# Constructive Negation in Extensional Higher-Order Logic Programming

**Angelos Charalambidis** [*]
Dept. of Informatics and Telecommunications
University of Athens
a.charalambidis@di.uoa.gr

**Panos Rondogiannis**
Dept. of Informatics and Telecommunications
University of Athens
prondo@di.uoa.gr

**Abstract**

Extensional higher-order logic programming has been recently proposed as an interesting extension of classical logic programming. An important characteristic of the new paradigm is that it preserves all the well-known properties of traditional logic programming. In this paper we enhance extensional higher-order logic programming with *constructive negation*. We argue that the main ideas underlying constructive negation are quite close to the existing proof procedure for extensional higher-order logic programming and for this reason the two notions amalgamate quite conveniently. We demonstrate the soundness of the resulting proof procedure and describe an actual implementation of a language that embodies the above ideas. In this way we obtain the first (to our knowledge) higher-order logic programming language supporting constructive negation and offering a new style of programming that genuinely extends that of traditional logic programming.

## 1 Introduction

Extensional higher-order logic programming has been recently proposed (Charalambidis et al. 2010; 2013) as an interesting extension of classical logic programming. The key idea behind the new paradigm is that all predicates defined in a program (even the higher-order ones) denote sets and therefore one can use standard extensional set theory in order to understand the meaning of programs and reason about them. A consequence of this fact is that the semantics and the proof theory of extensional higher-order logic programming smoothly extend the corresponding ones for traditional (ie., first-order) logic programming.

In this paper we extend the new paradigm of extensional higher-order logic programming with *constructive negation* (Chan 1988; 1989). The combination of higher-order characteristics and constructive negation in the new language, allows the programmer to specify in a compact way many interesting problems. For example, in the new language it is quite convenient to enumerate the sets that satisfy a specific property (such as all the colorings of a graph, all the subsets of a set that satisfy a condition, and so on). In order to theoretically justify the proposed higher-order language, we define its *completion semantics* and demonstrate the soundness of the proof procedure with respect to this semantics. Finally, we describe an implementation of the new language which embodies the above ideas. In conclusion, the main contributions of the present work are the following:

- We propose the first (to our knowledge) higher-order logic programming language with constructive negation. The amalgamation of the two ideas seems quite natural and expressive (but is non-trivial from a technical point of view).
- We provide a theoretical justification of our proposal by defining the completion semantics for higher-order logic programs and by demonstrating the soundness of the proposed proof procedure with respect to this semantics. In this way we provide a promising direction for the study of the interplay between higher-order logic programming and negation-as-failure.

The rest of the paper is organized as follows. Section 2 describes the motivation behind the present work. Section 3 provides the syntax and semantics of the proposed higher-order language and Section 4 presents the corresponding proof-procedure and discusses its implementation. Section 5 demonstrates the soundness of the proof-procedure. Section 6 concludes the paper with discussion on future work.

## 2 Motivation

The purpose of this paper is to enhance extensional higher-order logic programming with constructive negation. These two notions prove to be very compatible, for reasons that we are going to explain in detail below.

In (Charalambidis et al. 2013) the higher-order logic programming language $\mathcal{H}$ is defined. Intuitively, $\mathcal{H}$ extends classical logic programming with higher-order user-defined predicates. The main novel characteristic of $\mathcal{H}$ with respect to other existing higher-order logic programming languages (such as Hilog (Chen, Kifer, and Warren 1993) and $\lambda$-Prolog (Miller and Nadathur 1986)) is that it has a purely extensional semantics: program predicates denote sets of ob-

[*]The research of A. Charalambidis has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Heracleitus II. Investing in knowledge society through the European Social Fund.

jects and the semantics of the language is an extension of the traditional semantics of classical logic programming. Since predicates in $\mathcal{H}$ denote sets, the programmer is allowed to use uninstantiated predicate variables in clause bodies and in queries; when the proof-procedure of $\mathcal{H}$ encounters such a variable, it performs a systematic enumeration of the possible values that this variable can take. For further details regarding $\mathcal{H}$ the interested reader should consult (Charalambidis et al. 2013). The following two examples motivate the language. To simplify our presentation for this section, we use an extended Prolog-like syntax. The exact syntax of the language that will be the object of our study will be specified in the next section.

**Example 1.** Consider the program:

```
p(Q):-Q(0),Q(1).
```

The meaning of p can be understood in set-theoretic terms, ie., p is a predicate that is true of all relations that contain (at least) 0 and 1. Thus, it is meaningful to ask the query:

```
?-p(R).
```

which will respond with an answer of the form $\mathtt{R} = \{\mathtt{0}, \mathtt{1}\} \cup \mathtt{L}$ (having the obvious meaning described above).

**Example 2.** Consider the following program stating that a band is a group that has at least a singer and a guitarist:

```
band(B):-singer(S),B(S),guitarist(G),B(G).
```

Suppose that we also have a database of musicians:

```
singer(sally).
singer(steve).
guitarist(george).
guitarist(grace).
```

We can then ask the following query:

```
?-band(B).
```

which returns answers of the form $\mathtt{B} = \{\mathtt{sally}, \mathtt{george}\} \cup \mathtt{L}$, and so on.

It should be noted that in $\mathcal{H}$ one can ask queries with free predicate variables of any *order*; this means that we can have queries that return sets of sets of objects, and so on.

As demonstrated in (Charalambidis et al. 2013), one can define a sound and complete proof procedure for $\mathcal{H}$ and based on it to obtain a reasonably efficient implementation of the language. There are however certain aspects of the language that require further investigation. One basic property of all the higher-order predicates that can be defined in $\mathcal{H}$ is that they are *monotonic*. Intuitively, the monotonicity property states that if a predicate is true of a relation R then it is also true of every superset of R. In the above example, it is clear that if band is true of a relation B then it is also true of any band that is a superset of B. However, there are many natural higher-order predicates that are *non-monotonic*. For example, consider the predicate disconnected(G) which succeeds if its input argument, namely a graph G, is disconnected. A graph is simply a set of pairs, and therefore disconnected is a second-order predicate. Notice now that disconnected is obviously non-monotonic: given graphs G1 and G2 with $\mathtt{G1} \subseteq \mathtt{G2}$, it is possible that disconnected(G1) succeeds but disconnected(G2) fails.

The obvious idea in order to add non-monotonicity to $\mathcal{H}$ is to enhance the language with negation-as-failure. However, this is not as straightforward as it sounds, because even the simpler higher-order programs with negation face the well-known problem of *floundering* (Lloyd 1987). In classical logic programming a computation is said to *flounder* if at some point a goal is reached that contains only non-ground negative literals. In higher-order logic programming this problem is even more extended since a goal may additionally contain uninstantiated higher-order variables.

**Example 3.** Consider the following simple modification to the program of Example 1:

```
p(Q):-Q(0),not(Q(1)).
```

Intuitively, p is true of all relations that contain 0 but they do not contain 1. The query:

```
?-p(R).
```

will not behave correctly if a simple-minded implementation of negation-as-failure is followed; this is due to the fact that in order to answer the given query one has to answer the subquery not(R(1)). Since the relation R is not fully known at the point of this call, it is not obvious how the implementation should treat such a subquery.

