# Reasoning about computations using two-levels of logic

Dale Miller

# Reasoning about Computations Using Two-Levels of Logic

Dale Miller

INRIA Saclay & LIX, École Polytechnique
Palaiseau, France

**Abstract.** We describe an approach to using one logic to reason about specifications written in a second logic. One level of logic, called the "reasoning logic", is used to state theorems about computational specifications. This logic is classical or intuitionistic and should contain strong proof principles such as induction and co-induction. The second level of logic, called the "specification logic", is used to specify computation. While computation can be specified using a number of formal techniques—*e.g.*, Petri nets, process calculus, and state machines—we shall illustrate the merits and challenges of using logic programming-like specifications of computation.

## 1 Introduction

When choosing a formalism to use to specify computation (say, structured operational semantics, Petri nets, finite state machines, abstract machines, $\lambda$-calculus, or $\pi$-calculus), one needs that specification framework to be not only expressive but also amenable to various kinds of reasoning techniques. Typical kinds of reasoning techniques are algebraic, inductive, co-inductive, and category theoretical.

Logic, in the form of logic programming, has often been used to specify computation. For example, Horn clauses are a natural setting for formalizing structured operational semantics specifications and finite state machines; hereditary Harrop formulas are a natural choice for specifying typing judgments given their support for hypothetical and generic reasoning; and linear logic is a natural choice for the specification of stateful and concurrent computations. (See [27] for an overview of how operational semantics have been specified using the logic programming paradigm.) The fact that logic generally has a rich and deep meta-theory (soundness and completeness theorems, cut-elimination theorems, *etc*) should provide logic with powerful means to help in reasoning about computational specifications.

The activities of specifying computation and reasoning about those specifications are, of course, closely related activities. If we choose logic to formulate both of these activities, then it seems we must also choose between using one logic for both activities and using two different logics, one for each activity. While both approaches are possible, we shall focus on the challenges and merits of treating

these two logics as different. In particular, we shall assume that our "reasoning logic" formalizes some basic mathematical inferences, including inductive and co-inductive reasoning. On the other hand, we shall assume that our "specification logic" is more limited and designed to describe the evolution (unfolding) of computations. Speaking roughly, the reasoning logic will be a formalization of a part of mathematical reasoning while the specification logic will be a logic programming language.

This paper is a summary of some existing papers (particularly [16]) and is structured as follows. Section 2 presents a specific reasoning language $\mathcal{G}$ and Section 3 presents a specific specification logic $hH^2$. Section 4 describes how $hH^2$ is encoded in $\mathcal{G}$. Section 5 describes a few implemented systems that have been used to help explore and validate the intended uses of $hH^2$ and $\mathcal{G}$. Section 6 presents an overview of the various key ingredients of these two logics as well as suggesting other possibilities for them. Finally, Section 7 describes some related work.

## 2 The reasoning logic

Our reasoning logic, which we call $\mathcal{G}$ (following [14]) is a higher-order logic similar to Church's Simple Theory of Types [9] (axioms 1 - 6) but with the following differences.

*Intuitionistic vs classical logic* Our reasoning logic is based on intuitionistic logic instead of Church's choice of classical logic. While defaulting to a constructive approach to proving theorems about computation is certainly sensible, this choice is not essential and the sequent calculus proof system used to describe the intuitionistic reasoning logic can easily be modified to capture the classical variant. The choice between intuitionistic and classical logic can have, however, surprising consequences that are not immediately related to the familiar distinction between constructive and non-constructive logic. In particular, Tiu & Miller [44] have shown that, for a certain declarative treatment of binding in the $\pi$-calculus, provability of the bisimulation formula yields "open" bisimulation when the reasoning logic is intuitionistic and late ("closed") bisimulation if that logic is classical.

*Variables of higher-order type* Following Church, we used the type $o$ to denote formulas: thus, a variable of type $\tau_1 \to \cdots \to \tau_n \to o$ (for some $n \geq 0$) is a variable at "predicate type." In what follows, we shall not use such higher-order variables within formulas. We shall use variables of higher-order type that are not predicate types: in particular, we shall quantify over variables of type $\tau_1 \to \cdots \to \tau_n \to \tau_0$ (for some $n \geq 0$) where $\tau_0, \ldots, \tau_n$ are all primitive types. Removing restrictions on predicate quantification should be possible but, for the kind of project we intend here, it seems to be an unnecessary complication.

