\ (\lambda x.\, M\ x))
\end{array}$$

$$\begin{array}{lrl}
\textit{inductive atm} := \mathsf{is\_tm} : expr \rightarrow atm \mid \mathsf{eq}, \mathsf{equal} : expr \rightarrow expr \rightarrow atm & & \\
\textit{inductive}\ \_ \longleftarrow \_ : atm \rightarrow oo \rightarrow Prop := & & \\
tm\_lam : & [\![\ \mathsf{abstr}\, T\ ]\!] & \rightarrow \mathsf{is\_tm}\ (\mathsf{lam}\, x.\, Tx) \longleftarrow \mathsf{all}\, x.\, (\mathsf{is\_tm}\ x)\ \mathsf{imp}\ \langle \mathsf{is\_tm}\ (Tx) \rangle \\
tm\_app : & & \rightarrow \mathsf{is\_tm}\ (\mathsf{app}\ T_1\ T_2) \longleftarrow \langle \mathsf{is\_tm}\ T_1 \rangle\ \mathsf{and}\ \langle \mathsf{is\_tm}\ T_2 \rangle \\
eq\_lam : & [\![\ \mathsf{abstr}\, E;\ \mathsf{abstr}\, F\ ]\!] & \rightarrow \mathsf{eq}\ (\mathsf{lam}\, x.\, Ex)\ (\mathsf{lam}\, x.\, Fx) \longleftarrow \\
& & \quad \mathsf{all}\, x.\, (\mathsf{eq}\ x\ x)\ \mathsf{imp}\ \langle \mathsf{eq}\ (Ex)\ (Fx) \rangle \\
eq\_app : & & \rightarrow \mathsf{eq}\ (\mathsf{app}\ E_1\ E_2)\ (\mathsf{app}\ F_1\ F_2) \longleftarrow \\
& & \quad \langle \mathsf{eq}\ E_1\ F_1 \rangle\ \mathsf{and}\ \langle \mathsf{eq}\ E_2\ F_2 \rangle \\
e\_l : & [\![\ \mathsf{abstr}\, T;\ \mathsf{abstr}\, T'\ ]\!] & \rightarrow \mathsf{equal}\ (\mathsf{lam}\, x.\, Tx)\ (\mathsf{lam}\, x.\, T'x) \longleftarrow \\
& & \quad \mathsf{all}\, x.\, (\mathsf{is\_tm}\ x)\ \mathsf{imp}\ (\mathsf{equal}\ x\ x)\ \mathsf{imp}\ \langle \mathsf{equal}\ (Tx)\ (T'x) \rangle \\
e\_a : & & \rightarrow \mathsf{equal}\ (\mathsf{app}\ T_1\ T_2)\ (\mathsf{app}\ S_1\ S_2) \longleftarrow \\
& & \quad \langle \mathsf{equal}\ T_1\ S_1 \rangle\ \mathsf{and}\ \langle \mathsf{equal}\ T_2\ S_2 \rangle \\
e\_r : & & \rightarrow \mathsf{equal}\ T\ T \longleftarrow \langle \mathsf{is\_tm}\ T \rangle \\
e\_t : & & \rightarrow \mathsf{equal}\ T\ S \longleftarrow \langle \mathsf{equal}\ T\ R \rangle\ \mathsf{and}\ \langle \mathsf{equal}\ R\ S \rangle
\end{array}$$

The well-formedness clauses *tm_lam* and *tm_app* are required since Hybrid terms are untyped (all object-level terms have type *expr*). Each of the remaining clauses of the inductive definition is given the same name as the corresponding rule in the Twelf and Beluga encoding. Note that they are quite similar; the differences in the encodings include 1) the abstr conditions used to rule out meta-level functions that do not encode object-level syntax, and (2) the appearance of is_tm in the *e_l* and *e_r* clauses, which are required to prove *adequacy* of the encoding (see [5] for a fuller discussion of adequacy of Hybrid encodings). In particular, we prove:

$$\rhd\ \langle \mathsf{eq}\ T\ S \rangle \rightarrow \rhd\ \langle \mathsf{is\_tm}\ T \rangle \wedge \rhd\ \langle \mathsf{is\_tm}\ S \rangle$$
$$\rhd\ \langle \mathsf{equal}\ T\ S \rangle \rightarrow \rhd\ \langle \mathsf{is\_tm}\ T \rangle \wedge \rhd\ \langle \mathsf{is\_tm}\ S \rangle.$$