*Constructive negation* (Chan 1988; 1989) bypasses the problem of floundering. The main idea can be explained by a simple example. Consider the program:

```
p(1).
p(2).
```

and the query:

```
?-not(p(X)).
```

The original idea (Chan 1988) behind constructive negation is that in order to answer a negative query that contains uninstantiated variables, the following procedure must be applied: we run the positive version of the query and we collect the solutions as a disjunction; we then return the negation of the disjunction as the answer to the original query. This idea can be extended even to the case where a query has an infinite number of answers (Chan 1989). In our example, the corresponding positive query (namely ?-p(X).) has the answers X=1 and X=2. The negation of their disjunction is the conjunction $(\mathtt{X} \neq \mathtt{1}) \wedge (\mathtt{X} \neq \mathtt{2})$. Observe now that the procedure behind constructive negation returns as answers not only substitutions (as it happens in negationless logic programming) but also inequalities.

Generalizing the above idea to the higher-order setting requires the ability to express some form of inequalities regarding the elements of sets. Intuitively, we would like to express the property that some element *does not belong to a set*. For example, given the query in Example 3, a reasonable answer would be $\mathtt{R} = \{\mathtt{0}\} \cup \{\mathtt{X} \mid \mathtt{X} \neq \mathtt{1}\}$.

We start with the syntax of $\mathcal{H}$ as introduced in (Charalambidis et al. 2013) and extend it in order to allow negation in clause bodies. The new language, $\mathcal{H}_{cn}$, has only one limitation with respect to the initial $\mathcal{H}$: the existential predicate variables that appear in queries or in clause bodies of $\mathcal{H}_{cn}$ can only be sets of terms (and not, for example, sets of sets, or sets of sets of sets, and so on). The reasons for our restriction are two-fold: first, it is not straightforward to express inequalities regarding the elements of a set of sets; and second, it would be much more difficult to express the proof procedure if the queries were not restricted (and the implementation would also face significant complications).

It turns out that $\mathcal{H}_{cn}$ is quite appropriate in expressing problems that require a generate-and-test of sets in order to reach a solution. For example, many graph problems require the enumeration and testing of subgraphs of a given graph. In this respect, the language $\mathcal{H}_{cn}$ appears to have an application domain that is similar to that of modern ASP languages, which are being used for specifying computationally difficult problems (Gelfond and Lifschitz 1988; Lifschitz 2008). Of course, the higher-order approach is completely different than the ASP one in terms of syntax and semantics, and a comparative evaluation of the two paradigms deserves further investigation.

**Example 4.** Consider the following program which defines the subset relation:

```
subset(P,Q):-not(non_subset(P,Q)).
non_subset(P,Q):-P(X),not(Q(X)).
```

and assume that we have a predicate q which is true of 0, 1 and 2. Then, given the query

```
?-subset(P,q).
```

the interpreter produces all the subsets of the set $\{0, 1, 2\}$.

**Example 5.** Consider the following program which defines the relation twocolor(G,R) which is true if R is a subset of the set of vertices of G that can be painted with the same color in a two-coloring of G.

```
twocolor(G,R):-subset(R,vertex_set(G)),
               not(non_twocolor(G,R)).

vertex_set(G)(X):-G(X,_).
vertex_set(G)(X):-G(_,X).

non_twocolor(G,R):-G(X,Y),R(X),R(Y).
non_twocolor(G,R):-G(X,Y),not(R(X)),not(R(Y)).
```

Assume that g is a binary predicate defining a graph. By asking the query:

```
?-twocolor(g,R).
```

the implementation will enumerate all the possible relations R that constitute valid two-colorings of g.

**Example 6.** The following program can be used to enumerate all the subgraphs of a given graph that are cliques:

```
clique(G,R):-subset(R,vertex_set(G)),
             not(non_clique(G,R)).

non_clique(G,R):-R(X),R(Y),not(G(X,Y)).
```

## 3 Syntax and Semantics

This section defines in a formal way the syntax and the semantics of the higher-order language $\mathcal{H}_{cn}$.

### The Syntax of $\mathcal{H}_{cn}$

The language $\mathcal{H}_{cn}$ is in fact a more austere version of the ad-hoc language used for the examples of the previous section.

**Example 7.** The subset predicate of Example 4 can be expressed in $\mathcal{H}_{cn}$ as follows:

$$\texttt{subset} \leftarrow \lambda\texttt{P}.\lambda\texttt{Q}.\ {\sim}\exists\texttt{X}((\texttt{P X}) \wedge {\sim}(\texttt{Q X}))$$

The subset predicate is defined by a $\lambda$-expression (which obviates the need to have the formal parameters of the predicate in the left-hand side of the definition). Moreover, in the right-hand side we have an explicit existential quantifier for the variable X (in Prolog, if a variable appears in the body of a clause but not in the head, then it is implicitly existentially quantified). Consider the query:

$$\leftarrow \texttt{subset R p}$$

Notice that here the variable R is not explicitly existentially quantified: we assume that the free variables of the goal are the ones for which the proof procedure will attempt to find bindings in order for the goal to be satisfied. If a variable appears explicitly existentially quantified in the goal, then the implementation will not produce a binding for it.

The type system of $\mathcal{H}_{cn}$ is based on two base types: $o$, the type of the boolean domain, and $\iota$, the type of the domain of individuals (data objects). For example, every classical logic programming term is of type $\iota$.

The basic difference in the types of $\mathcal{H}_{cn}$ from those of $\mathcal{H}$ is the existence of a type $\mu$ which restricts the set of predicate variables that can be existentially quantified or appear free in goal clauses (as explained in the previous section).

**Definition 1.** A type can either be *functional*, *argument*, or *predicate*, denoted by $\sigma$, $\rho$ and $\pi$ and defined as:

$$\begin{array}{lcl} \sigma & := & \iota \mid (\iota \to \sigma) \\ \rho & := & \iota \mid \pi \\ \pi & := & o \mid (\rho \to \pi) \end{array}$$

The subtypes $\mu$ and $\kappa$ of $\rho$ and $\pi$ are defined as follows:

$$\begin{array}{lcl} \mu & := & \iota \mid \kappa \\ \kappa & := & \iota \to o \mid (\iota \to \kappa) \end{array}$$

The types $\mu$ and $\kappa$ will be called *existential* and *set* types respectively.

The operator $\to$ is right-associative. A functional type that is different from $\iota$ will often be written in the form $\iota^n \to \iota$, $n \geq 1$. Similarly, a set type will often be written in the form $\iota^n \to o$, $n \geq 1$. Moreover, it can be easily seen that every predicate type $\pi$ can be written in the form $\rho_1 \to \cdots \to \rho_n \to o$, $n \geq 0$ (for $n = 0$ we assume that $\pi = o$). We proceed with the alphabet of $\mathcal{H}_{cn}$:

**Definition 2.** The *alphabet* of $\mathcal{H}_{cn}$ consists of:

1. *Predicate variables* of every predicate type $\pi$ (denoted by uppercase letters such as $\mathsf{R}, \mathsf{Q}, \ldots$).

