HAL
open science

Proofs You Can Believe In. Proving Equivalences Between Prolog Semantics in Coq

Jael Kriener, Andy King, Sandrine Blazy

To cite this version:

Jael Kriener, Andy King, Sandrine Blazy. Proofs You Can Believe In. Proving Equivalences Between Prolog Semantics in Coq. 15th International Symposium on Principles and Practice of Declarative Programming (PPDP), Sep 2013, Madrid, Spain. pp.37-48. hal-00908848

HAL Id: hal-00908848

https://inria.hal.science/hal-00908848v1

Submitted on 25 Nov 2013

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.

Proofs You Can Believe In

Proving Equivalences Between Prolog Semantics in Coq

Jael Kriener
School of Computing
University of Kent
Canterbury, UK

Andy King
School of Computing
University of Kent
Canterbury, UK

Sandrine Blazy
IRISA
University of Rennes I
Rennes, France

ABSTRACT

Basing program analyses on formal semantics has a long and successful tradition in the logic programming paradigm. These analyses rely on results about the relative correctness of mathematically sophisticated semantics, and authors of such analyses often invest considerable effort into establishing these results. The development of interactive theorem provers such as Coq and their recent successes both in the field of program verification as well as in mathematics, poses the question whether these tools can be usefully deployed in logic programming. This paper presents formalisations in Coq of several general results about the correctness of semantics in different styles; forward and backward, top-down and bottom-up. The results chosen are paradigmatic of the kind of correctness theorems that semantic analyses rely on and are therefore well-suited to explore the possibilities afforded by the application of interactive theorem provers to this task, as well as the difficulties likely to be encountered in the endeavour. It turns out that the advantages offered by moving to a functional setting, including the possibility to apply higher-order abstract syntax, are considerable.

Keywords

abstract interpretation, Coq, fixpoint semantics, interactive theorem proving, logic programming

1. INTRODUCTION

Formal semantics are at the very heart of the logic programming paradigm [van Emden and Kowalski, 1976] and the practice of "find[ing] notions of models which really capture the operational semantics [..] for semantics-based program analysis" [Levi, 1991] can be traced back at least thirty years [Gabbrielli and Levi, 1991]. The success of formal semantics in capturing observable properties of logic programs is arguably due to the simplicity of the object language itself. The wide range of available semantics aids the task of program analysis since one merely needs to choose the simplest semantics which captures the property that one wants to abstract. Moreover, the process of abstraction itself has a long and rigorous tradition [Cousot and Cousot, 1979]. When abstract interpretation is used as a design methodology for formal semantics, it usually entails a certain style of stating and proving correctness results. This style has, therefore, become common place in the literature on logic programming semantics and analysis [Barbuti et al., 1993, Codish et al., 1994, Janssens and Bruynooghe, 1992, Muthukumar and Hermenegildo, 1992].

Over the same time, another community of logicians has made significant progress in developing tools to support mathematical and program-analytic reasoning; interactive theorem provers like ACL2, Coq and Isabelle are now successfully applied both in mathematics e.g. a mechanised proof of the 4-colour theorem [Gonthier, 2007] and recently the Feit-Thompson theorem [Gonthier et al., 2013], and in verification e.g. the verified C-compiler [Leroy, 2009, CompCert Development Team, 2012] and the verified operating system kernel SeL4 [Klein et al., 2010]. As of recent, Coq provides native support for induction over dependent data types [Coq Development Team, 2010], and for vectors a.k.a. tuples [Blanqui and Koprowski, 2011], which provide a convenient way to handle n-ary terms and n-ary predicates, which are ubiquitous in logic programming. These developments make it possible to apply Coq to the task of proving correctness results in the style of logic program semantics.

1.1 Motivation

We do not propose to do so simply 'because we can', but for the following, well-considered reasons:

Maintainable Proofs.

We know from painful experience [Kriener and King, 2011] that correctness proofs in the logic programming semantics style are like programs in a very pragmatic sense: they require maintenance. Our 'development cycle' often proceeds from the definition of a semantics, to a proof of its correctness, to its implementation as a tool. If the last stage encounters pragmatic problems, we may well go back and adjust the semantics accordingly. In that situation, the proofs need to be adjusted likewise. These adjustments are very prone to error; they consist in identifying all and only those cases where the proof 'breaks', that is where the changes have a theoretical impact. They are carried out by people who are convinced that the change does not affect overall correctness. These same people have invested considerably time in the details of a pen-and-pencil proof, and hence are

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPDP'13 September 16 - 18 2013, Madrid, Spain
Copyright 2013 ACM ACM 978-1-4503-2154-9/13/09 ...$15.00.

unlikely to have the objective distance required to critically re-examine their own work. Combined with time or other pressures, the probability of a relevant case being missed is high. We have found this in our own work [Kriener and King, 2011]: if it had not been for the diligence and the inordinate amount of time spent by a single reviewer, Maurice Bruynooghe, we would not have managed to achieve proper proof-maintenance. Interactive theorem provers are made for exactly this task.

Formal Continuity Proofs.

The vast majority of proofs in formal semantics is based on (a version of) Kleene's iteration theorem, stating that the least fixpoint of a continuous function is the join of its iterates. To apply this theorem, and hence standard proof techniques, semantic operators have to be continuous. The formal semantics community has developed the habit of observing, rather than proving, the continuity of these operators. While compositionality is generally enough to argue continuity of the inductive cases, it is often the basic cases that deserve attention: not until doing this formalisation did we appreciate that the forward operators discussed in Section 4.1 are continuous only if their domain is a complete Heyting algebra, that is to say meet distributes over join; which it is only because downward closures do indeed construct complete Heyting algebras.

The cavalier approach to continuity proofs becomes even more of a problem when semantics are extended to treat non-standard features, such as the *cut*: a number of standard denotational semantics for Prolog with *cut* are constructed by extending a semantics for Prolog with an additional clause for the *cut* [Debray and Mishra, 1988, de Vink, 1989, Billaud, 1990]. We have spent considerable time attempting to 'observe' the continuity of such definitions, but our observational powers have failed us. Really, though, the burden of proof is on the authors here.

These continuity proofs are tedious, but they turn out to be non-trivial in some cases. When an entire community routinely omits such proofs, the potential that a relevant case is missed, is high. Coq exhibits gaps in formal developments very clearly, and thus works strongly against any tendencies to omit technical details. At the same time, it offers a way of efficiently doing rigorous but laborious proofs. Even without the development of sophisticated tactics which take care of tedious details automatically (see Section 7), proof-effort does not have to be repeated: considerable parts of the continuity proofs in our development are literally copied, pasted, and then fixed in a few places.

Replacing Renaming by HOAS.

Finally, apart from these pragmatic human-centric reasons, there is a mathematical motivation for moving from pen-and-pencil proofs to proofs in a functional, automatically verified setting: logic program semantics community standardly deals with the issue of free variable by applying renaming operators, that are constructed from projections, which are approximating in the abstract context. The functional community has a similar, if somewhat harder, problem with name capturing; and has developed an elegant non-approximating solution - higher order abstract syntax (HOAS). Applying this approach in the logical context relieves us of the need to apply renaming operations, and thus renders the coresponding requirements on the domains superfluous and the definitions and proofs less complex. (In particular: the case of a predicate call becomes trivial throughout.) Though not conceptually easier then renaming, the HOAS-approach is much more natural when working within the functional setting of Coq, because it is based on function abstraction and application – native concepts in Coq (see Section 3.1).

1.2 Contributions

This paper concerns methodology rather than a novel semantic result itself. Its contribution lies in the exploration of a new, arguably better, way of stating and proving semantic results in general. All results presented in this paper have been mechanically verified using the Coq proof assistant. The complete Coq development is available online at `http://www.cs.kent.ac.uk/people/rpg/jek26/FvB`. Consequently, the paper omits the proofs for results stated in it; the reader is referred to the Coq development for the full proofs. This work makes the following contributions:

HOAS for Prolog: We present an agent-style HOAS for Prolog, which implements renaming as function application, thus making free variables a non-issue in proofs and significantly reducing proof-complexity.

Libraries for Semantic Proofs: We present a collection of libraries, based on work by Cachera and Pichardie [Cachera and Pichardie, 2010], and containing formalisations of the following results:
(a) continuous and co-continuous versions of Kleene's iteration theorem,
(b) the concept of a complete Heyting algebra,
(c) closure operators and proofs that they construct complete lattices and complete Heyting algebrae,
(d) and sticky domains [Hudak, 1987] and function spaces, i.e. structures which observe and propagate assertion violations [King and Lu, 2003].

These libraries form the basis for our formalisations of semantic operators and correctness proofs and are ready to be used by others.

Sample Formalised Semantics: We have chosen three semantics, representing the three most common semantic paradigms – forward, bottom-up and backward. We present formalisations of these based on the aforementioned HOAS, and formal proofs of the two properties, monotonicity and continuity, which guarantee that a semantic is well-defined and can be treated in the standard way.

Paradigm Verified Semantic Proof: Finally, we present fully verified proofs of the (well-known) equivalence between forward and backward approaches in logic program semantics, which:
(a) provide examples of how to construct such proofs in Coq,
(b) identify hidden assumptions in earlier results.