Formalization of Completeness for Algorithmic Equality. In place of classifying contexts using context schemas or worlds declarations, we adopt the notion of a *context invariant.* This notion is informal; since we have an expressive logic at our disposal, we can define any predicate on contexts. We present one approach and briefly discuss a second one. In the first, we have three context invariants, one each for the proofs of reflexivity, transitivity, and completeness.

$$\begin{array}{rcl}
\mathsf{ref\_inv}\ \Phi\ \Psi & == & (\forall x.\ \mathsf{is\_tm}\ x \in \Phi \rightarrow \mathsf{eq}\ x\ x \in \Psi) \\
\mathsf{tr\_inv}\ \Psi & == & (\forall x\ y.\ \mathsf{eq}\ x\ y \in \Psi \rightarrow x = y) \\
\mathsf{ceq\_inv}\ \Phi\ \Psi & == & \mathsf{ref\_inv}\ \Phi\ \Psi \wedge \mathsf{tr\_inv}\ \Psi \wedge (\forall x\ y.\ \mathsf{equal}\ x\ y \in \Phi \rightarrow \mathsf{eq}\ x\ y \in \Psi)
\end{array}$$

Context invariants are used for two purposes here: 1) to represent how two contexts in different judgments are related (e.g., ref_inv), and 2) to represent information contained in the Σ-type groupings found in the `block` declarations in Beluga and Twelf (e.g., tr_inv). If we include enough information, as we do here, no weakening or strengthening is needed in the completeness proof. Instead, the following property is needed.

H-Lemma 6 (Context Extension)
$\mathsf{ceq\_inv}\ \Phi\ \Psi \rightarrow \mathsf{ceq\_inv}\ (\mathsf{equal}\ x\ x, \mathsf{is\_tm}\ x, \Phi)\ (\mathsf{eq}\ x\ x, \Psi)$

We now state the reflexivity and completeness theorems, and discuss the proof of the completeness theorem.

H-Theorem 7 (Reflexivity). $[\![\ \mathsf{ref\_inv}\ \Phi\ \Psi;\ \Phi \rhd_n \langle \mathsf{is\_tm}\ T\rangle\ ]\!] \to \Psi \rhd_n \langle \mathsf{eq}\ T\ T\rangle$

In addition to being necessary for adequacy, well-formedness definitions provide a convenient form of induction, which is used to prove the above theorem.

H-Theorem 8 (Completeness)
$[\![\ \mathsf{ceq\_inv}\ \Phi\ \Psi;\ \Phi \rhd_n \langle \mathsf{equal}\ T\ S\rangle\ ]\!] \to \Psi \rhd_n \langle \mathsf{eq}\ T\ S\rangle$

The proof is by induction on n with induction hypothesis:

$$IH == [\![\ i < n;\ \mathsf{ceq\_inv}\ \Phi\ \Psi;\ \Phi \rhd_i \langle \mathsf{equal}\ T\ S\rangle\ ]\!] \to \Psi \rhd_i \langle \mathsf{eq}\ T\ S\rangle.$$

A derivation of $\Phi \rhd_n \langle \mathsf{equal}\ T\ S\rangle$ must end in an application of the last two clauses of the definition of the SL (*s_init* or *s_bc*, page 237). In the *s_init* case (the assumption from context case), we know that $(\mathsf{equal}\ T\ S) \in \Phi$. By the definition of ceq_inv, we know that $(\mathsf{eq}\ T\ S) \in \Psi$. We use this fact and simply apply *s_init* to obtain $\Psi \rhd_n \langle \mathsf{eq}\ T\ S\rangle$, as desired.