2. *Predicate constants* of every predicate type $\pi$ (denoted by lowercase letters such as $\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$).
3. *Individual variables* of type $\iota$ (denoted by uppercase letters such as $\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \ldots$).
4. *Individual constants* of type $\iota$ (denoted by lowercase letters such as $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$).
5. *Function symbols* of every functional type $\sigma \neq \iota$ (denoted by lowercase letters such as $\mathsf{f}, \mathsf{g}, \mathsf{h}, \ldots$).
6. The following logical constant symbols: the propositional constants $\mathsf{false}$ and $\mathsf{true}$ of type $o$; the equality constant $\approx$ of type $\iota \to \iota \to o$; the generalized disjunction and conjunction constants $\bigvee_\pi$ and $\bigwedge_\pi$ of type $\pi \to \pi \to \pi$, for every predicate type $\pi$; the generalized inverse implication constants $\leftarrow_\pi$ of type $\pi \to \pi \to o$, for every predicate type $\pi$; the equivalence constants $\leftrightarrow_\pi$ of type $\pi \to \pi \to o$, for every predicate type $\pi$; the existential quantifiers $\exists_\mu$ of type $(\mu \to o) \to o$, for every existential type $\mu$; the negation constant $\sim$ of type $o \to o$.
7. The abstractor $\lambda$ and the parentheses "(" and ")".

The set consisting of the predicate variables and the individual variables of $\mathcal{H}_{cn}$ will be called the set of *argument variables* of $\mathcal{H}_{cn}$. Argument variables will be usually denoted by $\mathsf{V}, \mathsf{U}$ and their subscripted versions.

**Definition 3.** The set of *body expressions* of the higher-order language $\mathcal{H}_{cn}$ is recursively defined as follows:

1. Every predicate variable (respectively, predicate constant) of type $\pi$ is a body expression of type $\pi$; every individual variable (respectively, individual constant) of type $\iota$ is a body expression of type $\iota$; the propositional constants $\mathsf{false}$ and $\mathsf{true}$ are body expressions of type $o$.
2. If $\mathsf{f}$ is an $n$-ary function symbol and $\mathsf{E}_1, \ldots, \mathsf{E}_n$ are body expressions of type $\iota$, then $(\mathsf{f}\ \mathsf{E}_1 \cdots \mathsf{E}_n)$ is a body expression of type $\iota$.
3. If $\mathsf{E}_1$ is a body expression of type $\rho \to \pi$ and $\mathsf{E}_2$ is a body expression of type $\rho$, then $(\mathsf{E}_1\ \mathsf{E}_2)$ is a body expression of type $\pi$.
4. If $\mathsf{V}$ is an argument variable of type $\rho$ and $\mathsf{E}$ is a body expression of type $\pi$, then $(\lambda \mathsf{V}.\mathsf{E})$ is a body expression of type $\rho \to \pi$.
5. If $\mathsf{E}_1, \mathsf{E}_2$ are expressions of type $\pi$, then $(\mathsf{E}_1 \bigwedge_\pi \mathsf{E}_2)$ and $(\mathsf{E}_1 \bigvee_\pi \mathsf{E}_2)$ are body expressions of type $\pi$.
6. If $\mathsf{E}_1, \mathsf{E}_2$ are expressions of type $\iota$, then $(\mathsf{E}_1 \approx \mathsf{E}_2)$ is a body expression of type $o$.
7. If $\mathsf{E}$ is an expression of type $o$ and $\mathsf{V}$ is an existential variable of type $\mu$, then $(\exists_\mu \mathsf{V}\ \mathsf{E})$ is a body expression of type $o$.
8. If $\mathsf{E}$ is a body expression of type $o$, then $(\sim\mathsf{E})$ is a body expression of type $o$.

The type subscripts (such as $\mu$ in $\exists_\mu$) will often be omitted when they are obvious or immaterial in the foregoing discussion. Moreover, we will write $\wedge$ and $\vee$ instead of $\bigwedge_o$ and $\bigvee_o$. We will use $\mathit{type}(\mathsf{E})$ to denote the type of the body expression $\mathsf{E}$. The notions of *free* and *bound* variables of a body expression are defined as usual. A body expression is called *closed* if it does not contain any free variables. Given an expression $\mathsf{E}$, we denote by $\mathit{fv}(\mathsf{E})$ the set of all free variables of $\mathsf{E}$. By overloading notation, we will also write $\mathit{fv}(S)$, where $S$ is a set of expressions.

We will often write $\hat{\mathsf{A}}$ to denote a (possibly empty) sequence $\langle \mathsf{A}_1, \ldots, \mathsf{A}_n \rangle$ of expressions. For example we will write $(\mathsf{E}\ \hat{\mathsf{A}})$ to denote an application $(\mathsf{E}\ \mathsf{A}_1 \cdots \mathsf{A}_n)$; $(\lambda \hat{\mathsf{X}}.\mathsf{E})$ to denote $(\lambda \mathsf{X}_1. \cdots \lambda \mathsf{X}_n.\mathsf{E})$; $(\exists \hat{\mathsf{V}}\ \mathsf{E})$ to denote $(\exists \mathsf{V}_1 \cdots \exists \mathsf{V}_n\ \mathsf{E})$. When $\hat{\mathsf{A}}$ or $\hat{\mathsf{X}}$ or $\hat{\mathsf{V}}$ is empty, then the corresponding expression is considered identical to $\mathsf{E}$.

A body expression of the form $(\mathsf{E}_1 \approx \mathsf{E}_2)$ will be called an *equality* while an expression of the form $\sim\exists \hat{\mathsf{V}}(\mathsf{E}_1 \approx \mathsf{E}_2)$ an *inequality*; in the latter case $\hat{\mathsf{V}}$ may be empty (in which case the inequality is of the form $\sim(\mathsf{E}_1 \approx \mathsf{E}_2)$). If $\hat{\mathsf{E}} = \langle \mathsf{E}_1, \ldots, \mathsf{E}_n \rangle$ and $\hat{\mathsf{E}}' = \langle \mathsf{E}'_1, \ldots, \mathsf{E}'_n \rangle$, where all $\mathsf{E}_i, \mathsf{E}'_i$ are of type $\iota$, we will write $(\hat{\mathsf{E}} \approx \hat{\mathsf{E}}')$ to denote the expression $(\mathsf{E}_1 \approx \mathsf{E}'_1) \wedge \cdots \wedge (\mathsf{E}_n \approx \mathsf{E}'_n)$; if $n = 0$, then the conjunction is the constant $\mathsf{true}$.

