HAL
open science

Computation-as-deduction in Abella: work in progress

Kaustuv Chaudhuri, Ulysse Gérard, Dale Miller

To cite this version:

Kaustuv Chaudhuri, Ulysse Gérard, Dale Miller. Computation-as-deduction in Abella: work in progress. 13th international Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, Jul 2018, Oxford, United Kingdom. hal-01806154

HAL Id: hal-01806154

https://inria.hal.science/hal-01806154v1

Submitted on 1 Jun 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Computation-as-deduction in Abella: work in progress

Kaustuv Chaudhuri Ulysse Gérard Dale Miller
Inria-Saclay & LIX, Ecole Polytechnique

The Abella theorem prover is based on a logic in which relations, and not functions, are defined by induction (and coinduction). Of course, many relations do, in fact, define functions and there is real value in separating functional computation (marked by determinism) from more general deduction (marked by nondeterminism and backtracking). Recent work on focused proof systems for the logic underlying Abella is used in this paper to motivate the design of various extensions to the Abella system. With these extensions to the system (which do not extend the logic), it is possible to fully automate functional computations within the relational setting as soon as a proof is provided that a given relation does, in fact, capture a total function. In this way, we can use Abella to compute functions even when data structures contain bindings.

1 Introduction

Currently, the Abella proof assistant has rather limited forms of automation. An interesting framework for exploring possible means of adding more automation to Abella is to take inspiration from focusing [1, 2, 9]. The basic idea of focusing is to control and reduce the non-determinism in proof search in Gentzen-style sequent calculi. This is achieved by composing the ordinary sequent rules that operate on single connectives at a time into compound rules that work on a collection of connectives, called *synthetic connectives*, that have similar properties. In the presence of inductive and coinductive definitions, such synthetic connectives can involve the unbounded unfolding of fixed points, thereby incorporating arbitrary deterministic and nondeterministic computation within synthetic inference rules.

Full focused proof search as a broad basis for the automation of Abella is an interesting project, but in this paper we explore a limited application of focusing to recover *computation*. It has been argued [8], that focusing in an intuitionistic logic with fixed points (essentially, Heyting arithmetic) can be used to turn *relational* specifications into *functional* computations. We present a concrete proposal for a slight and orthogonal extension of Abella that allows it to perform such computations without any change to its underlying logical basis.

2 Background

In this section we will present the logic and the proof system used to perform inductive and coinductive relational reasoning. It is a summary of the more expansive description that can be found in a sequence of papers published over the last decade [6, 7, 3].

2.1 The $\mathscr{G}$ Logic and the Abella Implementation

The Abella system implements two logics. The *specification logic* is a simple fragment of intuitionistic logic that is rich enough to specify many λProlog logic programs. This aspect of Abella is not the major concern of this progress report and we ignore it here. In this paper we concentrate on the *reasoning* logic of Abella, known as $\mathscr{G}$ [6], which is an extension of intuitionistic first-order logic with: (1) higher-order

Submitted to:
LFMTP 2018

© Chaudhuri, Gérard, & Miller
This work is licensed under the
Creative Commons Attribution License.

λ-terms together with the equational theory induced by $\alpha\beta\eta$-equivalence, (2) inductive and coinductive fixed point definitions, and (3) nominals, nominal abstraction, and generic (∇) quantification. We give a brief introduction to $\mathscr{G}$ using the concrete syntax of Abella [3].