When the derivation ends in *s_bc*, it must be the case that one of the four clauses defining declarative equality (page 238) was used. We consider reflexivity (*e_r*) and abstraction (*e_l*). In the former, we know that $T = S$ and $\Phi \rhd_{n-1} \langle \mathsf{is\_tm}\ T\rangle$. By H-Theorem 7, we can conclude $\Psi \rhd_{n-1} \langle \mathsf{eq}\ T\ T\rangle$ and by H-Theorem 5(a), that $\Psi \rhd_n \langle \mathsf{eq}\ T\ T\rangle$.

In the abstraction case (*e_l*), we know that T and S have the form $(\mathsf{lam}\, x.\, T'x)$ and $(\mathsf{lam}\, x.\, S'x)$, respectively, and we must show:

$$\begin{array}{l}[\![\ IH;\ \mathsf{ceq\_inv}\ \Phi\ \Psi;\ \Phi \rhd_n \langle \mathsf{equal}\ (\mathsf{lam}\, x.\, T'x)\ (\mathsf{lam}\, x.\, S'x)\rangle\ ]\!]\\ \quad \to \Psi \rhd_n \langle \mathsf{eq}\ (\mathsf{lam}\, x.\, T'x)\ (\mathsf{lam}\, x.\, S'x)\rangle\end{array}$$

By repeated inversion of the SL rules on the last premise, and repeated backward application of these rules to the conclusion, we obtain:

$$\begin{array}{l}[\![\ IH;\ \mathsf{ceq\_inv}\ \Phi\ \Psi;\ \mathsf{proper}\, x;\ (\mathsf{equal}\ x\ x, \mathsf{is\_tm}\ x, \Phi) \rhd_{n-4} \langle \mathsf{equal}\ (T'x)\ (S'x)\rangle\ ]\!]\\ \quad \to (\mathsf{eq}\ x\ x, \Psi) \rhd_{n-3} \langle \mathsf{eq}\ (T'x)\ (S'x)\rangle\end{array}$$

We can conclude $\mathsf{ceq\_inv}\ (\mathsf{equal}\ x\ x, \mathsf{is\_tm}\ x, \Phi)\ (\mathsf{eq}\ x\ x, \Psi)$ by H-Lemma 6 applied to the second premise, and then apply the induction hypothesis to obtain:

$$\begin{array}{l}[\![\ IH;\ \ldots;\ (\mathsf{eq}\ x\ x, \Psi) \rhd_{n-4} \langle \mathsf{eq}\ (T'x)\ (S'x)\rangle\ ]\!]\\ \quad \to (\mathsf{eq}\ x\ x, \Psi) \rhd_{n-3} \langle \mathsf{eq}\ (T'x)\ (S'x)\rangle\end{array}$$

which is provable directly by an application of H-Theorem 5(a).

We can also prove this theorem using a generalized context as is done in Twelf and Beluga. Using this alternate approach, we have only one context, so the context invariant no longer needs to express relationships between different contexts; now it only needs to contain the following information, which is also found in the `block` declarations in Beluga and Twelf.

$$\mathsf{ceq\_inv}'\ \Gamma == (\forall xy.\ \mathsf{eq}\ x\ y \in \Gamma \rightarrow x = y) \wedge (\forall xy.\ \mathsf{equal}\ x\ y \in \Gamma \rightarrow x = y).$$

Using this approach, we must also explicitly define weakening and strengthening functions on contexts, and lemmas about them. Such functions and lemmas depend on the object-language, but the lemmas are fairly easily proved using H-Theorem 5(b), which is the general weakening theorem of the SL. The reasoning required to prove this new version of H-Theorem 8 is similar to before, though slightly complicated by the need to explicitly apply the weakening and strengthening lemmas.

5 Criteria for Comparison

In this section we compare the approach taken in the three systems considered in this paper. More generally, we describe a list of questions which can be used to quantitatively compare systems and highlight their differences.

How do we represent contexts in proofs?. Beluga supports explicit contexts when implementing proofs about LF objects. Context variables allow us to abstract over concrete contexts and the structure of contexts is defined by context schemas. Σ-types tie different declarations together. While Beluga shares the general ideas regarding representing and reasoning about contexts with the Twelf system, it makes the meta-theoretic reasoning about contexts which is hidden in Twelf explicit. In Twelf, the actual context of hypotheses remains implicit. In Hybrid, contexts are explicitly modelled using lists or sets in the SL, but do not appear in the specification of the inference rules of the object-language in the inductive definition of $(\_ \longleftarrow \_)$.