*Generic quantification* We include in $\mathcal{G}$ the $\nabla$-quantifier [30] and the associated notion of *nominal abstraction* [14] so that the "generic" reasoning associated with eigenvariables in the specification logic can be modeled directly and declaratively in $\mathcal{G}$. While $\nabla$ is a genuine departure from Church's original logic, it is a weak addition to the logic and is only relevant to the treatment of bindings in syntax (it enriches the possibilities of *binder mobility* [26]). If one is not treating bindings in syntax expressions of the specification logic, this quantifier plays no role.

*Induction and co-induction* A reasoning logic must certainly be powerful enough to support induction and co-induction. The logic $\mathcal{G}$ allows for the direct specification of recursive predicate definitions and to interpret them either as a least and or greatest fixed point in the sense of [2, 5, 22, 31]. The rules for induction and co-induction use higher-order predicate schema variables in their premises in order to range over possible pre- and post-fixed points. For example, the recursive definitions (written like logic programming clauses)

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathit{nat}\ z & \stackrel{\mu}{=} & \top & \mathit{member}\ B\ (B :: L) & \stackrel{\mu}{=} & \top \\ \mathit{nat}\ (s\ N) & \stackrel{\mu}{=} & \mathit{nat}\ N & \mathit{member}\ B\ (C :: L) & \stackrel{\mu}{=} & \mathit{member}\ B\ L \end{array}$$

are admitted to $\mathcal{G}$ as the following fixed point expressions:

$$\begin{array}{rcl} \mathit{nat} & = & \mu(\lambda p \lambda x.(x = 0) \vee (\exists y.(s\ y) = x \wedge p\ y)) \\ \mathit{member} & = & \mu(\lambda m \lambda x \lambda l.(\exists k.\ l = (x :: k)) \vee (\exists k \exists y.\ l = (y :: k) \wedge m\ x\ k)) \end{array}$$

In order to support induction and co-induction, the *closed world assumption* must be made: that is, we need to know the complete specification of a predicate in order to state the induction and co-induction rule for that predicate. Thus, the reasoning logic will assume the closed world assumption. On the other hand, computing with $\lambda$-tree syntax [25] uses the higher-order judgments of GENERIC and AUGMENT. Since these two judgments only make sense assuming the *open world assumption*, the specification logic will make that assumption. The next two sections contain a description of the specification logic and its encoding in the reasoning logic.

## 3 The specification logic

For our purposes here, we shall use the intuitionistic theory of hereditary Harrop formulas [28] restricted to second order as the specification logic. In particular, formulas in $hH^2$ are of two kinds. The *goal formulas* are given by:

$$G \ = \ \top \mid A \mid G \wedge G \mid A \supset G \mid \forall_\tau x.G,$$

where $A$ denotes atomic formulas and $\tau$ ranges over types that do not themselves contain the type $o$ of formulas. The *definite clauses* are formulas of the form $\forall x_1 \ldots \forall x_n.(G_1 \supset \cdots \supset G_m \supset A)$, where $n, m \geq 0$ and where quantification is, again, over variables whose types do not contain $o$. This restricted set of formulas

$$\frac{}{\Sigma : \Delta \vdash \top}\ \text{TRUE} \qquad \frac{\Sigma : \Delta \vdash G_1 \quad \Sigma : \Delta \vdash G_2}{\Sigma : \Delta \vdash G_1 \wedge G_2}\ \text{AND}$$

$$\frac{\Sigma : \Delta, A \vdash G}{\Sigma : \Delta \vdash A \supset G}\ \text{AUGMENT} \qquad \frac{\Sigma \cup \{c{:}\tau\} : \Delta \vdash G[c/x]}{\Sigma : \Delta \vdash \forall_\tau x.G}\ \text{GENERIC}$$

$$\frac{\Sigma : \Delta \vdash G_1[\bar{t}/\bar{x}] \quad \cdots \quad \Sigma : \Delta \vdash G_n[\bar{t}/\bar{x}]}{\Sigma : \Delta \vdash A}\ \text{BACKCHAIN}$$

where $\forall \bar{x}.(G_1 \supset \ldots \supset G_n \supset A') \in \Delta$ and $A'[\bar{t}/\bar{x}]$ $\lambda$-*conv* $A$

**Fig. 1.** Derivation rules for the $hH^2$ logic

is "second-order" in that to the left of an implication in a definite formula one finds goal formulas and to the left of an implication in a goal formula, one finds only atomic formulas.