The *terms* in $\mathscr{G}$ are well-typed terms of Church's simple theory of types [5], where a given *type signature* declares a collection of basic types and constants that are interpreted as constructors for these declared basic types. For instance, the following is a declaration of two basic types, `nat` and `bool`, that are both declared to be types using the `Kind` keyword, and their constructors are indicated with `Type` declarations.

```
Kind bool type.
Type tt, ff  bool.

Kind nat type.
Type z nat.
Type s nat → nat.
```

Formulas of $\mathscr{G}$ are terms of type `prop`, built from the constructors $\wedge$ (for conjunction), $\vee$ (for disjunction), and $\rightarrow$ (for implication), all of type `prop → prop → prop` and written as infix; `true` and `false` of type `prop` for the constants; and `forall`, `exists`, and `nabla` of type `(`α `→ prop) → prop` (for every type α not containing `prop`). The term abstraction λ`x.t` is written concretely as `x\ t`, and quantified formulas are written in a more natural style rather than using abstractions, i.e., as `(forall x, f)` instead of `forall (x\ f)`.

Atomic formulas can be created from predicates of target type `prop` that may be declared with a `Type` declaration. More interestingly, $\mathscr{G}$ also allows atomic formulas to be built using inductively or coinductively defined fixed points. For instance, the following inductive definition characterizes all terms of type `nat` built from `z` and `s`:

```
Define nat : nat → prop by
  nat z ;
  nat (s X) := nat X.
```

Such definitions consist of a list of clauses where each clause begins with a *head* and is optionally followed by a *body* separated by `:=`. (An omitted body is understood to stand for `true`.) The head is always atomic using the predicate being defined, but the body can be any arbitrary formula; moreover, the head and body can share *variables* that are universally quantified over the entire clause and written using capital letters. Thus, the way to read the second clause above is: *for every `X`, the atom `nat (s X)` holds if and only if `nat X` holds*.

Note that in $\mathscr{G}$ and Abella the only form of induction or coinduction is with such defined predicates. There is no induction principle for the types – indeed, there is no reasoning principle of any kind for the types. Types are just used to enforce syntactic categories. As a consequence, we cannot prove the formula `forall (X:nat), nat X`: when we want to prove a theorem by structural induction on natural numbers, we need to explicitly use the `nat` predicate as a corresponding assumption. To illustrate this, let us introduce the predicate `plus` that relates two numbers to their sum and proceeds by structural induction on its first argument.

```
Define plus : nat → nat → nat → prop by
  plus z X X ;
  plus (s X) Y (s Z) := plus X Y Z.
```

Here is a simple theorem that would need to be proved by structural induction on the first argument.

```
Theorem plus_z2 : forall X, nat X → plus X z X.
```

Such a theorem would be proved by means of the `induction` tactic. In this case, we would proceed by `induction on 1`, i.e., on the first antecedent of the chain of implications in the theorem. This would generate an *inductive hypothesis* `IH`:[1]

```
IH : forall X, (nat X)* → plus X z X
```

This is apparently the same as the theorem itself, except the inductive argument is marked with a size restriction `*`. The meaning of `(nat X)*` is that it can be applied to any derivation of `(nat X)` that is strictly smaller than that of the `(nat X)` we started the induction on originally. That original derivation is itself indicated with `(nat X)@`, which is to say that the result of the `induction` tactic is to change the goal to the following after assuming the `IH`.

```
forall X, (nat X)@ → plus X z X
```

This goal is proved by means of ordinary logical reasoning, together with the `case` tactic that explores all the ways in which an inductively defined assumption may have been derived, i.e., it performs an *inversion* on its definition. This `case` step in turn changes the `@` annotation to a `*` to indicate that it has strictly reduced the size of the derivation; this reduction makes the `IH` applicable. More precisely, inverting `(nat X)@` produces two subgoals; in the first, `X` is instantiated to `z`, and in the other `X` is instantiated to `s X1` for a new variable `X1`, and we get the additional assumption `(nat X1)*`. Abella also has a collection of lower level tactics such as `unfold`, `witness`, `split`, `apply`, etc. for ordinary logical reasoning.