2. FORMALISED PRELIMINARIES

Let us start this section with a note on dependent types. As shall become clear in this section, virtually every type in the following is dependent on the natural numbers (the only exception being the abstract domain, which is a parameter to the entire development). The reason is, that vectors of variables are absolutely fundamental in logic program analysis; and vectors have an arity, i.e. are dependently typed. Building on a dependent type propagates the dependency upward to every semantic and syntactic structure constructed from vectors of variables. As of version

8.4, Coq's standard library contains a theory of polymorphic vectors.

As is explained in Section 6, when we started this work a year ago, there was no library containing formalisations of the results required for logic programming semantics available with Coq version 8.4. A collection of libraries for domain theory, based on category theory, [Benton et al., 2009] has since become compatible with version 8.4; however, at the time the only option was to develop our own libraries, proving from first principles the mathematical foundations of logic programming semantics. These libraries are based on work by Cachera and Pichardie [Cachera and Pichardie, 2010], who have formalised the Knaster-Tarksi fixpoint theorem [Tarski, 1955] and the lattice theory required for it. This section is intended as documentation of the developments in the following files: BottomGeneralCompleteLattice.v, CompleteHeytingAlgebra.v, KnasterTarski.v, Kleene-Theorem.v, CoContinuousTheorem.v, DownClosure.v, StickyCompleteLattice.v, and finally StickyCompleteHeytingAlgebra.v. Most of these results can be found, e.g. in [Birkhoff, 1967, Abramsky and Jung, 1994], but to make the paper self-contained, and since this paper aspires to be a road map for applying Coq to logic programming semantics, we nevertheless summarise the results in our libraries. A casual reader should be able to skim much of Section 2.1.

2.1 Semantic Operators and their Fixpoints

As mentioned in the introduction, we present formalisations of logic program semantics in the forward, or top-down, the bottom-up, and the backward style.

Forward semantics are generally operational. These semantics work over representations of current states of computation, encoded in some concrete or abstract domain. They construct mappings that simulate the effect that the execution of a goal would have on a given state [Janssens and Bruynooghe, 1992].

Bottom-up semantics are denotational. They, too, work over domains encoding current states. Rather than tracing the call- and answer-behaviour of a query, these techniques attempt to capture the behaviour of a goal in a compositional fashion. They derive the meaning of a predicate from the predicates that it calls [Barbuti et al., 1993].

Backward semantics, finally, are denotational in spirit, too. They derive pre-conditions for satisfying some requirement by propagating information backwards against the control flow [King and Lu, 2002].

Results about all the above kinds of semantics are founded on a corollary of Kleene's recursion theorem [Kleene, 1938], which states:

THEOREM 1 (KLEENE 1938). *Given a complete lattice A with respect to $\sqsubseteq$, and a continuous function $f : A \xrightarrow{c} A$, the least fixpoint (lfp) of f is the join of its iterates:*
$\mathsf{lfp}\ f = \bigsqcup\{\bot, f\bot, ff\bot, \ldots\}$. [COQ PROOF][1]

Forward and bottom-up semantics are standardly defined as the least fixpoint of some continuous operator. Backward semantics, in contrast, are usually defined as a greatest fixpoint. In these cases, the dual theorem is applied:

THEOREM 2 (DUAL OF KLEENE 1938). *Given a complete lattice A with respect to $\sqsubseteq$, and a co-continuous function $f : A \xrightarrow{co-c} A$, the greatest fixpoint (gfp) of f is the meet of its (downward) iterates: $\mathsf{gfp}\ f = \bigsqcap\{\top, f\top, ff\top, \ldots\}$.* [COQ PROOF]

[1]This is a direct link to the web page showing the corresponding Coq theorem and likewise for definitions.

Complete Lattices and Complete Heyting Algebras.

The structures underlying this result are complete lattices and continuous function spaces. Below, we shall sometimes omit some detail and simply say "A is a CL wrt $\sqsubseteq$". The intended meaning of this is "One can define join and meet operators and find bottom and top elements such that they form a CL in conjunction with A and $\sqsubseteq$."

Importantly, the structure of a CL can be lifted to function spaces which have a CL as their co-domain:

LEMMA 1 (LIFTING CL TO FUNCTION SPACE). *Given a CL wrt $\sqsubseteq_B$, B, and any set A, the function space $[A \to B]$ is a CL wrt $\sqsubseteq_{AB}$, defined by a pointwise lifting of $\sqsubseteq_B$: $f_1 \sqsubseteq_{AB} f_2 \iff \forall a,\ f_1\ a \sqsubseteq_B f_2\ a$.* [COQ PROOF]

Complete Heyting Algebras form the computational domain that is required for backwards analysis.

DEFINITION 1 (COMPLETE HEYTING ALGEBRA). *A* complete Heyting algebra *(cHa) is a CL in which binary meet distributes over join: $a \sqcap \bigsqcup S = \bigsqcup\{a \sqcap s \mid s \in S\}$.* [COQ DEFINITION]

A important property of a cHa is the existence of a unique pseudo-complement for each pair of elements in it.

DEFINITION 2 (PSEUDO-COMPLEMENT). *Given a CL wrt $\sqsubseteq$, A, the* pseudo-complement *of two elements of A, a and b, is denoted by $a \Rightarrow b$ and defined as the greatest $x \in A$, such that $a \sqcap x \sqsubseteq b$.* [COQ DEFINITION]

LEMMA 2 (CHA HAS UNIQUE PSEUDO-COMPLEMENTS). *Given a cHa wrt $\sqsubseteq$, A, for every two elements a and b of A, $a \Rightarrow b$ can be uniquely defined as $\bigsqcup\{x \in A \mid x \sqcap a \sqsubseteq b\}$.* [COQ PROOF]

Monotone and (Co-)Continuous Functions.

In these structures, certain functions have the property of possessing fixpoints. These functions are those that respect, or are compatible with, the structure of their domains.

Monotone functions are a particular case in point: they respect the order of their domain and co-domain. The Knaster-Tarski theorem [Tarski, 1955] states that these functions have least and greatest fixpoints.

DEFINITION 3 (MONOTONICITY). *Given a CL wrt $\sqsubseteq_A$, A, and a CL wrt $\sqsubseteq_B$, B, a function $f : A \to B$, is* monotone, *iff $\forall\ a_1\ a_2,\ a_1 \sqsubseteq_A a_2 \Longrightarrow fa_1 \sqsubseteq_B fa_2$.*

We denote the function space of monotone functions from A to B by $[A \xrightarrow{m} B]$. [COQ DEFINITION]

Note that the function space $[A \xrightarrow{m} B]$ is a CL wrt the standard lifting of the partial order from B. [COQ PROOF] Continuous functions are monotone functions which in addition preserve joins of non-empty chains:[2]

[2]There is some inconsistency in the literature concerning the requirement that S be a chain: [Abramsky and Hankin, 1987] include it in their definition of continuity, [Abramsky and Jung, 1994] do not. Including the chain requirement weakens the notion of continuity, while not changing the proof of Kleene's iteration theorem. Since we can prove our semantic operators continuous only in the weaker sense, we adopt the weaker definition throughout.

Definition 4 (continuity). *Given a CL wrt $\sqsubseteq_A$, A, and a CL wrt $\sqsubseteq_B$, B, a function $f : A \xrightarrow{m} B$ is* continuous, *iff for all non-empty chains in $S \subseteq A$, $f(\bigsqcup_A S) = \bigsqcup_B \{fs \mid s \in S\}$.*

We denote the function space of continuous functions from A to B by $[A \xrightarrow{c} B]$. [Coq Definition]

Again, the function space $[A \xrightarrow{c} B]$ is a CL wrt the standard lifting of the partial order from B. [Coq Proof]

At this point, recall the definition of a cHa and note that it entails that the binary meet operator $\sqcap$ in the underlying CL is continuous. There is another CL-structure in which the operators of the underlying CL are continuous; continuous lattices [Scott, 1971] are constructed by forming ideals with respect to an auxiliary relation, the 'well-below' relation. However, we opt for a formulation in terms of cHas, since they are simpler in this context: implementing continuous lattices would require defining additional structure and proving that it entails the continuity of lattice operators. A cHa, on the other hand, is really nothing other than an additional axiom on a CL.

Finally, the dual notion of continuity is co-continuity, or meet-preservation:

Definition 5 (co-continuity). *Given a CL wrt $\sqsubseteq_A$, A, and a CL wrt $\sqsubseteq_B$, B, a function $f : A \xrightarrow{m} B$ is* co-continuous, *iff for all $S \subseteq A, f(\bigsqcap_A S) = \bigsqcap_B \{fs \mid s \in S\}$.*

We denote the function space of co-continuous functions from A to B by $[A \xrightarrow{co-c} B]$. [Coq Definition]

The pragmatic message to take away from these definitions and their connections with fixpoint theorems is that, to prove existence of a least or greatest fixpoint, it is sufficient to show that a function is monotone. To prove interesting things about these fixpoints, one generally needs to show that the operator is continuous or co-continuous.

From a partial order to a cHa.