Provability in $hH^2$ is formalized by a sequent calculus proof system in which sequents are of the form $\Sigma : \Delta \vdash G$, where $\Delta$ is a list of definite clauses, $G$ is a goal formula, and $\Sigma$ is a set of eigenvariables. The inference rules for $hH^2$ are presented in Figure 1: these rules are shown in [28] to be complete for the intuitionistic theory of $hH^2$. The GENERIC rule introduces an eigenvariable (reading rules from conclusion to premise) and has the usual freshness side-condition: $c$ is not in $\Sigma$. In the BACKCHAIN rule, for each term $t_i$ in the list $\bar{t}$, we require that $\Sigma \vdash t_i : \tau_i$ holds, where $\tau_i$ is the type of the quantified variable $x_i$. An important property to note about these rules is that if we use them to search for a proof of the sequent $\Sigma : \Delta \vdash G$, then all the intermediate sequents that we will encounter will have the form $\Sigma' : \Delta, \mathcal{L} \vdash G'$ for some $\Sigma'$ with $\Sigma \subseteq \Sigma'$, some goal formula $G'$, and some set of atomic formulas $\mathcal{L}$. Thus the initial context $\Delta$ is *global*: changes occur only in the set of atoms on the left and the goal formula on the right. In presenting sequents, we will elide the signature when it is inessential to the discussion.

The logic $hH^2$ is a subset of the logic programming language $\lambda$Prolog [32] and is given an effective implementation by Teyjus implementation of $\lambda$Prolog [33]. This logic has also been used to formally specify a wide range of operational semantic specifications and static (typing) judgments [15, 27, 23].

*An example: a typing judgment* We briefly illustrate the ease with which type assignment for the simply typed $\lambda$-calculus can be encoded in $hH^2$. There are two classes of objects in this domain: types and terms. Types are built from a single base type called $i$ and the arrow constructor for forming function types. Terms can be variables $x$, applications $(m\ n)$ where $m$ and $n$ are terms, and typed abstractions $(\lambda x : a.r)$ where $r$ is a term and $a$ is the type of $x$. The standard rules for assigning types to terms are given as the following inference

rules.

$$\frac{x : a \in \Gamma}{\Gamma \vdash x : a} \qquad \frac{\Gamma \vdash m : (a \to b) \quad \Gamma \vdash n : a}{\Gamma \vdash m\ n : b} \qquad \frac{\Gamma, x : a \vdash r : b}{\Gamma \vdash (\lambda x{:}a.r) : (a \to b)}\ x \text{ not in } \Gamma$$

Object-level simple types and untyped $\lambda$-terms can be encoded in a simply typed (meta-level) $\lambda$-calculus as follows. We assume the types $ty$ and $tm$ for representing object-level simple types and untyped $\lambda$-terms. The simple types are built from the two constructors $i : ty$ and $arr : ty \to ty \to ty$ and terms are built using the two constructors $app : tm \to tm \to tm$ and $lam : ty \to (tm \to tm) \to tm$. Note that the bound variables in an object-level abstraction are encoded by an explicit, specification logic abstraction: for example, the object-level term $(\lambda f : i \to i.(\lambda x : i.(f\ x)))$ will be represented by the specification logic term $lam\ (arr\ i\ i)\ (\lambda f.lam\ i\ (\lambda x.app\ f\ x))$. Given this encoding of the untyped $\lambda$-calculus and simple types, the standard inference rules for the typing judgment can be specified by the following theory describing the binary predicate *of*.

$$\forall m, n, a, b.(of\ m\ (arr\ a\ b) \supset of\ n\ a\ \supset\ of\ (app\ m\ n)\ b)$$
$$\forall r, a, b.(\forall x.(of\ x\ a \supset of\ (r\ x)\ b) \supset of\ (lam\ a\ r)\ (arr\ a\ b))$$

This specification in $hH^2$ does not maintain an explicit context for typing assumptions but uses hypothetical judgments instead. Also, the explicit side-condition in the rule for typing abstractions is not needed since it is captured by the freshness side-condition of the GENERIC rule in $hH^2$.