How do we reason with contexts?. Reasoning with contexts is particularly important when reasoning about the relationship between two formal systems (such as the equality example) and when we assemble larger proofs using lemmas. Systems like Twelf and Beluga also support structural reasoning about contexts; for example, weakening is supported by the underlying typing rules and context subsumption. This built-in support is sensitive to the ordering of elements in a context (or world) schema and may require explicit weakening as in the implementation of the completeness proof for equality in Twelf. In Hybrid, H-Theorem 5 supports simple reasoning about contexts. This theorem is carried out once and for all at the SL level, and reused for every object-language. More complicated reasoning about weakening and strengthening can be avoided in Hybrid by expressing relationships between contexts in different judgments. The trade-off is that we must define these relationships explicitly as context invariants and prove context extension lemmas. The meta-logic, however, provides considerable flexibility in expressing such invariants. In fact, as discussed earlier, we can express them so that they use generalized contexts and lead to proofs that follow the corresponding Twelf and Beluga proofs quite closely. Doing so requires explicit weakening and strengthening lemmas that are specific to the object-language. Much of the reasoning that uses these kinds of lemmas is stereotyped and could be automated (although we have not yet done so).

How do we retrieve elements from a context?. As the context is implicit in Twelf and the user has no access to it, the user cannot pattern match on elements from it directly. Instead of generic cases which pattern match on an element from the context, base cases in proofs are handled whenever an assumption is introduced, and in fact are treated as part of the context. This may lead to scattering of base cases and redundancy, and in addition complicates reasoning about contexts. In Beluga, retrieval is supported using parameter variables and projections. This is in fact crucial in the reflexivity `ref` and in the completeness proof `ceq`, where we use Σ-types to tie assumptions together and use projections on a parameter variable to retrieve different parts. Since the context is explicit in the SL level in Hybrid, when retrieval of elements is needed in the base case and other cases, it is done via simple list operations such as membership. The Coq libraries provide support for using such operations in proofs.

How easy is it to state a given theorem?. The examples in sections 2.2 and 2.3 provide a wide range of statements. All discussed systems provide a two-level approach. However, the level which allows reasoning about formal systems is more expressive in Beluga and in Hybrid. These systems provide direct representations of the given theorems. Twelf's meta-logic which is used to verify that a given relation constitutes a proof is not rich enough to handle nested quantification and implications directly. One solution is to implement an assertion logic which is then used to encode the actual theorem [18].

How do we apply a substitution lemma?. In all known systems supporting HOAS encodings substitution lemmas come for "free." While the examples in this paper do not make use of the substitution lemma, there are several well-known examples such as type preservation for MiniML. In the Twelf system and in Beluga, applying the substitution lemma is reduced to the substitution operation in the underlying logical framework. In Hybrid, the substitution lemma corresponds to the application of the SL cut-rule, expressed as H-Theorem 5(c).

How do we know we have implemented a proof?. In a system such as Hybrid, we simply need to trust the underlying proof assistant and establish adequacy. In general, proofs proceed by induction on the definition of the SL with a sub-induction on the object-language. Coq provides extensive support for inductive reasoning, and the induction hypothesis is a premise that is applied explicitly when needed.

In systems such as Twelf or Beluga, we need to establish separately that the user implemented a total function. Twelf is a mature system and provides a coverage checker which in turn relies on the world declarations to ensure the base cases are covered. In addition, the termination checker verifies that a given relation is terminating according to a structural ordering specified by the user. This establishes that all appeals to the induction hypothesis are valid.

Beluga essentially adopts the same philosophy, although the current release does not include a coverage and termination checker. The theoretical foundation for coverage in Beluga is described in [4] and an implementation is planned for the future. Intuitively, pattern matching on a contextual object of type $A[\Psi]$ is exhaustive if we cover all constructors of type A plus the cases described by parameter variables, which cover the possibility that we have used an assumption from the context Ψ. For termination checking, we believe the ideas from [12] can be easily adapted.