2.2 Focusing: Synchronous and Asynchronous Rules

The `search` tactic of Abella tries to find a derivation for a given formula as a goal, unfolding the definitions of defined atoms as needed and guessing existential witnesses using unification. By default this tactic only applies rules on the goal ("below the line") and limits searches using a strict bound. This is clearly far from complete and is done primarily because the unconstrained search space for proofs is wild and unpredictable, particularly in the presence of induction. Nevertheless, it is possible to identify conditions where automated search can be improved without negatively affecting the structure of proofs. One well known technique is *focusing*, which states that the search for proofs can be organized into alternations of *synchronous* and *asynchronous* phases. The proof-theoretic basis for focusing can be found in a number of places [1, 2, 9] and will not be elaborated any further in this work. Instead, we will explain focusing in terms of the Abella implementation.

Roughly speaking, the asynchronous phase corresponds to invertible rules of the sequent calculus, which are rule applications that are guaranteed not to affect the derivability of the goal. Examples of such rules in the goal include the `intros` tactic that introduces variables and assumptions, the `split` tactic that divides a conjunctive goal into a collection of subgoals, one per conjunct. Asynchronous rules can also exist on formulas in the context: for examples, equality assumptions that can be fully solved may be eagerly inverted by instantiating eigenvariables using the most general solutions, recursively defined predicates can be unfolded, existential assumptions can be simplified by introducing new eigenvariables, and disjunctive assumptions can be simplified by generating additional subgoals for each disjunct. The tactic `case` performs only such invertible rules on a designated hypothesis.

A synchronous phase, in contrast, requires the user to indicate choices in the proof. Such choices include selection of a particular disjunct in a disjunctive goal, giving the witness terms for an existential goal, showing how to instantiate the variables of a universally quantified assumption, or inventing and

[1] The parentheses we use here for didactic reasons are omitted by Abella, i.e., Abella writes `nat X*` instead of `(nat X)*`.

using lemmas. Another important example of a synchronous rule is unfolding of definitions for defined atoms in the goal, which generally involves a choice of definitional clause. The crucial feature of the synchronous phase is that once the phase is entered, the proof is *focused* on a particular formula which is then used to indicate followup synchronous rules, maintaining focus as long as possible. This drastically reduces the choice points in the proof, since after a focus has been *decided* the choices are constrained to those relevant to the focused formula. From the perspective of Abella, the synchronous phase is mainly relevant for indicating expressive bounds for `search`: instead of considering the search depth one connective at a time, we can use a bound on the number of times a focus can be decided on, usually called the *decide depth*.

An important feature of phases is that they can encompass entire computations for inductive definitions whose definitional clauses use only connectives of a single polarity. For instance, if a definition uses only positive connectives (`=`, $\wedge$, $\vee$, `true`, `false`, and `exists`), then whenever it appears as a *ground* assumption it can be completely discharged within a single asynchronous phase. Dually, if it uses only negative connectives ($\wedge$, `true`, $\rightarrow$, and `forall`) and appears as a goal, it can be similarly discharged within a single asynchronous phase.[2]

Consider, for example, the definition of `plus` from Section 2.1. It contains only positive connectives. As a result, a closed `plus`-atom as an assumption can be discharged completely within one asynchronous phase. For example, the assumption `(plus (s z) (s z) (s (s z)))` can be removed since it is recognized as trivially true and the assumption `(plus (s z) (s (s z)) (s z))` would lead to a complete proof of the goal since it is recognized as false. Similarly, the assumption `(plus (s z) (s z) X)`, for an eigenvariable `X`, can be *solved*—i.e., removed and the variable `X` instantiated—because the only thing that `X` can be is `(s (s z))`.

3 Proposal: Computation and Suspension

Our first proposed extension of Abella is rather simple: the addition of a `compute` tactic that performs unfolding and subsequent asynchronous steps for assumptions involving predicates with a fully positive definition. Thus, for instance, if we have an assumption

```
H : plus (s z) (s z) X
```

then the invocation `compute H` would repeatedly unfold the definition of `plus` and handle the resulting subgoals eagerly if it can use purely asynchronous steps. In this particular case, the effect will be the removal of `H` entirely and the instantiation of `X` with `(s (s z))`. The `compute` tactic is allowed to produce multiple branches. For instance, in the following case:

```
H : plus X Y (s (s z))
```

the invocation `compute H` would produce three subgoals, one each for the three ways there are to divide 2 into two natural numbers.