**Definition 4.** The set of *clausal expressions* of the higher-order language $\mathcal{H}_{cn}$ is defined as follows:

1. If $\mathsf{p}$ is a predicate constant of type $\pi$ and $\mathsf{E}$ is a closed body expression of type $\pi$ then $\mathsf{p} \leftarrow_\pi \mathsf{E}$ is a clausal expression of $\mathcal{H}_{cn}$, also called a *program clause*.
2. If $\mathsf{E}$ is a body expression of type $o$ and each free variable in $\mathsf{E}$ is of type $\mu$ then $\mathsf{false} \leftarrow_o \mathsf{E}$ (usually denoted by $\leftarrow_o \mathsf{E}$ or just $\leftarrow \mathsf{E}$) is a clausal expression of $\mathcal{H}_{cn}$, also called a *goal clause*.
3. If $\mathsf{p}$ is a predicate constant of type $\pi$ and $\mathsf{E}$ is a closed body expression of type $\pi$ then $\mathsf{p} \leftrightarrow_\pi \mathsf{E}$ is a clausal expression of $\mathcal{H}_{cn}$, also called a *completion expression*.

All clausal expressions of $\mathcal{H}_{cn}$ have type $o$.

Notice that (following the tradition of first-order logic programming) we will talk about the "empty clause" which is denoted by $\Box$ and is equivalent to $\leftarrow \mathsf{true}$.

**Definition 5.** A program of $\mathcal{H}_{cn}$ is a finite set of program clauses of $\mathcal{H}_{cn}$.

We proceed by defining the completion of a program:

**Definition 6.** Let $\mathsf{P}$ be a program and let $\mathsf{p}$ be a predicate constant of type $\pi$. Then, the *completed definition* for $\mathsf{p}$ with respect to $\mathsf{P}$ is obtained as follows:

- if there exist exactly $k > 0$ program clauses of the form $\mathsf{p} \leftarrow_\pi \mathsf{E}_i$ where $i \in \{1, \ldots, k\}$ for $\mathsf{p}$ in $\mathsf{P}$, then the completed definition for $\mathsf{p}$ is the expression $\mathsf{p} \leftrightarrow_\pi \mathsf{E}$, where $\mathsf{E} = \mathsf{E}_1 \bigvee_\pi \cdots \bigvee_\pi \mathsf{E}_k$.
- if there are no program clauses for $\mathsf{p}$ in $\mathsf{P}$ then the completed definition for $\mathsf{p}$ is the expression $\mathsf{p} \leftrightarrow_\pi \mathsf{E}$, where $\mathsf{E}$ is of type $\pi$ and $\mathsf{E} = \lambda \hat{\mathsf{X}}.\mathsf{false}$.

The expression $\mathsf{E}$ on the right-hand side of the completed definition of $\mathsf{p}$ will be called the *completed expression* for $\mathsf{p}$ with respect to $\mathsf{P}$.

We can now extend the notion of completion (Clark 1977; Lloyd 1987) to apply to our higher-order programs:

**Definition 7.** Let $\mathsf{P}$ be a program. Then, the *completion* $\mathit{comp}(\mathsf{P})$ of $\mathsf{P}$ is the set consisting of all the completed definitions for all predicate constants that appear in $\mathsf{P}$.

## The Semantics of $\mathcal{H}_{cn}$

In this section we introduce the semantics of $\mathcal{H}_{cn}$. We start by defining the meaning of types of $\mathcal{H}_{cn}$; for the predicate types we also define a corresponding partial order (which will be used in the definition of the semantics of $\mathcal{H}_{cn}$).

**Definition 8.** Let $D$ be a non-empty set. Then the meanings of the types of our language can be specified as follows:

- $[\![\iota]\!]_D = D$.
- $[\![\iota^n \to \iota]\!]_D = D^n \to D$.
- $[\![o]\!]_D = \{\mathit{false}, \mathit{true}\}$. A partial order $\sqsubseteq_o$ can be defined on the set $[\![o]\!]_D$ as follows: $\mathit{false} \sqsubseteq_o \mathit{true}$, $\mathit{false} \sqsubseteq_o \mathit{false}$ and $\mathit{true} \sqsubseteq_o \mathit{true}$.
- $[\![\rho \to \pi]\!]_D = [\![\rho]\!]_D \to [\![\pi]\!]_D$. A partial order $\sqsubseteq_{\rho\to\pi}$ can be defined as follows: for all $f, g \in [\![\rho \to \pi]\!]_D$, $f \sqsubseteq_{\rho\to\pi} g$ if and only if $f(d) \sqsubseteq_\pi g(d)$, for all $d \in [\![\rho]\!]_D$.

It can be easily shown that for every predicate type $\pi$, $[\![\pi]\!]_D$ is a complete lattice, and therefore for every subset of $[\![\pi]\!]_D$ there exists a *lub* function (denoted by $\bigsqcup_\pi$) and a *glb* function (denoted by $\bigsqcap_\pi$).

The notions of *interpretation* and *state* can be defined in a standard way:

**Definition 9.** An interpretation $I$ of $\mathcal{H}_{cn}$ consists of:

1. a nonempty set $D$, called the *domain* of $I$
2. an assignment to each individual constant symbol $\mathsf{c}$, of an element $I(\mathsf{c}) \in D$
3. an assignment to each predicate constant $\mathsf{p}$ of type $\pi$, of an element $I(\mathsf{p}) \in [\![\pi]\!]_D$
4. an assignment to each function symbol $\mathsf{f}$ of type $\iota^n \to \iota$, of a function $I(\mathsf{f}) \in D^n \to D$.

**Definition 10.** Let $D$ be a nonempty set. Then, a *state $s$ of $\mathcal{H}_{cn}$ over $D$* is a function that assigns to each argument variable $\mathsf{V}$ of type $\rho$ of $\mathcal{H}_{cn}$ an element $s(\mathsf{V}) \in [\![\rho]\!]_D$.

In the following, $s[\mathsf{V}/d]$ is used to denote a state that is identical to $s$ the only difference being that the new state assigns to $\mathsf{V}$ the value $d$.

**Definition 11.** Let $I$ be an interpretation of $\mathcal{H}_{cn}$, let $D$ be the domain of $I$ and let $s$ be a state over $D$. Then, the semantics of expressions of $\mathcal{H}_{cn}$ with respect to $I$ and $s$ is defined as follows:

1. $[\![\mathsf{false}]\!]_s(I) = \mathit{false}$
2. $[\![\mathsf{true}]\!]_s(I) = \mathit{true}$
3. $[\![\mathsf{c}]\!]_s(I) = I(\mathsf{c})$, for every individual constant $\mathsf{c}$
4. $[\![\mathsf{p}]\!]_s(I) = I(\mathsf{p})$, for every predicate constant $\mathsf{p}$
5. $[\![\mathsf{V}]\!]_s(I) = s(\mathsf{V})$, for every argument variable $\mathsf{V}$
6. $[\![(\mathsf{f}\ \mathsf{E}_1 \cdots \mathsf{E}_n)]\!]_s(I) = I(\mathsf{f})\ [\![\mathsf{E}_1]\!]_s(I) \cdots [\![\mathsf{E}_n]\!]_s(I)$, for every $n$-ary function symbol $\mathsf{f}$
7. $[\![(\mathsf{E}_1\mathsf{E}_2)]\!]_s(I) = [\![\mathsf{E}_1]\!]_s(I)([\![\mathsf{E}_2]\!]_s(I))$
8. $[\![(\lambda\mathsf{V}.\mathsf{E})]\!]_s(I) = \lambda d.[\![\mathsf{E}]\!]_{s[\mathsf{V}/d]}(I)$, where $d$ ranges over $[\![\mathit{type}(\mathsf{V})]\!]_D$
9. $[\![(\mathsf{E}_1 \bigvee_\pi \mathsf{E}_2)]\!]_s(I) = \bigsqcup_\pi\{[\![\mathsf{E}_1]\!]_s(I), [\![\mathsf{E}_2]\!]_s(I)\}$, where $\bigsqcup_\pi$ is the least upper bound function on $[\![\pi]\!]_D$
10. $[\![(\mathsf{E}_1 \bigwedge_\pi \mathsf{E}_2)]\!]_s(I) = \bigsqcap_\pi\{[\![\mathsf{E}_1]\!]_s(I), [\![\mathsf{E}_2]\!]_s(I)\}$, where $\bigsqcap_\pi$ is the greatest lower bound function on $[\![\pi]\!]_D$
11. $[\![(\mathsf{E}_1 \approx \mathsf{E}_2)]\!]_s(I) = \begin{cases} \mathit{true}, & \text{if } [\![\mathsf{E}_1]\!]_s(I) = [\![\mathsf{E}_2]\!]_s(I) \\ \mathit{false}, & \text{otherwise} \end{cases}$
12. $[\![(\exists\mathsf{V}\ \mathsf{E})]\!]_s(I) = \begin{cases} \mathit{true}, & \text{if there exists } d \in [\![\mathit{type}(\mathsf{V})]\!]_D \\ & \text{such that } [\![\mathsf{E}]\!]_{s[\mathsf{V}/d]}(I) = \mathit{true} \\ \mathit{false}, & \text{otherwise} \end{cases}$
13. $[\![(\sim\!\mathsf{E})]\!]_s(I) = \begin{cases} \mathit{true}, & \text{if } [\![\mathsf{E}]\!]_s(I) = \mathit{false} \\ \mathit{false}, & \text{if } [\![\mathsf{E}]\!]_s(I) = \mathit{true} \end{cases}$
14. $[\![\mathsf{p} \leftarrow_\pi \mathsf{E}]\!]_s(I) = \begin{cases} \mathit{true}, & \text{if } [\![\mathsf{E}]\!]_s(I) \sqsubseteq_\pi I(\mathsf{p}) \\ \mathit{false}, & \text{otherwise} \end{cases}$
15. $[\![\mathsf{p} \leftrightarrow_\pi \mathsf{E}]\!]_s(I) = \begin{cases} \mathit{true}, & \text{if } [\![\mathsf{E}]\!]_s(I) = I(\mathsf{p}) \\ \mathit{false}, & \text{otherwise} \end{cases}$
16. $[\![\leftarrow \mathsf{E}]\!]_s(I) = \begin{cases} \mathit{true}, & \text{if } [\![\mathsf{E}]\!]_s(I) = \mathit{false} \\ \mathit{false}, & \text{otherwise} \end{cases}$

**Definition 12.** Let $S$ be a set of expressions of type $o$ and let $I$ be an interpretation of $\mathcal{H}_{cn}$. We say that $I$ is a *model* of $S$ if for every $\mathsf{E} \in S$ and for every state $s$ over the domain of $I$, $[\![\mathsf{E}]\!]_s(I) = \mathit{true}$.

Clark's Equality Theory can also be added to our fragment as follows:

**Definition 13.** Let $I$ be an interpretation of $\mathcal{H}_{cn}$. We will say that $I$ satisfies *Clark's Equality Theory (CET)* if for all states $s$ over the domain of $I$, all the following hold:

1. $[\![(\mathsf{f}\ \hat{\mathsf{E}}) \approx (\mathsf{f}\ \hat{\mathsf{E}}')]\!]_s(I) = [\![\hat{\mathsf{E}} \approx \hat{\mathsf{E}}']\!]_s(I)$ for every $n$-ary function symbol $\mathsf{f}$ and all $\hat{\mathsf{E}} = \langle \mathsf{E}_1, \ldots, \mathsf{E}_n\rangle$ and $\hat{\mathsf{E}}' = \langle \mathsf{E}'_1, \ldots, \mathsf{E}'_n\rangle$
2. $[\![\mathsf{c} \approx \mathsf{d}]\!]_s(I) = \mathit{false}$ for all different individual constants $\mathsf{c}$ and $\mathsf{d}$
3. $[\![(\mathsf{f}\ \hat{\mathsf{E}}) \approx \mathsf{c}]\!]_s(I) = \mathit{false}$ for every function symbol $\mathsf{f}$ and every individual constant $\mathsf{c}$
4. $[\![(\mathsf{f}\ \hat{\mathsf{E}}) \approx (\mathsf{g}\ \hat{\mathsf{E}}')]\!]_s(I) = \mathit{false}$ for all different function symbols $\mathsf{f}$ and $\mathsf{g}$ and all $\hat{\mathsf{E}}$ and $\hat{\mathsf{E}}'$
5. $[\![\mathsf{X} \approx \mathsf{E}]\!]_s(I) = \mathit{false}$ if $\mathsf{X} \in \mathit{fv}(\mathsf{E})$ and $\mathsf{X} \neq \mathsf{E}$

In the rest of this paper we will often talk about "models of the completion $\mathit{comp}(\mathsf{P})$ of a program $\mathsf{P}$"; in every such case we will implicitly assume that our models also satisfy Clark's Equality Theory.

## 4 Proof Procedure

In this section we define a proof procedure for $\mathcal{H}_{cn}$ and discuss its implementation. As in the case of classical logic programming, substitutions and unifiers also play an important role in our setting. However, they need to be significantly extended, as discussed below.

### Substitutions and Unifiers

In classical logic programming, a substitution is a mapping from variables to terms. Since in our case queries may contain uninstantiated variables of set types, the notion of substitution must be extended to assign to these variables expressions that represent sets. In the following we define the notion of *basic expression* which generalizes the notion of term from classical logic programming. Intuitively, a basic element is either a term or a set of terms.