In general, writing cases using pattern matching by hand may result in a more compact proof since it provides a flexible way to write fall-through patterns or to simultaneously match on several objects. Hence, we may get away with writing fewer cases explicitly as compared to an interactive prover.

How easy is it to interface the system with, for example, support for natural numbers?. In the example that counts variable occurrences, reasoning about natural numbers may be necessary and useful. Twelf and Beluga's reasoning infrastructure does not support them and hence properties like addition and the totality of addition must be proven separately. This leads to some overhead in the actual proofs. Hybrid, on the other hand, relies heavily on the theorem prover's built in data-type for natural numbers along with a large collection of lemmas and automated proof procedures (such as *omega* in Coq).

6 Conclusion

We presented several benchmark problems together with a general set of criteria for comparing reasoning systems which support the mechanization of formal systems. In addition, we discussed in detail the proofs of one of these problems in three systems (Beluga, Twelf, and Hybrid), and applied these criteria to compare them. This work is a starting point that will help users and developers to evaluate proof assistants which mechanize the reasoning about formal systems. It will also facilitate a better understanding of the differences between and limitations of these systems, as well as the impact of these design decisions in practice. This will provide guidance for users and stimulate discussion among developers.

We hope that these problems will subsequently also be implemented in systems using related approaches. In particular, the Delphin System [16] seems to lie between the three systems discussed in this paper. Similarly, it would be interesting to compare systems such as Abella as well as approaches not relying on HOAS encodings such as nominal encodings.

References

1. Aydemir, B., et al.: Mechanized metatheory for the masses: The POPLmark challenge. In: Hurd, J., Melham, T.F. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005)
2. Baelde, D., Snow, Z., Miller, D.: Focused inductive theorem proving. In: Giesl, J., Hähnle, R. (eds.) 5th International Joint Conference on Automated Reasoning (IJCAR'10). LNCS (LNAI), Springer, Heidelberg (forthcoming, 2010)

3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2004)
4. Dunfield, J., Pientka, B.: Case analysis of higher-order data. In: LFMTP'08. Electr. Notes in Theor. Comput. Sci, vol. 228, pp. 69–84 (2009)
5. Felty, A.P., Momigliano, A.: Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. CoRR, abs/0811.4367 (2008)
6. Gacek, A.: The Abella interactive theorem prover (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 154–161. Springer, Heidelberg (2008)
7. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM 40(1), 143–184 (1993)
8. Harper, R., Licata, D.R.: Mechanizing metatheory in a logical framework. Journal of Functional Programming 17(4-5), 613–673 (2007)
9. McDowell, R.C., Miller, D.A.: Reasoning with higher-order abstract syntax in a logical framework. ACM Transactions on Computational Logic 3(1), 80–136 (2002)
10. Momigliano, A., Martin, A.J., Felty, A.P.: Two-level Hybrid: A system for reasoning using higher-order abstract syntax. In: LFMTP'07. Electr. Notes Theor. Comput. Sci, vol. 196, pp. 85–93 (2008)
11. Nipkow, T., Paulson, L.C., Wenzel, M. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
12. Pientka, B.: Verifying termination and reduction properties about higher-order logic programs. Journal of Automated Reasoning 34(2), 179–207 (2005)
13. Pientka, B.: 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), pp. 371–382. ACM Press, New York (2008)
14. Pientka, B., Dunfield, J.: Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description). In: Giesl, J., Hähnle, R. (eds.) 5th International Joint Conference on Automated Reasoning (IJCAR'10). LNCS (LNAI). Springer, Heidelberg (2010)
15. Pfenning, F., Schürmann, C.: System description: Twelf — a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
16. Poswolsky, A.B., Schürmann, C.: Practical programming with higher-order encodings and dependent types. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 93–107. Springer, Heidelberg (2008)
17. Schürmann, C.: Automating the Meta Theory of Deductive Systems. PhD thesis, Department of Computer Science, Carnegie Mellon University. CMU-CS-00-146 (2000)
18. Schürmann, C., Sarnat, J.: Structural logical relations. In: 23rd Annual Symposium on Logic in Computer Science (LICS), pp. 69–80. IEEE Computer Society, Los Alamitos (2008)