This kind of feature has long been recognized as an important need in Abella.[3] A very common form is encountered in meta-theoretic proofs involving memberships in contexts, which are represented as lists in Abella, where we have an assumption such as:

[2]Interestingly, $\wedge$ and `true` can be seen as belonging to both the positive and the negative classes, since there is a fundamental ambiguity about their polarity. The positive interpretation corresponds to the rules for $\otimes$ and 1 from intuitionistic linear logic, while the negative interpretation corresponds to those of & and $\top$. In most formulations of focusing for intuitionistic logic (e.g., in LJF [9]), this ambiguity is resolved by means of a *polarity assignment*, which ascribes a specific polarity to each occurrence of $\wedge$ and `true`.

[3]See, for instance, `https://github.com/abella-prover/abella/issues/35`.

```
H : member X (E1 :: E2 :: Rest)
```

In this case we would like `compute H` to yield three subgoals: the first with `X = E1`, the second with `X = E2`, and the last with `member X Rest`.

A more interesting scenario is when the `compute` tactic is used on a purely positive predicate that cannot be fully solved. For instance, given:

```
H : nat (s (s X))
```

where `X` is an eigenvariable, it can be asynchronously simplified to `(nat X)` by just using the second clause of the definition of `nat`. However, to go further we would need to consider the cases where `X = z` and the case for `X = s X1`, and we would be left with a further assumption `nat X1`. We can repeat this process now with `X1` and so on. This *eager* treatment of `nat` not only leads to non-terminating search (which will eventually be forcefully terminating because it reaches a depth bound), but may be unwarranted before we know anything else about `X`. In this case, it would be useful to *suspend* the eager unfolding of `nat`.

To account for this premature unfolding of definitions when the inductive structure is already a variable, we add a new kind of `Suspend` declaration that will make Abella stop the asynchronous phase prematurely. The following declaration declares that `(nat X)` should not be unfolded if `X` is a variable; we call this a *suspension condition*.

```
Suspend nat X on X.
```

A suspension condition can list more than one argument: unfolding is suspended if any of the indicated arguments is a variable. For example:

```
Suspend plus X Y _ on X, Y.
```

Note this declaration means that `compute` would terminate early even on a situation such as:

```
H : plus (s z) Y Z
```

even though we could have finished the phase with `Z` instantiated with `(s Y)`, even though `Y` isn't ground. This is fine because we could have left out the `Y` from the suspension conditions. Also note that although the suspension condition mentions variables, the suspension declaration itself can be any arbitrary pattern. For instance:

```
Suspend plus (s X) _ _ on X.
```

suspends unfolding on `plus` before its first argument is a variable. The pattern can also have repetitions such as:

```
Suspend plus X X _ on X.
```

Finally, a given predicate can have multiple suspension declarations: unfolding is suspended if any suspension declaration matches or if the predicate has no suspension declarations at all. The following pair is equivalent to the first `Suspend` declaration above.

```
Suspend plus X _ _ on X.
Suspend plus _ Y _ on Y.
```

The `compute` tactic has been implemented in prototype form already in Abella—which in fact led to the discovery of the need for `Suspend`—but the full proposal is still being debated. One obvious extension would be to perform the asynchronous phase on *all* assumptions instead of a specific one. Another issue to consider is whether we should allow `compute` to operate on goals as well. What would be the equivalent notion of `Suspend` for goals? Another open question is if the `Suspend` declarations can be inferred from the form of the definition itself.