Abstract interpreters for logic programs are normally based on domains of constraints over vectors of variables (see e.g. [de la Banda et al., 1996] for a survey and an explanation of such constraint domains). Constraints are naturally ordered by entailment, which constitutes a partial order. One would think that requiring constraint domains to be CLs or even cHas puts limits on their applicability. However, the mechanism of downward closure constructs CLs and even cHas from partially ordered sets. It is standardly applied in the context of logic programming semantics to construct CLs from constraint domains. However, downward closure also constructs cHas, and a cHa is sufficient to establish continuity of operators defined in terms of $\sqcap$; a step that has previously been overlooked.

Definition 6 (closure operator & closed sets). *Given a set S, a* closure operator *$c : \mathcal{P}(S) \xrightarrow{m} \mathcal{P}(S)$ is any monotone function which has the following two properties:*

$$\forall S' \subseteq S, S' \subseteq c(S') \qquad \textit{(c is extensive)}$$
$$\forall S', \subseteq S, c(S') = c(c(S')) \qquad \textit{(c is idempotent)}$$

[Coq Definition]

We call a subset $S' \subseteq S$ closed *under c, iff $c(S') = S'$.*

Lemma 3 (Closed Subsets form a CL). *Given a partially ordered set S and a closure operator c, the set of subsets of S closed under c, $\{S' \mid S' \subseteq S \wedge c(S') = S'\}$, is a CL with respect to $\subseteq$, where $\bigsqcup \mathcal{S} := c(\bigcup \mathcal{S})$ and $\bigsqcap := \bigcap$.* [Coq Proof]

Definition 7 (downward closure & ideals). *Given a partially ordered set S, the* downward closure *operator, $\downarrow : \mathcal{P}(S) \xrightarrow{m} \mathcal{P}(S)$, defined as follows, is a closure operator: $\downarrow S' := \{x \mid \exists s \in S',\ x \sqsubseteq s\}$.* [Coq Definition] *We call a set closed with respect to $\downarrow$ an* ideal.

By Lemma 3, given a partially ordered set S, the set of ideals is a CL with respect to $\subseteq$. Since that CL is based on $\mathcal{P}(S)$, and in particular, its join and meet are simply the set-theoretic union and intersection, the lattice thus constructed is in fact a cHa:

Lemma 4 (Ideals form a cHa). *A CL constructed as in Lemma 3 is a cHa.* [Coq Proof]

Sticky Domains and Functions.

Since the results we present in Section 4 concern the relative correctness and precision of different styles of semantics, we require a way of telling when two semantics observe the same property. For this purpose, we follow [King and Lu, 2003] and include a general way of asserting propositions in programs (the ask-constructor, see Section 3), and observe their violation. The property all semantics observe whether a program contains an assertion that is not met.

To reflect these assertion violations in the semantic domains, we use so-called sticky domains [Hudak, 1987]. Sticky domains are a clean mathematical formulation which allow the fixpoint computation in an abstract interpreter to continue when a violated assertion is encountered. The alternative is to abort the fixpoint computation at this stage, which makes both the implementation and the correctness argument messy. The idea of a sticky domain is to extend a set A by a special value $\widehat{\top}$, called 'sticky top', which is formally above all other elements of A, and is intended to be returned by a semantic operator only when an asserted proposition is not true. We denote $A \cup \{\widehat{\top}\}$ by $\widehat{A}$. Unsurprisingly, if A is a CL or a cHa, its structure is not affected by this extension. Once an assertion has been violated, i.e. semantic operator has returned $\widehat{\top}$, that fact is propagated onward. To characterise functions that respect this structure, we introduce the space of sticky functions, which are continuous functions that propagate $\widehat{\top}$:

Definition 8 (sticky function). *Given a CL wrt $\sqsubseteq_A$, A, and a function $f : \widehat{A} \xrightarrow{c} \widehat{A}$ is* sticky, *iff $f\widehat{\top} = \widehat{\top}$.*

We denote the function space of sticky functions over $\widehat{A}$ by $[\widehat{A} \xrightarrow{s} \widehat{A}]$. [Coq Definition]

$[\widehat{A} \xrightarrow{s} \widehat{A}]$ is a CL wrt the standard lifting of the partial order from $\widehat{A}$, where $\bot = \lambda a$, if $a = \widehat{\top}_A$ then $\widehat{\top}_A$ else $\bot_A$. [Coq Proof]

Sticky functions are an elegant solution to a problem we encountered when first starting this formalisation: the proof of correctness between the forward and backward semantics (see Theorem 3) was incomplete [King and Lu, 2003], because the base case did not go through. The definition of a function space in which the bottom function preserves $\widehat{\top}$ solves this problem which Coq exposed.

2.2 Base Domains

In this subsection we define the concrete and the abstract domains which form the basis for the syntactic and semantic constructions in the rest of the paper. It can be read as a commentary on BaseDomains.v ([Coq Definitions]).

Concrete Domain.

As mentioned above, the concrete domain is based on a domain of constraints over vectors of variables $\mathcal{C}on$, partially ordered by entailment. Note that this set is implicitly sorted into subsets, based on the size of the vector: there are two constraints over zero variables, ${}_0\mathcal{C}on = \{true, false\}$, there constraints over a single variable ${}_1\mathcal{C}on$, over pairs of variables ${}_2\mathcal{C}on$, and so on. Since our formalisation uses a dependently typed representation of vectors, the representation of the type of constraints is also as a dependent type. In fact, this dependency is propagated from here on all the way through to the type of a program (see Section 3); everything is really a family of things dependent on an arity (a natural number). Hence we assume a family of partially ordered types (constraints), depending on the natural numbers (the number of constrained variables). This assumption is represented as a parameter in the Coq scripts, with the code looking thus:

```
Parameter constraint : ∀(n : nat), Type.
Parameter constraint_Partial_Order ∀n, Poset.t (constraint n).
```

Since we will be working in a parametric (higher-order) setting (see Section 3.1), we require a parametric notion of constraint. The idea is simply to bind the free variables in a constrained with a λ; e.g. $x = [], y = z$ becomes $\lambda xyz, x = [], y = z$. We call the thus constructed family of types $\mathcal{PC}$. Like $\mathcal{C}on$, $\mathcal{PC}$ is implicitly sorted into subsets, with each ${}_ne \in {}_nPC$ expecting a vector of n variables (VarVec n), and returning a constraint over them.

```
Definition par_constraint (n : nat) := VarVec n → constraint n.
```

Note that each ${}_ne$ has infinitely many semantically equivalent 'siblings' of higher arity: $\lambda x, x = [] \in {}_1PC$ is equivalent to $\lambda xy, x = [] \in {}_2PC$ and $\lambda xyz, x = [] \in {}_3PC$ and so-forth.

Finally, we define the family of domains $\mathcal{C}$ from $\mathcal{C}on$, which by Lemmas 3 and 4 is a CL wrt $\subseteq$ and a cHa:

```
Definition C (n : nat) :=
  closedsubset (down_closure (constraint_Partial_Order n)).
```

Abstract Domain.

Our development is parameterised by the abstract domain D, the requirements being that it be a CL, and that that CL be a cHa:

```
Parameter D : Type.
Parameter CL_D : CompleteLattice.t D.
Parameter cHA_D : CompleteHeytingAlgebra.t D CL_D.
```

Note that D is the only construct without an arity.

Like the concrete domain $\mathcal{C}$, the abstract domain D is parametrised by vectors of variables to construct a family of domains $\mathcal{PD}$, depending on the arity of the vector of variables:

```
Definition par_D (n : nat) := VarVec n → D.
```

Finally, we require the abstract domain D to be related to each concrete domain ${}_nC$ by a Galois connection $\langle {}_nC, D, {}_n\alpha, {}_n\gamma \rangle$, where ${}_n\alpha : {}_nC \xrightarrow{m} D$ and ${}_n\gamma : D \xrightarrow{m} {}_nC$.

3. SYNTAX

The syntax we will be working with describes somewhat normalised pure Prolog programs. This normalisation is achieved by three transformations:

(a) All clauses of a predicate are renamed to align the variables in their heads, and then contracted into a single clause, using disjunction ; in the clause. Hence, every predicate is defined by exactly one clause.

(b) All free variables occurring in the body of a predicate are bound by adding them as parameters to the head. This increases the arity of the predicate.

(c) Throughout a program P, the arities of all predicates are normalised to the maximum arity occurring in P. This is done by adding unconstrained 'dummy' parameters, similar to constructing equivalent siblings of parametric constraints (see Section 2.2).

These transformations make the dependent typing we are using 'harmless' (see Section 5). Reasoning with dependent types in Coq can be a non-trivial undertaking. The above transformations push the dependence all the way up to the level of programs. This way, a program has an arity n, and that arity determines the arity of all its predicates and of all the constraints occurring in their bodies. This means we know up-front which subset of $\mathcal{C}$, $\mathcal{PC}$, and $\mathcal{PD}$ we will be working with, and never need to 'cast' constraints or elements of $\mathcal{PD}$ between them. From here on, we index all constructs that have an arity by a subscript indicating that arity. Since it will be the same everywhere, it will never play a role, other than to remind us that we are in fact constructing families of definitions and proofs, dependent on this arity.