## 4 Encoding provability of the specification logic

The definitional clauses in Figure 2 encode $hH^2$ provability in $\mathcal{G}$; this encoding is based on ideas taken from [23]. Formulas in $hH^2$ are represented in this setting by terms of type *form* and we reuse the symbols $\wedge$, $\vee$, $\supset$, $\top$, and $\forall$ for constants involving this type in $\mathcal{G}$; we assume that the context will make clear which reading of these symbols is meant. The constructor $\langle\cdot\rangle$ is used to inject atomic formulas in $hH^2$ into the type *form* in $\mathcal{G}$. As we have seen earlier, provability in $hH^2$ is about deriving sequents of the form $\Delta, \mathcal{L} \vdash G$, where $\Delta$ is a fixed list of definite clauses and $\mathcal{L}$ is a varying list of atomic formulas. Our encoding uses the $\mathcal{G}$ predicate *prog* to represent the definite clauses in $\Delta$. In particular, the definite clause $\forall \bar{x}.[G_1 \supset \cdots \supset G_n \supset A]$ is encoded as the clause $\forall \bar{x}.prog\ A\ (G_1 \wedge \cdots \wedge G_n) \stackrel{\mu}{=} \top$ and a set of such $hH^2$ definite clauses is encoded as a set of *prog* clauses. (The descriptions of *prog* above and of *seq* in Figure 2 use the symbol $\stackrel{\mu}{=}$ to indicate that these names are to be associated with fixed point definitions.) Sequents in $hH^2$ are represented in $\mathcal{G}$ by formulas of the form $seq_N\ L\ G$ where $L$ is a list encoding the atomic formulas in $\mathcal{L}$ and where $G$ encodes the goal formula. The provability of such sequents in $hH^2$, given by the rules in Figure 1, leads to the clauses that define *seq* in Figure 2. The argument $N$ that is written as a subscript in the expression $seq_N\ L\ G$ encodes an upper bound on the height of the corresponding $hH^2$ derivation and is used to formalize proofs by induction

$$
\begin{array}{lcl}
seq_{(s\ N)}\ L\ \top & \stackrel{\mu}{=} & \top \\
seq_{(s\ N)}\ L\ (B \wedge C) & \stackrel{\mu}{=} & seq_N\ L\ B \wedge seq_N\ L\ C \\
seq_{(s\ N)}\ L\ (A \supset B) & \stackrel{\mu}{=} & seq_N\ (A :: L)\ B \\
seq_{(s\ N)}\ L\ (\forall B) & \stackrel{\mu}{=} & \nabla x.seq_N\ L\ (B\ x) \\
seq_{(s\ N)}\ L\ \langle A \rangle & \stackrel{\mu}{=} & member\ A\ L \\
seq_{(s\ N)}\ L\ \langle A \rangle & \stackrel{\mu}{=} & \exists b.prog\ A\ b \wedge seq_N\ L\ b
\end{array}
$$

**Fig. 2.** Encoding provability of $hH^2$ in $\mathcal{G}$

on the height of proofs. This argument has type *nt* for which there are two constructors: $z$ of type *nt* and $s$ of type $nt \rightarrow nt$. Similarly, the type of the non-empty list constructor :: is $atm \rightarrow lst \rightarrow lst$, where *atm* denotes the type of atomic formulas and *lst* denotes the type of a list of atomic formulas.

Notice the following points about this specification of provability. First, the $\nabla$-quantifier is used in the reasoning logic to capture the "generic" reasoning involved with using eigenvariables in specifying the provability of the specification logic universal quantifier. Second, the *seq* predicate contains an explicit list of atomic formulas and this is augmented by an atomic assumption whenever the proof of an implication is attempted. Third, the last clause for *seq* specifies backchaining over a given $hH^2$ definite clauses stored as *prog* clauses. The matching of atomic judgments to heads of clauses is handled by the treatment of definitions in the logic $\mathcal{G}$; thus the last rule for *seq* simply performs this matching and makes a recursive call on the corresponding clause body. Finally, the natural number (subscript) argument to *seq* is used to measure the height of specification logic proofs.

Since we have encoded derivability in $hH^2$, we can prove general properties about it in $\mathcal{G}$. For example, the following theorem in $\mathcal{G}$ states that the judgment $seq_n\ \ell\ g$ is not affected by permuting, contracting, or weakening the context $\ell$.

$$\forall n, \ell_1, \ell_2, g.(seq_n\ \ell_1\ g) \wedge (\forall e.member\ e\ \ell_1 \supset member\ e\ \ell_2) \supset (seq_n\ \ell_2\ g)$$

Using this theorem with the encoding of typing judgments for the simply typed $\lambda$-calculus, for example, we immediately obtain that permuting, contracting, or weakening the typing context of a typing judgment does not invalidate that judgment.