4 Proposal: Deterministic Computation using Singleton Predicates

A monadic predicate p that holds for exactly one argument is a *singleton*. Singletons are interesting from the perspective of focusing. The formula `forall x, p x → Q x` and `exists x, p x ∧ Q x` are equivalent if and only if p is a singleton. That is, the following is a theorem of higher-order logic:

```
(forall q, ((forall x, p x → q x) ≡ (exists x, p x ∧ q x)))
   ≡ singleton p
```

where `singleton` has the following definition:

```
Define singleton : (A → prop) → prop by
  singleton P :=
      (exists X, P X)
    ∧ (forall X Y, P X → P Y → X = Y).
```

As a consequence, the formulas `forall x, p x → Q x` and `exists x, p x ∧ Q x` may be freely converted into each other in the course of proof search.

Now, since Abella is a first-order logic, the definition and theorem above are not acceptable. The theorem is explicitly ruled out because Abella does not allow universal quantification over terms whose types contain `prop`. The definition is accepted with a *stratification warning*, because the higher-order parameter P is used in a negative position, to the left of →. Such definitions can be used in trivial ways to prove `false` and hence for consistency Abella refuses to certify developments using such definitions.

Both the proof theory of sequent calculus and the tactics of Abella require that to make progress on proving `exists x, p x ∧ Q x`, we must first supply a witness term `t` such that `(p t)` is true, and then the goal can become `(Q t)`. If we know that p is a singleton, then this requirement is unfortunate since one might hope that we could use Abella to actually *compute* this witness term `t` by means of the `compute` tactic in the previous section. It is tempting to extend Abella with *logic variables* or *placeholder variables* such as `?X` so that we can change the query to `p ?X ∧ Q ?X`, and then in the course of proving the first conjunct `p ?X` we would replace the variable with the witness term. Such variables have been a part of Isabelle and Agda from the very beginning and have also been introduced to Coq (see, for example, [12]).

We propose here a more lightweight treatment by admitting the definition `singleton` to Abella (and syntactically preventing its abuse such as applying `singleton` to itself). Then, the issue of computing the witness term `t` is no different from transforming the goal `exists x, p x ∧ Q x` to `forall x, p x → Q x`, introducing the variable and its hypothesis (using `intros`), and then using `compute` on that hypothesis. Thus, we switch from "guess `t` and check `(p t)`" to "compute the `t` for which `(p t)`."

Singleton predicates arise whenever a relation is actually a function. In particular, the fact that an n-ary predicate R actually specifies a function from its, say, first $n-1$ arguments to its nth argument can be captured by:

```
forall x1 x2 ... xn-1, singleton (x\ R x1 x2 ··· xn-1 x)
```

Note that we could have η-contracted the argument to `singleton` above to just `(R x1 ··· xn-1)`. More generally, the relation R may be a singleton only under certain conditions on its "input" arguments, in which case we would add them as antecedents in an implication chain. For example, consider the `plus` relation from before; its third argument is always uniquely determined by its first two, assuming that they are natural numbers. Hence, we can prove the following theorem.

```
Theorem plus_funct: forall X Y, nat X → nat Y → singleton (plus X Y).
```

This is an ordinary Abella theorem that can be readily proved by `induction on 1`. As another illustration, consider the partial relation `pred` for predecessors that relates natural numbers greater than 0 to their predecessor.

```
Define pred : nat → nat → prop by
  pred (s X) X.
```

To show that it is a function, we have to supply the precondition that its first argument is a natural number greater than `z`, which we can do as follows.

```
Define nat_gt : nat → nat → prop by
  nat_gt (s X) z := nat X ;
  nat_gt (s X) (s Y) := nat_gt X Y.

Theorem pred_funct: forall X, nat_gt X z → singleton (pred X).
```

To make use of `singleton` to convert between the two `exists` and `forall` forms, we add new tactic forms to `witness` and `apply`. When the goal has the form:

```
========================================
 exists X, P X ∧ Q X
```

then the invocation `witness compute` first attempts to prove `singleton P` from the same context, and then continues with modified goals of the form:

```
H : P X
========================================
 Q X
```

and follows up with `compute H`. Dually, whenever we have a hypothesis of the form:

```
H : forall X, P X → Q X
```

then an invocation `apply compute H` has the effect of first trying to prove `(singleton P)` and then continuing with the modified hypotheses

```
H1 : P X
H  : Q X
```

following up with `compute H1`.