A normalised program of arity n is constructed from agents of arity n (where ${}_nc \in {}_nC$ and $d \in D$):

$$
\begin{aligned}
{}_nA := {}_n\mathsf{ask}\ d \\
\mid {}_n\mathsf{tell}\ {}_nc \\
\mid {}_n\mathsf{conj}\ {}_nA_1\ {}_nA_2 \\
\mid {}_n\mathsf{disj}\ {}_nA_1\ {}_nA_2 \\
\mid {}_n\mathsf{head}\ p\ {}_n\vec{x}
\end{aligned}
$$

Note that ask is included in order to be able to observe assertion violations, which will form the basis of the correctness and precision theorems presented in Section 4. The statement $\mathsf{ask}_n d$ is intended to mean something akin to '$\mathsf{assert}_n d$'. The behaviour of e.g. $\mathsf{ask}(\mathsf{x} = [])$ is to check whether x is the empty list at this point in the program. If it is, continue execution; if it is not abort execution with a message like "assertion violated at line...: x = []"; in the operational semantics this situation is characterised by $\widehat{\top}$.

A predicate definition of an identifier p, a vector of parameters ${}_n\vec{x}$, and a body, which is simply an agent: $p({}_n\vec{x}) \leftarrow {}_nA$. A program is simply a finite list of predicates.

This syntax constructs an abstract syntax tree (AST) which contains a number of free variables.

Example 1 (Standard AST for `append`). *For example, the append predicate is normalised as follows:*

```
app(x,y,z,h,u,w):- x = [], z = y ;
                   x = [h | u], z = [h | w],
                     app(u,y,w,_, _, _).
```

app	$(\vec{x})$	$\leftarrow$	
disj	conj	tell	$x = []$
		tell	$z = y$
	conj	tell	$x = [h\|u]$
		conj	tell $z = [h\|w]$
			head app $(u, y, w, _, _, _)$
		where	$\vec{x} = (x, y, z, h, u, w)$

Table 1: AST for `append` **– standard syntax**

app $\leftarrow$				
	DISJ	CONJ	TELL	$\lambda\vec{x}, x = []$
			TELL	$\lambda\vec{x}, z = y$
		CONJ	TELL	$\lambda\vec{x}, x = [h\|u]$
			CONJ	TELL $\lambda\vec{x}, z = [h\|w]$
				HEAD app
			where	$\vec{x} = (x, y, z, h, u, w)$

Table 2: AST for `append` **– HOAS**

The AST for this constructed by the above syntax is shown in Table 1.

When reasoning about a program represented in this way, one has to consider issues of freshness, and carefully map variables onto each other while not capturing and thus over-constraining variables. These issues are standardly handled by using projection and renaming operators, and the theory of these is rigorous and well-developed [Giacobazzi, 1993]. Since projection is an approximation, different styles of semantics require different projections: a semantics defined as a least fixpoint needs an over-approximating projection onto a vector of variables $\vec{x}$ ($\overline{\exists}_{\vec{x}}$), one defined as a greatest fixpoint needs an under-approximating projection ($\overline{\forall}_{\vec{x}}$). These operators need to be linked by a Galois connection in order for standard correctness and precision theorems to hold between the respective semantics [King and Lu, 2003]. In short, the occurrence of free variables in the syntactic constructs necessitate a considerable amount of mathematical machinery and put certain constraints on the underlying domains. The functional setting of Coq offers an alternative way to approach this problem.

3.1 Higher-Order Abstract Syntax

The idea is to construct an AST without free variables. In this syntactic construction renaming becomes unnecessary and is replaced by function application. Since the latter is a native concept of Coq, we get a whole theory hidden 'for free' – we simply do not have to worry about variable names in the proofs. A syntax that constructs such an AST is called a Higher-Order Abstract Syntax (HOAS).

The concept of a HOAS was introduced in the context of logical frameworks [Pfenning and Elliott, 1988]. It has recently been applied in the context of analysis of functional programs. In that context, it offers a way of avoid complications in proof structure arising from the possibility of name capturing [Chlipala, 2008]. A common way of implementing an HOAS is by using De Bruijn indices to make the scopes of variables explicit [Bruijn, 1972]. The case in logic programming is somewhat simpler than the functional case. In particular, the defined type (${}_nPA$ below) does not occur as the argument of a negatively occuring function type, which is a common characteristic of a HOAS for other languages. Though it is conceptionally simpler than common HOASs, the syntax we define below carries considerable benefits for the proofs of semantic theorems such as those in Section 4.

The choice to use an HOAS means we shift our focus from agents to parametric agents: a parametric agent is a function from a vector of variables to an agent: ${}_nPA : {}_nVarVec \rightarrow {}_nA$. A parametric agent is a closed object (in the sense that it does not contain free variables), which constructs an agent. An agent constructed in this way is an open object; it contains exactly those free variables, that were handed to the parametric agent. To construct such parametric agents, we amend the syntax in three ways:

(a) The constructors ask, tell, conj, disj, and head are substituted by ASK, TELL, CONJ, DISJ, and HEAD, which construct parametric agents, i.e. expect a further parameter ${}_n\vec{v}$.

(b) ASK (resp. TELL) expects a (closed) parametric abstract domain element (resp. parametric constraint), not an (open) abstract domain element (resp. (open) constraint).

(c) CONJ and DISJ expect parametric agents.

The new syntax looks thus (where ${}_ne \in {}_nPC$ and ${}_na \in {}_nPD$):

$$
\begin{aligned}
{}_nPA := {}&{}_n\text{ASK}\ {}_na \\
&\mid {}_n\text{TELL}\ {}_ne \\
&\mid {}_n\text{CONJ}\ {}_nPA_1\ {}_nPA_2 \\
&\mid {}_n\text{DISJ}\ {}_nPA_1\ {}_nPA_2 \\
&\mid {}_n\text{HEAD}\ p
\end{aligned}
$$

[Coq Definition]

Finally, predicates are now defined by an identifier and a parametric agent: $p \leftarrow {}_nPA$. That is to say, the body of a predicate is a function from a vector of parameters to an agent over these parameters.

Example 2 (HOAS AST for `append`). *The new AST constructed for* app *is shown in Table 2. Note that all variables here are explicitly bound by λs. When implementing such a syntax, the additional information about the scope of each variable can be represented using De Bruijn indices [Bruijn, 1972].*

There are two important differences between the ASTs in Table 1 and Table 2:

(a) **free variables vs explicit scoping** The AST in Table 1 contains free variables. In contrast, in Table 2 all variables in the constraints under TELL-constructors and the abstract domain elements under ASK-constructors are bound by a λ. Recall that the entire AST is interpreted as a function; when applied to a vector $\vec{x}$, CONJ and DISJ pass $\vec{x}$ down to their component parametric agents. Thus $\vec{x}$ is propagated to the leafs of a tree, where it is either used as an argument for the parametric abstract domain element under an ASK or for the constraint under a TELL, or is passed on as the parameter for the parametric agent that is the body of the predicate called by HEAD. This propagation is done in the semantic operators described in the Section 4.

(b) **parametric predicate call** In Table 2, there is no vector of variables under the HEAD-constructor. This is where the true advantage of using a HOAS for semantic proofs lies: because there are no parameters *named* under the HEAD-constructor, there are no variable names to be used for renaming in a predicate call. Put differently: because the body of `app` is a parametric agent, it does not contain any free variables. There is therefore no need to project or rename anything in it when calling `app`. When the entire AST is applied to a concrete vector of parameters $\vec{x}$, $\vec{x}$ is propagated throughout, without ever being touched (again, see Section 4).

3.2 Lifted Domains

The domains for semantic operators are mapping from syntax to base domains. These liftings define the following function spaces:

DEFINITION 9 ($Mono-$ AND $Poly-$ FUNCTION SPACES). *One lifting step constructs mappings from agents to concrete or abstract domain elements:*

$$ {}_nMono_C := {}_nA \to {}_nC, \text{ and} $$
$$ {}_nMono_D := {}_nA \to D. $$

Two steps construct mappings from agents to sticky functions:

$$ {}_nPoly_C := [{}_nA \to (\widehat{{}_nC} \xrightarrow{s} \widehat{{}_nC})], \text{ and} $$
$$ {}_nPoly_D := [{}_nA \to (\widehat{D} \xrightarrow{s} \widehat{D})]. $$

[COQ DEFINITIONS]

These function spaces inherit the structure of their codomains, and therefore are CLs.

4. FORMAL SEMANTICS

In this section we finally present formalisations of semantic operators and results about their relative correctness and precision. The definition are identical to those in [King and Lu, 2003], except for the fact that they work over the HOAS defined in the last section. That is to say, in all these definitions, the third parameter to the semantic operator is an n-ary *agent*, constructed by applying an n-ary *parametric agent* to a vector of n variables $\vec{x}$. The effects of this difference will quickly become apparent.

4.1 Forward Semantics

As mentioned previously (Section 2), forward semantics are generally operational, and construct mappings that simulate the effect that the execution of a goal would have on a given state. Their domains therefore are ${}_nPoly_C$ and ${}_nPoly_D$.