Two additional $\mathcal{G}$ theorems are called the *instantiation* and *cut* properties. To state these properties, we use the following definition to abstract away from proof sizes.

$$seq\ \ell\ g \stackrel{\mu}{=} \exists n.nat\ n \wedge seq_n\ \ell\ g.$$

The *instantiation property* states that if a sequent judgment is proved generically (using $\nabla$) then, in fact, it holds universally (that is, for all substitution instances). The exact property is

$$\forall \ell, g.(\nabla x.\ seq\ (\ell\ x)\ (g\ x)) \supset (\forall t.\ seq\ (\ell\ t)\ (g\ t)).$$

The *cut property* allows us to remove hypothetical judgments using a proof of such judgments. This property is stated as the $\mathcal{G}$ theorem

$$\forall \ell, a, g.(seq\ \ell\ \langle a \rangle) \wedge (seq\ (a :: \ell)\ g) \supset seq\ \ell\ g,$$

To demonstrate the usefulness of the instantiation and cut properties, note that using them together with our encoding of typing for the simply typed $\lambda$-calculus leads to an easy proof of the type substitution property, *i.e.*, if $\Gamma, x : a \vdash m : b$ and $\Gamma \vdash n : a$ then $\Gamma \vdash m[x := n] : b$.

## 5 Various implemented systems

Various systems and prototypes have been built to test and exploit the concepts of $\lambda$-tree syntax, higher-order judgments, and two-level logic. We overview these systems here.

### 5.1 Teyjus

Nadathur and his students and colleagues have developed the Teyjus compiler and run-time system [33] for $\lambda$Prolog. Although Teyjus is designed to compile and execute a rich subset of higher-order intuitionistic logic, it provides an effective environment for developing and animating the more restricted logic of $hH^2$.

### 5.2 Bedwyr

Baelde *et. al.* have implemented the Bedwyr model checker [4] which automates deduction for a subset of $\mathcal{G}$. The core logic engine in Bedwyr implements a sequent calculus prover that unfolds fixed points on both sides of sequents. As a result, it is able to perform standard model checking operations such as reachability, simulation, and winning strategies. Since unfolding is the only rule used with fixed points, such unfoldings must terminate in order to guarantee termination of the model checker. Bedwyr also provides the $\nabla$-quantifier so model checking problems can directly express problems involving bindings. A particularly successful application of Bedwyr is on determining (open) simulation for the finite $\pi$-calculus [43, 44].

### 5.3 Abella

Gacek has built the Abella interactive theorem prover [12] for proving theorems in $\mathcal{G}$. The two level logic approach is built into Abella and the cut and instantiation properties of Section 4 are available as reasoning steps (tactics). Abella accepts $hH^2$ specifications written as $\lambda$Prolog programs. Reasoning level predicates can then be defined inductively or co-inductively: these can also refer to provability of $hH^2$ specifications. Examples of theorems proved in Abella: pre-congruence of open bisimulation for the finite $\pi$-calculus; POPLmark challenge

problems 1a and 2a; the Church-Rosser property and standardization theorems for the $\lambda$-calculus; and a number of properties related to the specification of the static and dynamic semantics of programming languages (type preservation, progress, and determinacy).

### 5.4 Tac

Baelde *et. al.* [7] have built an automated theorem prover for a fragment of $\mathcal{G}$. This prototype prover was developed to test various theorem prover designs that are motivated by the theory of focused proofs for fixed points [6]. This prover is able to automatically prove a number of simple theorems about relational specifications. Currently, Tac does not have convenient support for treating two-level logic although there is no particular problem with having such support added.

## 6 Various aspects of logic

There have been a number of papers and a number of logics that have been proposed during the past several years that shaped our understanding of the two-level logic approach to specifying and reasoning about computation. In this section, I briefly overview the key ingredients to that understanding.

### 6.1 Abstract syntax as $\lambda$-tree syntax

The $\lambda$Prolog programming language [32] was the first programming language to support what was later called "higher-order abstract syntax" [35]. This later term referred to the encoding practice of using "meta-level" binding in a programming language to encode the "object-level" bindings in syntactic objects. Unfortunately, the meta-level bindings available in functional programming (which build functions) and logic programming (which build syntactic expressions with bindings) are quite different. Since using the term "higher-order abstract syntax" to refer to both styles of encoding is confusing, the term *$\lambda$-tree syntax* was introduced in [25] to denote the treatment of syntax using weak equality (such as $\alpha$, $\beta$, and $\eta$ on simply typed $\lambda$-terms). A key ingredient to the manipulation of $\lambda$-tree syntax involves the unification of $\lambda$-terms [19, 24].