**Definition 14.** The set of *basic expressions* of $\mathcal{H}_{cn}$ of type $\mu$ is defined recursively as follows:

1. Every expression of $\mathcal{H}_{cn}$ of type $\iota$ is a basic expression of type $\iota$.
2. Every predicate variable of type $\kappa$ is a basic expression of type $\kappa$.
3. If $\mathsf{E}_1, \mathsf{E}_2$ are basic expressions of type $\kappa$ then $\mathsf{E}_1 \bigvee_\kappa \mathsf{E}_2$ and $\mathsf{E}_1 \bigwedge_\kappa \mathsf{E}_2$ are basic expressions of type $\kappa$.
4. The expressions of the following form, are basic expressions of type $\iota^n \to o$:
   - $\lambda\hat{\mathsf{X}}.\exists\hat{\mathsf{V}}(\hat{\mathsf{X}} \approx \hat{\mathsf{A}})$
   - $\lambda\hat{\mathsf{X}}.\sim\exists\hat{\mathsf{V}}(\hat{\mathsf{X}} \approx \hat{\mathsf{A}})$

   where $\hat{\mathsf{X}} = \langle \mathsf{X}_1, \ldots, \mathsf{X}_n\rangle$, $\hat{\mathsf{A}} = \langle \mathsf{A}_1, \ldots, \mathsf{A}_n\rangle$, each $\mathsf{X}_i$ is a variable of type $\iota$, each $\hat{\mathsf{A}}_i$ is a basic expressions of type $\iota$ and $\hat{\mathsf{V}}$ is a possibly empty subset of $fv(\hat{\mathsf{A}})$.

**Example 8.** The basic element $\lambda\mathtt{X}.\sim\exists\mathtt{Y}(\mathtt{X} \approx \mathtt{Y})$ represents the empty set while $\lambda\mathtt{X}.(\mathtt{X} \approx \mathtt{a}) \bigvee \lambda\mathtt{X}.(\mathtt{X} \approx \mathtt{f(b)})$ represents the set $\{\mathtt{a}, \mathtt{f(b)}\}$. The basic element $\lambda\mathtt{X}.\exists\mathtt{Y}(\mathtt{X} \approx \mathtt{f(Y)})$ is the set that contains all the terms of the form $\mathtt{f}(\cdots)$. The basic element $\lambda\mathtt{X}.\sim\exists\mathtt{Y}(\mathtt{X} \approx \mathtt{f(Y)})$ is the set that contains all the terms that are not of the form $\mathtt{f}(\cdots)$. The basic element $\lambda\mathtt{X}.\exists\mathtt{Y}(\mathtt{X} \approx \mathtt{f(Y)}) \bigwedge \lambda\mathtt{X}.\sim(\mathtt{X} \approx \mathtt{f(b)})$ is the set that contains all the terms of the form $\mathtt{f}(\cdots)$ with the exception of $\mathtt{f(b)}$.

The definition of a substitution can be extended:

**Definition 15.** A substitution $\theta$ is a finite set of the form $\{\mathsf{V}_1/\mathsf{E}_1, \ldots, \mathsf{V}_n/\mathsf{E}_n\}$ where $\mathsf{V}_i$ are different argument variables of $\mathcal{H}_{cn}$ and each $\mathsf{E}_i$ is a body expression of $\mathcal{H}_{cn}$ having the same type as $\mathsf{V}_i$. We write $dom(\theta) = \{\mathsf{V}_1, \ldots, \mathsf{V}_n\}$ and $range(\theta) = \{\mathsf{E}_1, \ldots, \mathsf{E}_n\}$. A substitution is called *basic* if all $\mathsf{E}_i$ are basic expressions. A substitution is called *zero-order* if the type of $\mathsf{V}_i$ is $\iota$ for all $i \in \{1, \ldots, n\}$. The substitution corresponding to the empty set will be called the *identity substitution* and will be denoted by $\epsilon$.

In the following definition (and also in the rest of the paper) we assume that if expressions $\mathsf{E}_1, \ldots, \mathsf{E}_n$ occur in a certain mathematical context, then in these expressions all bound variables are chosen to be different from the free variables. This is called the *the bound variable convention* in (Barendregt 1984, pages 26–27) (and is actually equivalent to the $\alpha$-renaming operation of $\lambda$-calculus).

**Definition 16.** Let $\theta$ be a substitution and let $\mathsf{E}$ be a body expression. Then, $\mathsf{E}\theta$ is an expression obtained as follows:

- $\mathsf{E}\theta = \mathsf{E}$, if $\mathsf{E}$ is false, true, c, or p
- $\mathsf{V}\theta = \theta(\mathsf{V})$ if $\mathsf{V} \in dom(\theta)$; otherwise, $\mathsf{V}\theta = \mathsf{V}$
- $(\mathsf{f}\ \mathsf{E}_1 \cdots \mathsf{E}_n)\theta = (\mathsf{f}\ \mathsf{E}_1\theta \cdots \mathsf{E}_n\theta)$
- $(\mathsf{E}_1\ \mathsf{E}_2)\theta = (\mathsf{E}_1\theta\ \mathsf{E}_2\theta)$
- $(\lambda\mathsf{V}.\mathsf{E})\theta = (\lambda\mathsf{V}.\mathsf{E}\theta)$
- $(\mathsf{E}_1 \bigvee_\pi \mathsf{E}_2)\theta = (\mathsf{E}_1\theta \bigvee_\pi \mathsf{E}_2\theta)$
- $(\mathsf{E}_1 \bigwedge_\pi \mathsf{E}_2)\theta = (\mathsf{E}_1\theta \bigwedge_\pi \mathsf{E}_2\theta)$
- $(\mathsf{E}_1 \approx \mathsf{E}_2)\theta = (\mathsf{E}_1\theta \approx \mathsf{E}_2\theta)$
- $(\exists\mathsf{V}\ \mathsf{E})\theta = (\exists\mathsf{V}\ \mathsf{E}\theta)$
- $(\sim\mathsf{E})\theta = \sim\mathsf{E}\theta$

In the following, if $\hat{\mathsf{E}} = \langle \mathsf{E}_1, \ldots, \mathsf{E}_n\rangle$, we will write $\hat{\mathsf{E}}\theta$ to denote the sequence $\langle \mathsf{E}_1\theta, \ldots, \mathsf{E}_n\theta\rangle$.

**Definition 17.** Let $\theta = \{\mathsf{V}_1/\mathsf{E}_1, \ldots, \mathsf{V}_m/\mathsf{E}_m\}$ and $\sigma = \{\mathsf{V}'_1/\mathsf{E}'_1, \ldots, \mathsf{V}'_m/\mathsf{E}'_m\}$ be substitutions. Then the composition $\theta\sigma$ of $\theta$ and $\sigma$ is the substitution obtained from the set $\{\mathsf{V}_1/\mathsf{E}_1\sigma, \ldots, \mathsf{V}_m/\mathsf{E}_m\sigma, \mathsf{V}'_1/\mathsf{E}'_1, \ldots, \mathsf{V}_n/\mathsf{E}'_n\}$ by deleting any $\mathsf{V}_i/\mathsf{E}_i\sigma$ for which $\mathsf{V}_i = \mathsf{E}_i\sigma$ and deleting any $\mathsf{V}'_j/\mathsf{E}'_j$ for which $\mathsf{V}'_j \in dom(\theta)$.

The notions of "unifier" and "most general unifier" are the same as in classical first-order logic programming.