4.1.1 Concrete Forward Semantics

DEFINITION 10 (CONCRETE FORWARD SEMANTICS). *The concrete forward semantics is defined as the least fixpoint of the operator ${}_n\mathcal{F}_C : {}_nPoly_C \xrightarrow{c} {}_nPoly_C$, which is defined by the equations below, where ${}_nP$ is an n-ary program, ${}_nf : {}_nPoly_C$, ${}_na \in {}_nPD$, ${}_ne \in {}_nPC$, ${}_n\kappa$ is an n-ary parametric agent, and ${}_n\vec{x}$ is a vector of n variables.*[3]

[3] For the sake of legibility, arity-subscripts are omitted in the equations defining the semantic operators in this section. The reader is asked to keep in mind that all symbols are implicitly subscripted, indicating that they are of arity n.

$$
\begin{aligned}
&\mathcal{F}_C\ (P,\ f,\ \text{ASK}\ a\ \vec{x}) \ = \\
&\qquad \lambda\, c.\, \texttt{if}\ \alpha(c) \sqsubseteq a(\vec{x})\ \texttt{then}\ c\ \texttt{else}\ \widehat{\top} \\
&\mathcal{F}_C\ (P,\ f,\ \text{TELL}\ e\ \vec{x}) \ = \\
&\qquad \lambda\, c.\, \texttt{if}\ c = \widehat{\top}\ \texttt{then}\ \widehat{\top}\ \texttt{else}\ \downarrow\!\{e(\vec{x})\} \sqcap c \\
&\mathcal{F}_C\ (P,\ f,\ \text{CONJ}\ \kappa_1\ \kappa_2\ \vec{x}) \ = \\
&\qquad \lambda\, c.\, f\ \kappa_2(\vec{x})\ (f\ \kappa_1(\vec{x})\ c) \\
&\mathcal{F}_C\ (P,\ f,\ \text{DISJ}\ \kappa_1\ \kappa_2\ \vec{x}) \ = \\
&\qquad \lambda\, c.\, (f\ \kappa_1(\vec{x})\ c) \sqcup (f\ \kappa_2(\vec{x})\ c) \\
&\mathcal{F}_C\ (P,\ f,\ \text{HEAD}\ p\ \vec{x}) \ = \lambda\, c.\, f\ \kappa(\vec{x})\ c \\
&\qquad\qquad\qquad \textit{where}\ p \leftarrow \kappa \in P
\end{aligned}
$$

[COQ DEFINITION]

The fundamental difference between this definition and its original version in [King and Lu, 2003] is the presence of a vector of variables ${}_n\vec{x}$ in each case. Let us therefore consider the role that ${}_n\vec{x}$ plays:

(a) In the first two cases for ${}_n$ASK (resp. ${}_n$TELL), ${}_n\vec{x}$ is used to derive elements of D (resp. ${}_nC$), as required by the semantics, from parametric elements of ${}_nPD$ (resp. ${}_nPC$), which occur in the syntactic constructs.

(b) The third and fourth case for ${}_n$CONJ and ${}_n$DISJ are recursive, i.e. they apply ${}_nf$ to construct a new ${}_nf'$. Here ${}_n\vec{x}$ is simply propagated to the two parametric agents ${}_n\kappa_1$ and ${}_n\kappa_2$ from which the compositional agent is constructed.

(c) The final case is probably the most puzzling to anyone not familiar with HOAS; it is important to remember that predicates in ${}_nP$ are defined by a predicate head p and a parametric agent ${}_n\kappa$ and that ${}_n\kappa$ *does not contain free variables.* Rather, ${}_n\kappa$ is a function from a vector of variables to an agent constraining only those variables. Since there are no variable names in the definition of p, no renaming is required: when simulating a predicate call, the semantics simply takes a parametric body and constructs the right instance of that body for the given call.
Consider again the sample ASTs in Tables 1 and 2: passing ${}_n\vec{x}$ to the body of `app` as shown in Table 2, constructs the same agent that would be the result of renaming the free variables in the body of `app` as defined in Table 1 to the variables in ${}_n\vec{x}$.

The reader will agree that the move to HOAS hardly makes the first four cases more complicated. The last, however, is simplified significantly. To compare. consider that line as it occurs in the original definition in [King and Lu, 2003]:

$$
\begin{aligned}
&\mathcal{F}_C\ (P,\ f,\ \mathsf{head}\ p\ \vec{x}) = \\
&\qquad \lambda c.\ \bigsqcup\{f\ A\ (\downarrow\!(\{\vec{x} = \vec{y}\})\widehat{\sqcap} c \mid p(\vec{y}) \leftarrow A \ll_{p(\vec{x},c}\ P\}
\end{aligned}
$$

The HOAS approach releaves us of any worry concerning variables names and renders the case of a predicate call no more conceptually involved than that of a disjunctive agent.

4.1.2 Abstract Forward Semantics

DEFINITION 11 (ABSTRACT FORWARD SEMANTICS). *The abstract forward semantics is defined as the least fixpoint of the operator ${}_n\mathcal{F}_D : {}_nPoly_D \xrightarrow{c} {}_nPoly_D$, which is defined by the equations below, where ${}_nP$ is an n-ary program, ${}_nf : {}_nPoly_D$, ${}_na \in {}_nPD$, ${}_ne \in {}_nPC$, ${}_n\kappa$ is an n-ary parametric agent, and ${}_n\vec{x}$ is a vector of n variables.*

$$
\begin{aligned}
&\mathcal{F}_D\ (P,\ f,\ \textsc{ask}\ a\ \vec{x}) \ = \\
&\qquad \lambda\, d.\, \texttt{if}\ d \sqsubseteq a(\vec{x})\ \texttt{then}\ d\ \texttt{else}\ \widehat{\top} \\
&\mathcal{F}_D\,(P,\ f,\ \textsc{tell}\ e\ \vec{x}) \ = \\
&\qquad \lambda\, d.\, \texttt{if}\ d = \widehat{\top}\ \texttt{then}\ \widehat{\top}\ \texttt{else}\ \alpha(\downarrow\!\{e(\vec{x})\}) \sqcap d \\
&\mathcal{F}_D\,(P,\ f,\ \textsc{conj}\ \kappa_1\ \kappa_2\ \vec{x}) \ = \\
&\qquad \lambda\, d.\, f\ \kappa_2(\vec{x})\ (f\ \kappa_1(\vec{x})\ d) \\
&\mathcal{F}_D\,(P,\ f,\ \textsc{disj}\ \kappa_1\ \kappa_2\ \vec{x}) \ = \\
&\qquad \lambda\, d.\, (f\ \kappa_1(\vec{x})\ d) \sqcup (f\ \kappa_2(\vec{x})\ d) \\
&\mathcal{F}_D\,(P,\ f,\ \textsc{head}\ p\ \vec{x}) \ = \lambda\, d.\, f\ \kappa(\vec{x})\ d \\
&\qquad\qquad\qquad\qquad \textit{where}\ p \leftarrow \kappa \in P
\end{aligned}
$$

[Coq Definition]

Again, to assess the effect of the HOAS approach, consider the case for a predicate call, as it appears in the original definition in [King and Lu, 2003] (where $\rho_{\vec{x},\vec{y}}$ is the renaming operator substituting $\vec{x}$ with $\vec{y}$):

$\mathcal{F}_D\,(P,\ f,\ \mathsf{head}\, p\, \vec{x}) =$
$\quad \lambda d.\rho_{\vec{y},\vec{x}}(\overline{\exists}_{\vec{y}}(f\ A\ (\rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}(d)))))\widehat{\sqcap}$ where $p(\vec{y}) \leftarrow A \ll_{p(\vec{x})} P$

The following result states that $\mathcal{F}_D$ is a sound approximation of $\mathcal{F}_C$, that is to say that, for any program ${}_nP$ if $\mathsf{lfp}\ (\mathcal{F}_D\, {}_nP)$ does not observe an assertion violation, than $\mathsf{lfp}\ (\mathcal{F}_C\, {}_nP)$ does not observe such a violation, either (compare [King and Lu, 2003, Proposition 1]).

Lemma 5 (${}_n\mathcal{F}_D$ is correct wrt ${}_n\mathcal{F}_C$). *For any arity n, program ${}_nP$, agent ${}_nA$ and ${}_nc \in {}_nC$,*

$(\mathsf{lfp}\ ({}_n\mathcal{F}_D\ {}_nP))\ {}_nA\ \alpha({}_nc) \neq \widehat{\top} \Longrightarrow (\mathsf{lfp}\ ({}_n\mathcal{F}_C\ {}_nP))\ {}_nA\ {}_nc \neq \widehat{\top}$.