### 6.2 Fixed points

Schroeder-Heister [39, 40] and Girard [17] independently proposed a proof-theoretic approach to the closed-world assumption. The key development was the shift from viewing a logic program as a theory that provided *some* of the meaning of *undefined* predicates to viewing logic programs as recursive *definitions* that *completely* describe predicates. In this later case, it is easy to view predicates then as only convenient names for fixed point expressions. The proof theoretic treatment of such fixed points involves the first-order unification of eigenvariables. It

was straightforward to extend that unification to also involve the unification of simply typed $\lambda$-terms and, as a result, this treatment of fixed points could be extended to the treatment of $\lambda$-tree syntax [21, 22].

### 6.3 $\nabla$-quantification

The $\nabla$-quantifier was introduced by Miller & Tiu [29, 30] in order to help complete the picture of fixed point reasoning with $\lambda$-tree syntax. To provide a quick motivation for this new quantifier, consider the usual inference rule for proving the equality of two $\lambda$-abstracted terms.

$$(\zeta) \qquad \text{if } M = N \text{ then } \lambda x.M = \lambda x.N$$

In a formalized meta-theory, the quantification of $x$ in the premise equation must be resolved and the universal quantification of $x$ is a natural candidate. This choice leads to accepting the equivalence

$$(\forall x.M = N) \equiv (\lambda x.M = \lambda x.N).$$

This equivalence is, however, problematic when negation is involved. For example, since there is no (capture-avoiding) substitution for the variable $w$ that makes the two (simply typed) term $\lambda x.w$ and $\lambda x.x$ equal (modulo $\lambda$-conversion), one would expect that our reasoning logic is strong enough to prove $\forall w.\neg(\lambda x.x = \lambda x.w)$. Using the equivalence above, however, this is equivalent to $\forall w.\neg\forall x.x = w$. Unfortunately, this formula should not be provable since it is true if the domain of quantification is a singleton. The $\nabla$-quantifier is designed to be the proper quantifier to match the $\lambda$-binder: in fact, the formula $\forall w.\neg\nabla x.x = w$ has a simple proof in the proof systems for $\nabla$. (As this example suggests, it is probably challenging to find a model-theory semantics for $\nabla$.)

Two variants of $\nabla$ appear in the literature and they differ on whether or not they accept the following *exchange* and *strengthening* equivalences:

$$\nabla x \nabla y.Bxy \equiv \nabla y \nabla x.Bxy \qquad\qquad \nabla x_\tau.B \equiv B \quad (x \text{ not free in } B)$$

While the first equivalence is often admissible, accepting the second rule is significant since it forces the domain of quantification for $x$ (the type $\tau$) to be infinite: that is, the formula

$$\exists_\tau x_1 \ldots \exists_\tau x_n. \left[ \bigwedge_{1 \le i,j \le n, i \neq j} x_i \neq x_j \right]$$

is provable for any $n \geq 1$. The *minimal generic quantification* of Baelde [3] rejects these as proof principle in part because there are times when a specification logic might need to allow possibly empty types: accepting these principles in the reasoning logic would force types using $\nabla$-quantification to have infinite extent. On the other hand, the *nominal generic quantification* of Gacek *et. al.* [13, 14] accepts these two additional equivalences.

### 6.4 Induction and co-induction

The earlier work on fixed points only allowed for the unfolding of fixed points: as a result, it was not possible to reason specifically about the least or the greatest fixed point. In the series of PhD thesis, McDowell [21], Tiu [42], Baelde [2], and Gacek [13] developed numerous enrichments to our understanding of induction and co-induction: the last three of these theses have also been concerned with the interaction of the $\nabla$-quantifier and fixed point reasoning.