In both these cases, the proof of `(singleton P)` must be trivial: the way it will be implemented is that the proved lemmas such as `plus_funct` will be searched for a predicate that matches `P`, and if so the antecedents of that lemma will be attempted to be proved with simple proofs. An important consideration in these simple proofs is that assumptions on predicates such as `nat` or `nat_gt` are attempted eagerly first to reduce them to their simplest forms. This will be done with the `compute` tactic as defined in Section 3. Note that it is important to supply suitable `Suspend` declarations for such antecedents to prevent infinite loops in the implicitly invoked `compute` invocations.

The above could have been done with a weaker assumption than `singleton`; it would have sufficed for the predicate `p` to be non-empty, which is just the first conjunct in the definition of `singleton`. The real power of the `singleton` assumption comes from the fact that it makes the computations deterministic. This means that whenever we perform `compute` on a singleton predicate, we never need to consider any but a single possibility. In other words, *conjunctive* branches in the search space caused by unfolding the singleton predicate can be pruned eagerly. To illustrate this, suppose we had the following variant definition of `plus`:

```
Define plus : nat → nat → nat → prop by
  plus z X X ;
  plus (s X) Y (s Z) := plus X Y Z ;
  plus X Y Z := plus Y X Z.
```

It is still a function from its first two to its third argument, but unfolding the definition of `plus` is not unitary: the third clause overlaps with the first two. The `compute` tactic should be satisfied with the first unfolding sequence it finds, and not get distracted computing variants that have different numbers of uses of the third clause.

5 Perspectives

We have proposed a small extension to Abella's tactics to enable it to *perform* deterministic computation without step-by-step guidance by the user. We leave the kernel and the core tactics of Abella untouched, but add a new `compute` tactic that is designed to perform the asynchronous phase of focused proof search for inductively defined predicates whose definitions are fully positive. Together with this mechanism is a new declaration that allows eager unfolding of definitions to be suspended when it is premature to continue unfolding, for instance where the arguments involve variables in "input" positions. Finally, we propose to allow Abella to express specific lemmas that prove that a given relation on a given collection of inputs determines a singleton on its output, meaning that the output both exists and is uniquely determined. Such lemmas can be used to transform an existential goal to a universal goal and move from a *guess and check* to a *compute and use* paradigm. A crucial feature of this use of singletons is that it treats computations as deterministic, meaning that any answer is as good as any other.

There are a few ways in which these proposals can be generalized further. First, the notion of singleton can be relaxed to a notion of *singleton up to equivalence*. For instance, we can say:

```
Define singleton_upto : (A → A → prop) → (A → prop) → prop by
  singleton_upto Eq P :=
      (exists X, P X)
    ∧ (forall X Y, P X → P Y → Eq X Y).
```

As long as `Eq` is an equivalence relation, we get all of the benefits of the `singleton` definition, such as the free conversion of `exists` goals into `forall` goals. This more general definition can be very useful in meta-theoretic proofs that reason about contexts: ordinarily they are represented as lists, but two contexts-as-lists that are merely permutations are considered to represent the same context. It has been observed in [4] that a majority of the effort in formalizing standard meta-theorems such as cut-elimination is due to the complications resulting from reasoning about lists up to permutations.