[Coq Proof]

4.2 Bottom-Up Semantics

Bottom-up semantics capture the behaviour of a goal in a compositional fashion, by constructing a summary of the its success patterns (see Section 2):

Definition 12 (bottom-up semantics). *The bottom-up semantics defined as the least fixpoint of the operator ${}_n\mathcal{S}_D : {}_nMono_D \xrightarrow{c} {}_nMono_D$, shown below, where ${}_nP$ is an n-ary program, ${}_nf : {}_nMono_D$, ${}_na \in {}_nPD$, ${}_ne \in {}_nPC$, ${}_n\kappa$ is an n-ary parametric agent, and ${}_n\vec{x}$ is a vector of n variables.*

$$
\begin{aligned}
&\mathcal{S}_D\ (P,\ f,\ \textsc{ask}\ a\ \vec{x}) \ = \top \\
&\mathcal{S}_D\,(P,\ f,\ \textsc{tell}\ e\ \vec{x}) \ = \alpha(\downarrow\!\{e(\vec{x})\}) \\
&\mathcal{S}_D\,(P,\ f,\ \textsc{conj}\ \kappa_1\ \kappa_2\ \vec{x}) \ = (f\ \kappa_1(\vec{x})) \sqcap (f\ \kappa_2(\vec{x})) \\
&\mathcal{S}_D\,(P,\ f,\ \textsc{disj}\ \kappa_1\ \kappa_2\ \vec{x}) \ = (f\ \kappa_1(\vec{x})) \sqcup (f\ \kappa_2(\vec{x})) \\
&\mathcal{S}_D\,(P,\ f,\ \textsc{head}\ p\ \vec{x}) \ = f\ \kappa(\vec{x}) \\
&\qquad\qquad\qquad\qquad \textit{where}\ p \leftarrow \kappa \in P
\end{aligned}
$$

[Coq Definition]

Again, allow us to compare the final case above to its original in [King and Lu, 2003]:

$\mathcal{S}_D\,(P,\ f,\ \mathsf{head}\, p\, \vec{x}) =$
$\quad \rho_{\vec{y},\vec{s}}(\overline{\exists}_{\vec{y}}(f\ A))$ where $p(\vec{y}) \leftarrow A \ll_{p(\vec{x})} P$

The following two lemmas show that ${}_n\mathcal{S}_D$ correctly characterises ${}_n\mathcal{F}_D$ except when an assertion is violated (compare [King and Lu, 2003, Lemma 3, Lemma 2]). Lemma 6 states that $\mathcal{S}_D$ constructs a safe under-approximation of $\mathcal{F}_D$.

Lemma 6 (${}_n\mathcal{S}_D$ under-approximates ${}_n\mathcal{F}_D$). *For any arity n, program ${}_nP$, agent ${}_nA$ and $d \in D$,*

$d \sqcap (\mathsf{lfp}\ ({}_n\mathcal{S}_D\ {}_nP))\ {}_nA \sqsubseteq (\mathsf{lfp}\ ({}_n\mathcal{F}_D\ {}_nP))\ {}_nA\ d$.

[Coq Proof]

Lemma 7 states that, if no assertion is violated ${}_n\mathcal{S}_D$ constructs a safe over-approximation of ${}_n\mathcal{F}_D$. In combination with Lemma 6 it amounts to showing that ${}_n\mathcal{S}_D$ and ${}_n\mathcal{F}_D$ are equivalent, as long as no assertion is violated.

Lemma 7 (${}_n\mathcal{S}_D$ over-approximates ${}_n\mathcal{F}_D$). *For any arity n, program ${}_nP$, agent ${}_nA$ and $d \in D$,*

$(\mathsf{lfp}\ ({}_n\mathcal{F}_D\ {}_nP))\ {}_nA\ d \neq \widehat{\top} \Longrightarrow$
$\quad (\mathsf{lfp}\ ({}_n\mathcal{F}_D\ {}_nP))\ {}_nA\ d \sqsubseteq d \sqcap (\mathsf{lfp}\ ({}_n\mathcal{S}_D\ {}_nP))\ {}_nA$.

[Coq Proof]

4.3 Backward Semantics

Finally, a backward semantics derives pre-conditions sufficient for the satisfaction of some requirement by propagating information backwards against the execution order. Since these pre-conditions should be as weak as possible, they are computed by iterating down the lattice, starting from $\top$, until a sound condition is found. The semantics is therefore defined as a *greatest*, not a least fixpoint:

Definition 13 (backward semantics).
Our backward semantics is defined as the greatest fixpoint of the operator ${}_n\mathcal{B}_D : {}_nMono_D \xrightarrow{c} {}_nMono_D$, defined below, where ${}_nP$ is an n-ary program, ${}_nf : {}_nMono_D$, ${}_na \in {}_nPD$, ${}_ne \in {}_nPC$, ${}_n\kappa$ is an n-ary parametric agent, and ${}_n\vec{x}$ is a vector of n variables.

$$
\begin{aligned}
&\mathcal{B}_D\ (P,\ f,\ \textsc{ask}\ a\ \vec{x}) \ = a(\vec{x}) \\
&\mathcal{B}_D\,(P,\ f,\ \textsc{tell}\ e\ \vec{x}) \ = \top \\
&\mathcal{B}_D\,(P,\ f,\ \textsc{conj}\ \kappa_1\ \kappa_2\ \vec{x}) \ = \\
&\qquad (f\ (\kappa_1\ \vec{x})) \sqcap ((\mathsf{lfp}\ \mathcal{S}_D)\ \kappa_1(\vec{x}) \Rightarrow f\ \kappa_2(\vec{x})) \\
&\mathcal{B}_D\,(P,\ f,\ \textsc{disj}\ \kappa_1\ \kappa_2\ \vec{x}) \ = (f\ \kappa_1(\vec{x})) \sqcap (f\ \kappa_2(\vec{x})) \\
&\mathcal{B}_D\,(P,\ f,\ \textsc{head}\ p\ \vec{x}) \ = f\ \kappa(\vec{x}) \\
&\qquad\qquad\qquad\qquad \textit{where}\ p \leftarrow \kappa \in P
\end{aligned}
$$

[Coq Definition]

Note that the final case above is in fact identical to that in the definition of $\mathcal{S}_D$. Its original in terms of projection and renaming looks thus [King and Lu, 2003]:

$\mathcal{B}_D\,(P,\ f,\ \mathsf{head}\, p\, \vec{x}) =$
$\quad \rho_{\vec{y},\vec{s}}(\overline{\forall}_{\vec{y}}(f\ A))$ where $p(\vec{y}) \leftarrow A \ll_{p(\vec{x})} P$

Theorem 3 below states that the pre-conditions calculated by ${}_n\mathcal{B}_D$ precisely characterise those queries for which ${}_n\mathcal{F}_D$ does not observe a violated assertion. That is to say, backwards and forwards analysis are equivalent (compare [King and Lu, 2003, Theorem 3, Theorem 4]).

Theorem 3 (${}_n\mathcal{B}_D$ and ${}_n\mathcal{F}_D$ are equivalent). *For any arity n, program ${}_nP$, agent ${}_nA$ and ${}_nc \in {}_nC$,*

$(\mathsf{lfp}\ ({}_n\mathcal{F}_D\ {}_nP))\ {}_nA\ \alpha({}_nc) \neq \widehat{\top} \Longleftrightarrow \alpha({}_nc) \sqsubseteq (\mathsf{gfp}\ ({}_n\mathcal{B}_D\ {}_nP))\ {}_nA$.

[Coq Proof]

Finally, Theorem 3 in conjunction with Lemma 5 are sufficient to show that ${}_n\mathcal{B}_D$ is correct with respect to the original operational semantics ${}_n\mathcal{F}_C$; that is to say, a call that satisfies the pre-condition derived by ${}_n\mathcal{B}_D$ will not violate an assertion (compare [King and Lu, 2003, Corollary 2]).

Theorem 4 (${}_n\mathcal{B}_D$ is correct wrt ${}_n\mathcal{F}_C$). *For any arity n, program ${}_nP$, agent ${}_nA$ and ${}_nc \in {}_nC$,*

$\alpha({}_nc) \sqsubseteq (\mathsf{gfp}\ ({}_n\mathcal{B}_D\ {}_nP))\ {}_nA \Longrightarrow (\mathsf{lfp}\ ({}_n\mathcal{F}_C\ {}_nP))\ {}_nA\ {}_nc \neq \widehat{\top}$.

[Coq Proof]

5. REFLECTIONS

At this point let us pause to reflect on the question of whether it was worth learning Coq for this. There is no denying that there is a considerable effort required for a practitioner of logic programming semantics to become sufficiently fluent in their use of Coq to be able to construct proofs like the ones presented above. In order to allow a reader to make a somewhat informed decision on this question, this section discusses the difficulties we encountered, and assesses whether the benefits claimed when motivating this work have materialised (see Section 1.1).

5.1 Automated vs Pen-and-Pencil Reasoning

First of all, let us re-iterate the most fundamental point in favour of automation: humans make mistakes. Even though carefully thought through and reviewed, there were some slips in the original formulations of these semantic operators and proofs. As mentioned above (see Section 2), semantic operators that are defined in terms of $\sqcap$ are only continuous when the meet-operator in their domain is continuous, as it is in a cHa [Coq Proof] – a requirement largely overlooked until now. In addition, some of the cases in the proofs did not quite work the way they were stated. (In particular, the base case of the right-to-left direction of Theorem 3 only goes through when the iteration is over the sticky functions, where the bottom function returns $\widehat{\top}$ when given $\widehat{\top}$ [Coq Proof]. In Lemma 7, the induction hypothesis has an antecedent that was not satisfied when applying it in the conjunctive case; the solution there is to prove Lemma 6 first and apply it at this point [Coq Proof].)
vNow, none of these fundamentally compromise the correctness of the results stated. However, a logician will appreciate the appeal of a technique that provides high confidence that such slips do not occur – automation is that technique.