### 6.5 Two-level logic

The force behind developing a two-level logic approach to reasoning about logic specifications is the necessity to treat both induction (and co-induction) and higher-order judgments (AUGMENT and GENERIC in Figure 1). These latter judgments only make sense when the "open world assumption" is in force: that is, atoms are undefined and we can always add more clauses describing their meaning. On the other hand, induction and co-induction only makes sense when the "close world assumption" is in force: that is, we can only induct when we have completely defined a predicate. It seems that we are at an impasse: in order to reason about logic specifications employing higher-order judgments, we need to have a logic that does not have higher-order judgments. To get around this impasse, McDowell & Miller [21, 22] proposed using two logics: the reasoning logic assumes the closed world assumption and contains the principles for induction and co-induction while the specification logic assumes the open-world assumption and allows for higher-order judgments. The interface between these two logics has two parts. First, the term structures (including those treating binding) are shared between the two logics, and, second, provability of the specification logic is encoded as a predicate in the reasoning logic (as in Figure 2).

There are, of course, choices in the selection of not only the specification logic but also the proof system used to encode that logic. For example, extending $hH^2$ to allow the linear logic implication $\multimap$ was considered in [21, 22]: the linear specification logic allowed for natural specifications of the operational semantics of a number of imperative programming features. There are also choices in how one describes specification logic provability: while two different proof systems should describe the same notion of provability, the form of that definition plays a large role in theorems involving provability. For example, the proof system in Figure 1 describes the *uniform proofs* of [28], which, in the terminology of focused proofs systems for intuitionistic logic [20], arises from assigning the *negative polarity* to all atomic formulas. The resulting "goal-directed" ("top-down") proofs mimic the head-normal form of typed $\lambda$-terms. Thus, induction over the *seq* judgment corresponds closely to induction over head-normal form. It is also possible to consider a proof system for *seq* in which all atoms are assigned a positive polarity. The resulting proofs would be "bottom-up": such proofs would naturally encode terms that contains explicit sharing. There may be domains where an induction principle over such bottom-up proofs would be more natural and revealing than for top-down proofs.

## 7 Related work

This paper provides an overview of a multi-year effort to develop a logic and its proof theory that treats binding and fixed point together. It is common, however, that these two aspects of logic have been treated separately, as we describe below.

Many systems for reasoning about computations start with established inductive logic theorem provers such as Coq [10, 8] and Isabelle/HOL [34], and then use those systems to build approaches to binding and substitution. Three such notable approaches are the locally nameless representation [1], the Nominal package for Isabelle/HOL [45], and Hybrid [11].

On the other hand, there are a variety of systems for specifying computations which take binding as a primitive notion and then attempt to define separately notions of induction. Many of these systems start with the LF logical framework [18], a dependently typed $\lambda$-calculus with a direct treatment of variable binding. While the LF type system can directly treat $\lambda$-tree syntax, it does not include a notion of induction. Twelf [36] is able to establish that various recursively defined relations on LF-typed terms are, in fact, determinate (*i.e.*, functional) and/or total. These conclusions can be used in concert with the dependently typed $\lambda$-terms to conclude a wide range of properties of the original LF specification. Similar functional approaches have been developed starting with $\mathcal{M}_2^+$ [41], a simple meta-logic for reasoning over LF representations where proof terms are represented as recursive functions. More recent work includes the Delphin [38] and Beluga [37] functional languages which can be used in the same spirit as $\mathcal{M}_2^+$. In all of these approaches, however, side-conditions for termination and coverage are required and algorithms have been devised to check for such properties. Since termination and coverage are in general undecidable, such algorithms are necessarily incomplete.

*Acknowledgments.* I thank Andrew Gacek and Alwen Tiu for their comments on this paper.

## References

1. B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *35th ACM Symp. on Principles of Programming Languages*, pages 3–15. ACM, Jan. 2008.
2. D. Baelde. *A linear approach to the proof-theory of least and greatest fixed points.* PhD thesis, Ecole Polytechnique, Dec. 2008.
3. D. Baelde. On the expressivity of minimal generic quantification. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*, ENTCS 228, pages 3–19, 2008.
4. D. Baelde, A. Gacek, D. Miller, G. Nadathur, and A. Tiu. The Bedwyr system for model checking over syntactic expressions. In F. Pfenning, editor, *21th Conf. on Automated Deduction (CADE)*, LNAI 4603, pages 391–397. Springer, 2007.
5. D. Baelde and D. Miller. Least and greatest fixed points in linear logic. In N. Dershowitz and A. Voronkov, editors, *International Conference on Logic for Programming and Automated Reasoning (LPAR)*, volume LNCS 4790, pages 92–106, 2007.