A second obvious extension has to do with data defined by higher-order type signatures, such as terms represented using λ-tree syntax (sometimes known as higher-order abstract syntax). Many common relations that are defined on such higher-order data can be seen as functions, but it takes a bit more care to use the `singleton` relations. In particular, with higher-order representations the "typing relation" such as `nat` are no longer a natural fit for the reasoning logic; in this case, it is much simpler to write these relations using the specification logic, using the *two-level logic approach* [7]. Recent extensions of Abella to handle the full hereditary Harrop specification language [11] have allowed the expression of arbitrary higher-order (and even dependently typed) relations in terms of the specification language (see, e.g.,[10] for the LF dependent type theory). In these cases, not only the antecedents but also the *argument* to the `singleton` relation may well be a specification-language sequent.

References

[1] Jean-Marc Andreoli (1992): *Logic Programming with Focusing Proofs in Linear Logic*. J. of Logic and Computation 2(3), pp. 297–347, doi:10.1093/logcom/2.3.297.

[2] David Baelde (2012): *Least and greatest fixed points in linear logic*. ACM Trans. on Computational Logic 13(1), doi:10.1145/2071368.2071370. Available at `http://tocl.acm.org/accepted/427baelde.pdf`.

[3] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu & Yuting Wang (2014): *Abella: A System for Reasoning about Relational Specifications*. Journal of Formalized Reasoning 7(2), doi:10.6092/issn.1972-5787/4650. Available at `http://jfr.unibo.it/article/download/4650/4137`.

[4] Kaustuv Chaudhuri, Leonardo Lima & Giselle Reis (2016): *Formalized Meta-Theory of Sequent Calculi for Substructural Logics*. In: Workshop on Logical and Semantic Frameworks, with Applications (LSFA). Post proceedings version to appear; Formalization `https://github.com/meta-logic/abella-reasoning`.

[5] Alonzo Church (1940): *A Formulation of the Simple Theory of Types*. J. of Symbolic Logic 5, pp. 56–68.

[6] Andrew Gacek, Dale Miller & Gopalan Nadathur (2011): *Nominal abstraction*. Information and Computation 209(1), pp. 48–73, doi:10.1016/j.ic.2010.09.004.

[7] Andrew Gacek, Dale Miller & Gopalan Nadathur (2012): *A two-level logic approach to reasoning about computations*. J. of Automated Reasoning 49(2), pp. 241–273, doi:10.1007/s10817-011-9218-1. Available at `http://arxiv.org/abs/0911.2993`.

[8] Ulysse Gérard & Dale Miller (2017): *Separating Functional Computation from Relations*. In Valentin Goranko & Mads Dam, editors: 26th EACSL Annual Conference on Computer Science Logic (CSL 2017), LIPIcs 82, pp. 23:1–23:17, doi:10.4230/LIPIcs.CSL.2017.23.

[9] Chuck Liang & Dale Miller (2009): *Focusing and Polarization in Linear, Intuitionistic, and Classical Logics*. Theoretical Computer Science 410(46), pp. 4747–4768, doi:10.1016/j.tcs.2009.07.041.

[10] Mary Southern & Kaustuv Chaudhuri (2014): *A Two-Level Logic Approach to Reasoning about Typed Specification Languages*. In Venkatesh Raman & S. P. Suresh, editors: 34th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), Leibniz International Proceedings in Informatics (LIPIcs) 29, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, New Delhi, India, pp. 557–569, doi:10.4230/LIPIcs.FSTTCS.2014.557. Available at `http://chaudhuri.info/papers/draft14abellf.pdf`.

[11] Yuting Wang, Kaustuv Chaudhuri, Andrew Gacek & Gopalan Nadathur (2013): *Reasoning about Higher-Order Relational Specifications*. In Tom Schrijvers, editor: Proceedings of the 15th International Symposium on Princples and Practice of Declarative Programming (PPDP), Madrid, Spain, pp. 157–168, doi:10.1145/2505879.2505889. Available at `http://chaudhuri.info/papers/draft13hhw.pdf`.

[12] Beta Ziliani & Matthieu Sozeau (2015): *A unification algorithm for Coq featuring universe polymorphism and overloading*. In Kathleen Fisher & John H. Reppy, editors: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015, ACM, pp. 179–191.