5.2 Difficulties

First of all, writing formal definitions of semantics and theorems like the ones in Section 4 in Coq requires one to learn Gallina, Coq's specification language; proving things about them requires to learn how to 'communicate' mathematical reasoning to Coq in the proof mode by means of tactics. Both requires some getting used to. Secondly, there are two features of Coq which are indispensable for the work we have done and not trivial to utilise: dependent types and type classes.

Dependent Types.

As explained above (see Section 2), formalising logic programming semantics without dependent types is impossible, as far as we can see, because of the omnipresence of vectors of variables. Reasoning about dependent types can be painful in Coq. The reason is that a family of types, depending on the natural numbers, requires families of relations to compare them: the statement '$(x_1, x_2) = (y_1, y_2)$' is really '${}_2(x_1, x_2) {}_2{=} {}_2(y_1, y_2)$'. The statement '${}_2(x_1, x_2) = {}_3(y_1, y_2, y_3)$' is ill-typed and Coq will not accept it. Now, what about the more general statement '${}_n\vec{x} = {}_m\vec{y}$'? It may well be the case that in the context m and n are equal, and therefore the statement that ${}_n\vec{x}$ is equal to ${}_m\vec{y}$ is perfectly valid. However, in order to state it in a way that the type checker will accept, one of the two will have to be cast to the other's type, e.g. ${}_n\vec{x}$ to ${}_m\vec{x}$. Cases like this, in which something has to be cast from one dependent type to another, which is actually the same type, arise frequently and can be frustrating.
However, by constructing our syntax so that there only ever is one subscript in the dependent types (see Section 3), we avoid this difficulty entirely. At this point, therefore, potential future users can benefit from our pain; careful set-up renders dependent types no hindrance to formalising logic programming semantics.

Type Classes.

Type classes [Coq Development Team, 2010, Chapter 18] are effectively a way to do object-oriented programming in Coq. A statement like "*For any set A, and partial order $\sqsubseteq$ over A, a CL is defined as ...*" is formalised using type classes. When using a concrete A later, one defines an instance of the type class CL. Facts about, say, CLs work the same way. Hence when applying a lemma, Coq has to find the right instance of the lemma for the case at hand. That inference is not trivial and in practice does not always work correctly. Where it fails, it is up to the human to specify exactly which instance is required. Having to go deep into these details is painful at first; the continuity proofs for the forward semantics took us a long time partly because of this issue - it is visible in the proofs in form of statements starting `apply (@Lemma ...)`. This is a real difficulty and there seems to be no way around it.

5.3 Benefits

However, the benefits that are bought for this price are considerable.

Renaming vs HOAS.

HOAS really does make the semantics cleaner and the proofs easier; look at the last case in any of the proofs in Theorems34Corollary2.v and you will see that it is straightforward throughout.

Proof Maintenance.

Our experience shows that proof maintenance with Coq is every bit as easy as we hoped it would be; the process of adjusting the proofs to a change in the semantics is very clean. It is difficult to convince others of this subjective experience simply by explaining it. However, we encourage the reader to download the scripts, change a minor detail somewhere (the easiest is probably to change the name of a constructor in the syntax) and then see what happens when re-compiling.

Talking a Common Language.

Finally, a practitioner of logic programming semantics may well feel slightly resentful to have to change the way they are doing maths in order for it to fit into the functional setting of Coq; we certainly were initially annoyed by what we perceived as the unnecessary complication of dependent types. However, we have come to look at this requirement in a more favourable light: it seems desirable to us, for the disciplines of logic and functional programming to work together as closely as possible. In order to do that, we have to talk a common language. Since we can expect further advances in the (functional) proof technology, it seems a good

idea to start doing Prolog semantics in this setting, not only because of the benefits mentioned above, but also in order to be ready to benefit from these advances when they come along.

6. RELATED WORK

Domain Theory in Coq.

As far as we are aware, the only existing Coq formalisation of large parts of domain theory required for logic program analysis is the category-theoretic work by [Benton et al., 2009]. That work is based on ssreflect [Gonthier and Mahboubi, 2010], which is an extension of Coq containing libraries and a tactic language for mathematical reasoning. It was developed for formalising 'real' mathematics [Gonthier, 2007] and [Gonthier et al., 2013], and at the time we started this development, ssreflect was not compatible with Coq 8.4. Since we judged it easier to re-implement the domain theory, than the dependent typing available in Coq 8.4, we chose not to use [Benton et al., 2009].

Abstract interpretation and fixpoint semantics in Coq.

Work has been done on formalising abstract interpretation frameworks in Coq, including recently the following. Members of the Celtique group have developed a fully verified abstract interpreter [Cachera and Pichardie, 2010, Besson et al., 2009, Besson et al., 2006, Cachera et al., 2005] for small imperative languages. In the process of that development, they present formalisations of several fixpoint theorems. Bertot and Komendantsky have presented a way of representing partial and non-terminating functions in Coq [Bertot and Komendantsky, 2008] based on a fixpoint construction, which involves a formalisation of the Knaster-Tarski theorem. Finally, Benton et al. formalised a proof of correctness between an operational and a denotational semantics of a functional language [Benton et al., 2009].

Prolog semantics in Coq.

As mentioned in the introduction, we are not aware on any work on mechanising correctness arguments about logic programming semantics in the abstract interpretation style. In fact, it appears that the only formalisation work that has been done in the context of Prolog is Pusch's verification of a compilation from an operational semantics of Prolog to the Warren Abstract Machine in Isabelle/HOL [Pusch, 1996].

HOAS for Prolog.

Logic programming with higher order abstract syntax is not new. As mentioned above, the idea of HOAS stems from the context of logical frameworks [Pfenning and Elliott, 1988], which is implemented in the Twelf system [Schürmann, 2009]. Other Prolog dialects that are implemented using HOAS are λProlog [Felty et al., 1988] and αProlog [Cheney and Urban, 2004].

7. FURTHER WORK

There is a number of directions we plan to take this work.

Proof Automation.

Given all the basic structure, the next step is start writing some tactics for semantic reasoning to go with the libraries presented. Starting with tactics that take care of the details that proofs from first principles require, so that, e.g. commutativity or associativity of operators need not explicitly be mentioned. That would already shorten the proofs considerably. From there one can proceed to tactics that automatically perform the right induction and case analysis, and take care of easy cases automatically.

A tactic that automatically proofs continuity for operators or relationships between them is unlikely to happen, but we can realistically construct tactics that take care of unnecessary details and leave the user to focus on the interesting cases only.

Applications.

There are several potential applications for the techniques presented here:

Program Transformations Program transformations have been applied widely in logic program analyses. A good example is the magic transformation [Ramakrishnan, 1991], which derives call- and answer-patterns using an bottom-up abstract interpreter. There has been some discussion of the proof of correctness for this transformation, with several 'revised', i.e. simpler, proofs being published recently (see [Drabent, 2012]). The fact that correctness proofs for program transformations can be complex enough to make their simplification interesting in itself, suggests that formalising them may be useful in establishing confidence in their correctness. It would be interesting to see whether the work presented here can be applied to this purpose.
We therefore plan to attempt to prove the (hitherto unproven) correctness of the program transformation for backward analysis presented in [Gallagher, 2003].

Constraint Handling Rules (CHR) Another possible application is to prove correctness of abstractions of CHR programs like the one presented in [Schrijvers et al., 2005]. Since CHR are multi-headed, their semantics are considerably more involved. It would be interesting to see whether our formalisation can handle the added mathematical complexity.

Analysis of Programs containing 'cut' In the same spirit, we plan to attempt to formalise the correctness proofs of our own determinacy analysis for Prolog with 'cut' [Kriener and King, 2011]. The domain underlying the forward semantics comprises sequences of ideals, and the programs are normalised and cut-stratified, and hence the mathematical reasoning is considerably more involved than what is required for the results above.

8. CONCLUSIONS

We have presented formalisations of logic programming semantics in the three most common styles, foward or top-down, bottom-up, and backward, and verified results about them. These semantics are based on a HOAS for Prolog, which replaces projection and renaming operators and thus makes both the semantic operators and the proofs simpler.

More importantly, however, we have provided a framework and given an example of how to formally verify correctness of logic program analysis. We hope that our report on the difficulties we encountered and the benefits we perceive in this move to verified analyses will be interesting to the community in assessing the possibilities for future work.

Acknowledgements.
We would like to thank the following: the annonymous reviewers for their effort and their valuable and constructive comments; the Celtique team for their hospitality, and particularly Frederic Besson for his patience and help; Maurice Bruynooghe for his tenacity as a reviewer of earlier work, which motivated this study; the School of Computing, University of Kent, for funding this work.

9. REFERENCES

[Abramsky and Hankin, 1987] Abramsky, S. and Hankin, C. (1987). *Abstract Interpretation of Declarative Languages*. Ellis Horwood.