6. D. Baelde, D. Miller, and Z. Snow. Focused inductive theorem proving. In J. Giesl and R. Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, LNCS 6173, pages 278–292, 2010.
7. D. Baelde, D. Miller, Z. Snow, and A. Viel. Tac: A generic and adaptable interactive theorem prover. http://slimmer.gforge.inria.fr/tac/, 2009.
8. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
9. A. Church. A formulation of the simple theory of types. *J. of Symbolic Logic*, 5:56–68, 1940.
10. T. Coquand and C. Paulin. Inductively defined types. In *Conference on Computer Logic*, LNCS 417, pages 50–66. Springer-Verlag, 1988.
11. A. Felty and A. Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. To appear in the *J. of Automated Reasoning*.
12. A. Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Fourth International Joint Conference on Automated Reasoning*, LNCS 5195, pages 154–161. Springer, 2008.
13. A. Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. PhD thesis, University of Minnesota, 2009.
14. A. Gacek, D. Miller, and G. Nadathur. Combining generic judgments with recursive definitions. In F. Pfenning, editor, *23th Symp. on Logic in Computer Science*, pages 33–44. IEEE Computer Society Press, 2008.
15. A. Gacek, D. Miller, and G. Nadathur. Reasoning in Abella about structural operational semantics specifications. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*, ENTCS 228, pages 85–100, 2008.
16. A. Gacek, D. Miller, and G. Nadathur. A two-level logic approach to reasoning about computations. Submitted 16 November, Nov. 2009.
17. J.-Y. Girard. A fixpoint theorem in linear logic. An email posting to the mailing list linear@cs.stanford.edu, Feb. 1992.
18. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. of the ACM*, 40(1):143–184, 1993.
19. G. Huet. A unification algorithm for typed $\lambda$-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
20. C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.
21. R. McDowell. *Reasoning in a Logic with Definitions and Induction*. PhD thesis, University of Pennsylvania, Dec. 1997.
22. R. McDowell and D. Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.
23. R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1):80–136, 2002.
24. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991.
25. D. Miller. Abstract syntax for variable binders: An overview. In J. Lloyd and *et al.*, editors, *Computational Logic - CL 2000*, LNAI 1861, pages 239–253.
26. D. Miller. Bindings, mobility of bindings, and the $\nabla$-quantifier. In J. Marcinkowski and A. Tarlecki, editors, *18th International Conference on Computer Science Logic (CSL) 2004*, LNCS 3210, page 24, 2004.
27. D. Miller. Formalizing operational semantic specifications in logic. *Concurrency Column of the Bulletin of the EATCS*, Oct. 2008.

28. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
29. D. Miller and A. Tiu. A proof theory for generic judgments: An extended abstract. In P. Kolaitis, editor, *18th Symp. on Logic in Computer Science*, pages 118–127. IEEE, June 2003.
30. D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, Oct. 2005.
31. A. Momigliano and A. Tiu. Induction and co-induction in sequent calculus. In M. Coppo, S. Berardi, and F. Damiani, editors, *Post-proceedings of TYPES 2003*, LNCS 3085, pages 293–308, Jan. 2003.
32. G. Nadathur and D. Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Aug. 1988. MIT Press.
33. G. Nadathur and D. J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, LNAI 1632, pages 287–291, Trento, 1999.
34. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS Tutorial 2283.
35. F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.
36. F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, LNAI 1632, pages 202–206, Trento, 1999. Springer.
37. B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th Annual ACM Symposium on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM, 2008.
38. A. Poswolsky and C. Schürmann. System description: Delphin - A functional programming language for deductive systems. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*, volume 228, pages 113–120, 2008.
39. P. Schroeder-Heister. Cut-elimination in logics with definitional reflection. In D. Pearce and H. Wansing, editors, *Nonclassical Logics and Information Processing*, LNCS 619, pages 146–171. Springer, 1992.
40. P. Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 222–232. IEEE Computer Society Press, IEEE, June 1993.
41. C. Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, Oct. 2000. CMU-CS-00-146.
42. A. Tiu. *A Logical Framework for Reasoning about Logical Specifications*. PhD thesis, Pennsylvania State University, May 2004.
43. A. Tiu. Model checking for π-calculus using proof search. In M. Abadi and L. de Alfaro, editors, *CONCUR*, LNCS 3653, pages 36–50. Springer, 2005.
44. A. Tiu and D. Miller. Proof search specifications of bisimulation and modal logics for the π-calculus. *ACM Trans. on Computational Logic*, 11(2), 2010.
45. C. Urban. Nominal reasoning techniques in Isabelle/HOL. *J. of Automated Reasoning*, 40(4):327–356, 2008.