[Abramsky and Jung, 1994] Abramsky, S. and Jung, A. (1994). Domain theory. In Abramsky, S., Gabbay, D., and Maibaum, T. S. E., editors, *Handbook of Logic in Computer Science*, pages 1–168. Oxford University Press.

[Barbuti et al., 1993] Barbuti, R., Giacobazzi, R., and Levi, G. (1993). A General Framework for Semantics-Based Bottom-Up Abstract Interpretation of Logic Programs. *ACM TOPLAS*, 15(1):133–181.

[Benton et al., 2009] Benton, N., Kennedy, A., and Varming, C. (2009). Some Domain Theory and Denotational Semantics in Coq. In [Berghofer et al., 2009], pages 115–130.

[Berghofer et al., 2009] Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors (2009). *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*. Springer.

[Bertot and Komendantsky, 2008] Bertot, Y. and Komendantsky, V. (2008). Fixed point semantics and partial recursion in Coq. In Antoy, S. and Albert, E., editors, *PPDP*, pages 89–96. ACM.

[Besson et al., 2009] Besson, F., Cachera, D., Jensen, T. P., and Pichardie, D. (2009). Certified Static Analysis by Abstract Interpretation. In *FOSAD*, volume 5705 of *LNCS*, pages 223–257. Springer.

[Besson et al., 2006] Besson, F., Jensen, T. P., and Pichardie, D. (2006). Proof-carrying code from certified abstract interpretation and fixpoint compression. *TCS*, 364(3):273–291.

[Billaud, 1990] Billaud, M. (1990). Simple Operational and Denotational Semantics for Prolog with Cut. *Theoretical Computer Science*, 71(2):193–208.

[Birkhoff, 1967] Birkhoff, G. (1967). Lattice Theory. In *Colloquium Publications*, volume 25. Amererican Mathematical Society, 3. edition.

[Blanqui and Koprowski, 2011] Blanqui, F. and Koprowski, A. (2011). CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science*, 21(4):827–859.

[Bruijn, 1972] Bruijn, N. G. D. (1972). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *INDAG. MATH*, 34:381–392.

[Cachera et al., 2005] Cachera, D., Jensen, T. P., Pichardie, D., and Rusu, V. (2005). Extracting a data flow analyser in constructive logic. *TCS*, 342(1):56–78.

[Cachera and Pichardie, 2010] Cachera, D. and Pichardie, D. (2010). A certified denotational abstract interpreter. In *Proc. of the conference on Interactive Theorem Proving (ITP-10)*, volume 6172 of *Lecture Notes in Computer Science*, pages 9–24. Springer-Verlag.

[Cheney and Urban, 2004] Cheney, J. and Urban, C. (2004). αProlog: A Logic Programming Language with Names, Binding and α-Equivalence. In Demoen, B. and Lifschitz, V., editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 269–283. Springer.

[Chlipala, 2008] Chlipala, A. (2008). Parametric higher-order abstract syntax for mechanized semantics. In Hook, J. and Thiemann, P., editors, *ICFP*, pages 143–156. ACM.

[Codish et al., 1994] Codish, M., Dams, D., and Yardeni, E. (1994). Bottom-up Abstract Interpretation of Logic Programs. *Theoretical Computer Science*, 124(1):93–125.

[CompCert Development Team, 2012] CompCert Development Team (2012). The CompCert formally verified C compiler. `http://compcert.inria.fr/`.

[Coq Development Team, 2010] Coq Development Team (2010). *The Coq Proof Assistant Reference Manual, Version 8.3*. INRIA. `http://coq.inria.fr/refman/`.

[Cousot and Cousot, 1979] Cousot, P. and Cousot, R. (1979). Systematic Design of Program Analysis Frameworks. In *POPL*, pages 269–282. ACM Press.

[de la Banda et al., 1996] de la Banda, M. J. G., Hermenegildo, M. V., Bruynooghe, M., Dumortier, V., Janssens, G., and Simoens, W. (1996). Global Analysis of Constraint Logic Programs. *ACM Trans. Program. Lang. Syst.*, 18(5):564–614.

[de Vink, 1989] de Vink, E. P. (1989). Comparative Semantics for Prolog with Cut. *Science of Computer Programming*, 13(1):237–264.

[Debray and Mishra, 1988] Debray, S. K. and Mishra, P. (1988). Denotational and Operational Semantics for Prolog. *Journal of Logic Programming*, 5(1):81–91.

[Drabent, 2012] Drabent, W. (2012). A Simple Correctness Proof for Magic Transformation. *Theory and Practice of Logic Programming*, 12:929–936.

[Felty et al., 1988] Felty, A. P., Gunter, E. L., Hannan, J., Miller, D., Nadathur, G., and Scedrov, A. (1988). Lambda-Prolog: An Extended Logic Programming Language. In Lusk, E. L. and Overbeek, R. A., editors, *CADE*, volume 310 of *Lecture Notes in Computer Science*, pages 754–755. Springer.

[Gabbrielli and Levi, 1991] Gabbrielli, M. and Levi, G. (1991). On the Semantics of Logic Programs. In *ICALP*, volume 510 of *LNCS*, pages 1–19. Springer.

[Gallagher, 2003] Gallagher, J. P. (2003). A Program Transformation for Backwards Analysis of Logic Programs. In Bruynooghe, M., editor, *LOPSTR*, volume 3018 of *Lecture Notes in Computer Science*, pages 92–105. Springer.

[Giacobazzi, 1993] Giacobazzi, R. (1993). *Semantic Aspects of Logic Program Analysis*. PhD thesis, Dipartimento di Informatica, Universitá di Pisa.

[Gonthier, 2007] Gonthier, G. (2007). The Four Colour Theorem: Engineering of a Formal Proof. In Kapur, D., editor, *ASCM*, volume 5081 of *LNCS*, page 333. Springer.

[Gonthier et al., 2013] Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Roux, S. L., Mahboubi, A., O'Connor, R., Biha, S. O., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., and Thery, L. (2013). A machine-checked proof of the odd order theorem. In *Proc. of the 4th conference on Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179, Rennes, France. Springer.

[Gonthier and Mahboubi, 2010] Gonthier, G. and Mahboubi, A. (2010). An introduction to small scale reflection in Coq. *Journal of Formalised Reasoning*, 3(2):95–152.

[Hudak, 1987] Hudak, P. (1987). A Semantic Model of Reference Counting and its Abstraction. In *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood.

[Janssens and Bruynooghe, 1992] Janssens, G. and Bruynooghe, M. (1992). Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation. *JLP*, 13(2&3):205–258.

[King and Lu, 2002] King, A. and Lu, L. (2002). A Backward Analysis for Constraint Logic Programs. *TPLP*, 2(4-5):517–547.

[King and Lu, 2003] King, A. and Lu, L. (2003). Forward versus Backward Verification of Logic Programs. In *ICLP*, volume 2916 of *LNCS*, pages 315–330. Springer.

[Kleene, 1938] Kleene, S. C. (1938). On notation for ordinal numbers. *The Journal of Symbolic Logic*, 3(4):pp. 150–155.

[Klein et al., 2010] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2010). SeL4: formal verification of an operating-system kernel. *CACM*, 53(6):107–115.

[Kriener and King, 2011] Kriener, J. and King, A. (2011). RedAlert: Determinacy inference for Prolog. *TPLP*, 11(4-5):537–553. Proofs in CoRR volume abs/1109.2548 at `http://arxiv.org/abs/1109.2548`.

[Leroy, 2009] Leroy, X. (2009). Formal verification of a realistic compiler. *CACM*, 52(7):107–115.

[Levi, 1991] Levi, G. (1991). On the Semantics of Logic Programs. In *ICLP*, page 945. MIT Press.

[Muthukumar and Hermenegildo, 1992] Muthukumar, K. and Hermenegildo, M. V. (1992). Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *JLP*, 13(2&3):315–347.

[Pfenning and Elliott, 1988] Pfenning, F. and Elliott, C. (1988). Higher-Order Abstract Syntax. In Wexelblat, R. L., editor, *PLDI*, pages 199–208. ACM.

[Pusch, 1996] Pusch, C. (1996). Verification of Compiler Correctness for the WAM. In *TPHOLs*, volume 1125 of *LNCS*, pages 347–361. Springer.

[Ramakrishnan, 1991] Ramakrishnan, R. (1991). Magic templates: A spellbinding approach to logic programs. *J. Log. Program.*, 11(3&4):189–216.

[Schrijvers et al., 2005] Schrijvers, T., Stuckey, P. J., and Duck, G. J. (2005). Abstract interpretation for constraint handling rules. In Barahona, P. and Felty, A. P., editors, *PPDP*, pages 218–229. ACM.

[Schürmann, 2009] Schürmann, C. (2009). The Twelf Proof Assistant. In [Berghofer et al., 2009], pages 79–83.

[Scott, 1971] Scott, D. (1971). Continuous lattices. Technical Report PRG07, OUCL.

[Tarski, 1955] Tarski, A. (1955). A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5(2):285–309.

[van Emden and Kowalski, 1976] van Emden, M. H. and Kowalski, R. A. (1976). The Semantics of Predicate Logic as a Programming Language. *JACM*, 23(4):733–742.