**Definition 18.** Let $S$ be a set of terms of $\mathcal{H}_{cn}$ (ie., expressions of type $\iota$). A zero-order substitution $\theta$ will be called a *unifier* of the expressions in $S$ if the set $S\theta = \{\mathsf{E}\theta \mid \mathsf{E} \in S\}$ is a singleton. The zero-order substitution $\theta$ will be called a *most general unifier of* $S$ (denoted by $mgu(S)$), if for every unifier $\sigma$ of the expressions in $S$, there exists a zero-order substitution $\gamma$ such that $\sigma = \theta\gamma$.

The following *substitution lemma* can easily be established by structural induction on $\mathsf{E}$:

**Lemma 1** (Substitution Lemma)**.** *Let $I$ be an interpretation of $\mathcal{H}_{cn}$ and let $s$ be a state over the domain of $I$. Let $\theta$ be a basic substitution and $\mathsf{E}$ be a body expression. Then, $[\![\mathsf{E}\theta]\!]_s(I) = [\![\mathsf{E}]\!]_{s'}(I)$, where $s'(\mathsf{V}) = [\![\theta(\mathsf{V})]\!]_s(I)$ if $\mathsf{V} \in dom(\theta)$ and $s'(\mathsf{V}) = s(\mathsf{V})$, otherwise.*

The above lemma will be used in the proof of the soundness of the resolution proof procedure that follows.

## The Procedure

In this subsection we describe the actual proof procedure for $\mathcal{H}_{cn}$. Intuitively, we could say that our approach generalizes to the higher-order case the constructive negation procedure proposed by D. Chan in (Chan 1989). In order to define the overall proof procedure we utilize three definitions, described as follows:

- The central definition of our procedure is Definition 20 which defines the *single-step derivation* of goals. Intuitively, the single-step derivation is an extension of the single-step derivation of classical logic programming (see for example (Lloyd 1987, pages 50–41)) in order to apply to higher-order logic programming and to constructive negation. Notice that the single-step derivation is the only one among the three that may produce substitutions (whose final composition will lead to the computed answer for the initial goal).
- The *reduction* (Definition 21) is the simplest among the three definitions. Roughly speaking it corresponds to certain simple computational steps that advance the evaluation of the goal. For example, the $\beta$-reduction step for a $\lambda$-expression belongs to this reduction relation. Another simple reduction step is the replacement of a predicate constant with its defining expression in the program.
- The *negative reduction* (Definition 22) is the most complicated among the three. Intuitively, it corresponds to what D. Chan in (Chan 1989) refers to as "the negation of

answers" and the "normalization of answers". The definition is more complicated in our case since it applies to the more demanding setting of higher-order logic programming.

One difference between the proof procedures for constructive negation (such as for example the one given in (Chan 1989)) and the traditional SLD-resolution for negationless logic programming, is the following: in classical SLD-resolution, a derivation is successful if we reach the empty goal; however, in constructive negation, a derivation may be successful if we reach a goal that consists of inequalities that can not be further simplified. Therefore, the termination condition for constructive negation is different than that of SLD-resolution. Moreover, an *answer* in classical resolution is simply a substitution for some of the variables of the goal. In constructive negation, an answer is a pair consisting of a substitution and of a conjunction of inequalities (intuitively, the inequalities that remain at the end of the derivation)[1]. In the higher-order case that we are examining, an answer is also a pair, the only difference being that the domain of the substitution may contain higher-order variables.

Before we present the actual definitions, we need the following categorization of inequalities, which is pretty standard in the area of constructive negation (see for example (Chan 1989)):

**Definition 19.** An inequality $\sim\exists\hat{\mathsf{V}}(\mathsf{E}_1 \approx \mathsf{E}_2)$ is considered

- *valid* if $\mathsf{E}_1$ and $\mathsf{E}_2$ cannot be unified
- *unsatisfiable* if there is a substitution $\theta$ that unifies $\mathsf{E}_1$ and $\mathsf{E}_2$ and contains only bindings of variables in $\hat{\mathsf{V}}$.
- *satisfiable* if it is not unsatisfiable.

An inequality will be called *primitive* if it is satisfiable, non valid and either $\mathsf{E}_1$ or $\mathsf{E}_2$ is a variable.

In the following definition we will refer to a conjunction $\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n$ of expressions. We will assume that each $\mathsf{A}_i$ is not a conjunction itself. The conjunction of a single expression is the expression itself and the conjunction of zero expressions is the expression $\mathsf{true}$.

**Definition 20.** Let $\mathsf{P}$ a program and let $\mathsf{G}_k$ and $\mathsf{G}_{k+1}$ be goal clauses. Moreover, let $\mathsf{G}_k$ be a conjunction $\leftarrow \mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n$, where each $\mathsf{A}_i$ is a body expression of type $o$. Let $\mathsf{A}_i$ be one of the $\mathsf{A}_1, \ldots, \mathsf{A}_n$ and let us call it the *selected expression*. Let $\mathsf{A}' = \mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_{i-1} \wedge \mathsf{A}_{i+1} \wedge \cdots \wedge \mathsf{A}_n$. We say that $\mathsf{G}_{k+1}$ is *derived in one step* from $\mathsf{G}_k$ using $\theta$ (denoted as $\mathsf{G}_k \xrightarrow{\theta} \mathsf{G}_{k+1}$) if one of the following conditions applies:

1. if $\mathsf{A}_i$ is $\mathsf{true}$ and $n > 1$ then $\mathsf{G}_{k+1} = \leftarrow \mathsf{A}'$ is derived from $\mathsf{G}_k$ using $\theta = \epsilon$.
2. if $\mathsf{A}_i$ is $(\mathsf{E}_1 \vee \mathsf{E}_2)$ then $\mathsf{G}_{k+1} = \leftarrow \mathsf{A}_1 \wedge \cdots \wedge \mathsf{E}_j \wedge \cdots \wedge \mathsf{A}_n$ is derived from $\mathsf{G}_k$ using $\theta = \epsilon$ where $j \in \{1, 2\}$.
3. if $\mathsf{A}_i$ is $(\exists \mathsf{V}\ \mathsf{E})$ then $\mathsf{G}_{k+1} = \leftarrow \mathsf{A}_1 \wedge \cdots \wedge \mathsf{E} \wedge \cdots \wedge \mathsf{A}_n$ is derived[2] from $\mathsf{G}_k$ using $\theta = \epsilon$.
4. if $\mathsf{A}_i \rightsquigarrow \mathsf{A}'_i$ then $\mathsf{G}_{k+1} = \leftarrow \mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_{i-1} \wedge \mathsf{A}'_i \wedge \mathsf{A}_{i+1} \wedge \cdots \wedge \mathsf{A}_n$ is derived from $\mathsf{G}_k$ using $\theta = \epsilon$.
5. if $\mathsf{A}_i$ is $(\mathsf{E}_1 \approx \mathsf{E}_2)$ then $\mathsf{G}_{k+1} = \leftarrow \mathsf{A}'\theta$ is derived from $\mathsf{G}_k$ using $\theta = mgu(\mathsf{E}_1, \mathsf{E}_2)$.
6. if $\mathsf{A}_i$ is $(\mathsf{R}\ \hat{\mathsf{E}})$ and $\mathsf{R}$ is a predicate variable of type $\kappa$ then $\mathsf{G}_{k+1} = \leftarrow \mathsf{A}'\theta$ is derived from $\mathsf{G}_k$ using $\theta = \{\mathsf{R}/(\lambda\hat{\mathsf{X}}.(\hat{\mathsf{X}} \approx \hat{\mathsf{E}}) \bigvee_\kappa \mathsf{R}')\}$ where $\mathsf{R}'$ is a fresh[3] predicate variable of type $\kappa$.
7. if $\mathsf{A}_i$ is $\sim\exists\hat{\mathsf{V}}\ \mathsf{E}$ and $\mathsf{A}_i$ is negatively-reduced to $\mathsf{A}'_i$ then $\mathsf{G}_{k+1} = \leftarrow \mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_{i-1} \wedge \mathsf{A}'_i \wedge \mathsf{A}_{i+1} \wedge \cdots \wedge \mathsf{A}_n$ is derived from $\mathsf{G}_k$ using $\theta = \epsilon$.
8. if $\mathsf{A}_i$ is $\sim\exists\hat{\mathsf{V}}(\mathsf{R}\ \hat{\mathsf{E}})$ and $\mathsf{R}$ is a predicate variable of type $\kappa$ and $\mathsf{R} \notin \hat{\mathsf{V}}$ then $\mathsf{G}_{k+1} = \leftarrow \mathsf{A}'\theta$ is derived from $\mathsf{G}_k$ using $\theta = \{\mathsf{R}/(\lambda\hat{\mathsf{X}}.\ \sim\exists\hat{\mathsf{V}}(\hat{\mathsf{X}} \approx \hat{\mathsf{E}}) \bigwedge_\kappa \mathsf{R}')\}$ where $\mathsf{R}'$ is a fresh predicate variable of type $\kappa$.
9. if $\mathsf{A}_i$ is $\sim\exists\hat{\mathsf{V}} \sim(\mathsf{R}\ \hat{\mathsf{E}})$ and $\mathsf{R}$ is a predicate variable of type $\kappa$ and $\mathsf{R} \notin \hat{\mathsf{V}}$ then $\mathsf{G}_{k+1} = \leftarrow \mathsf{A}'\theta$ is derived from $\mathsf{G}_k$ using $\theta = \{\mathsf{R}/(\lambda\hat{\mathsf{X}}.\exists\hat{\mathsf{V}}(\hat{\mathsf{X}} \approx \hat{\mathsf{E}}) \bigvee_\kappa \mathsf{R}')\}$ where $\mathsf{R}'$ is a fresh predicate variable of type $\kappa$.

**Definition 21.** Let $\mathsf{P}$ be a program and $\mathsf{E}, \mathsf{E}'$ be body expressions of type $o$. We say that $\mathsf{E}$ is reduced (wrt. to $\mathsf{P}$) to $\mathsf{E}'$ (denoted as $\mathsf{E} \rightsquigarrow \mathsf{E}'$) if one of the following conditions applies:

1. $\mathsf{p}\ \hat{\mathsf{A}} \rightsquigarrow \mathsf{E}\ \hat{\mathsf{A}}$, where $\mathsf{E}$ is the completed expression for $\mathsf{p}$ with respect to $\mathsf{P}$.
2. $(\lambda\mathsf{X}.\mathsf{E})\ \mathsf{B}\ \hat{\mathsf{A}} \rightsquigarrow \mathsf{E}\{\mathsf{X}/\mathsf{B}\}\ \hat{\mathsf{A}}$
3. $(\mathsf{E}_1 \bigvee_\pi \mathsf{E}_2)\ \hat{\mathsf{A}} \rightsquigarrow (\mathsf{E}_1\ \hat{\mathsf{A}}) \vee_o (\mathsf{E}_2\ \hat{\mathsf{A}})$
4. $(\mathsf{E}_1 \bigwedge_\pi \mathsf{E}_2)\ \hat{\mathsf{A}} \rightsquigarrow (\mathsf{E}_1\ \hat{\mathsf{A}}) \wedge_o (\mathsf{E}_2\ \hat{\mathsf{A}})$

**Definition 22.** Let $\mathsf{P}$ be a program and $\mathsf{B}, \mathsf{B}'$ be body expressions where $\mathsf{B} = \sim\exists\hat{\mathsf{U}}(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n)$ and each $\mathsf{A}_i$ is a body expression except from conjunction. Let $\mathsf{A}_i$ be the selected expression and $\mathsf{A}' = \mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_{i-1} \wedge \mathsf{A}_{i+1} \wedge \cdots \wedge \mathsf{A}_n$. Then, we say that $\mathsf{B}$ is *negatively-reduced* to $\mathsf{B}'$ if one of the following conditions applies:

1. if $\mathsf{A}_i$ is $\mathsf{false}$ then $\mathsf{B}' = \mathsf{true}$
2. if $\mathsf{A}_i$ is $\mathsf{true}$ and $n = 1$ then $\mathsf{B}' = \mathsf{false}$ else $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}\ \mathsf{A}'$
3. if $\mathsf{A}_i$ is $(\mathsf{E}_1 \vee \mathsf{E}_2)$ then $\mathsf{B}' = \mathsf{B}'_1 \wedge \mathsf{B}'_2$ where $\mathsf{B}'_j = \sim\exists\hat{\mathsf{U}}(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{E}_j \wedge \cdots \wedge \mathsf{A}_n), j \in \{1, 2\}$
4. if $\mathsf{A}_i$ is $(\exists \mathsf{V}\ \mathsf{E})$ then $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}\mathsf{V}(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{E} \wedge \cdots \wedge \mathsf{A}_n)$[4]
5. if $\mathsf{A}_i \rightsquigarrow \mathsf{A}'_i$ then $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}'_i \wedge \cdots \wedge \mathsf{A}_n)$
6. if $\mathsf{A}_i$ is $(\mathsf{E}_1 \approx \mathsf{E}_2)$ then
   (a) if $\sim\exists\hat{\mathsf{U}}(\mathsf{E}_1 \approx \mathsf{E}_2)$ is valid then $\mathsf{B}' = \mathsf{true}$
   (b) if $\sim\exists\hat{\mathsf{U}}(\mathsf{E}_1 \approx \mathsf{E}_2)$ is non-valid and neither $\mathsf{E}_1$ nor $\mathsf{E}_2$ is a variable then $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_{i-1} \wedge (\hat{\mathsf{X}} \approx \hat{\mathsf{X}}\theta) \wedge \mathsf{A}_{i+1} \wedge \cdots \wedge \mathsf{A}_n)$ where $\theta = mgu(\mathsf{E}_1, \mathsf{E}_2)$ and $\hat{\mathsf{X}} = dom(\theta)$.

[1]Actually, in (Chan 1989) the substitution is combined with the inequalities so as to obtain a sequence consisting of equalities and inequalities.

[2]Recall that by the Bound Variable Convention, $\mathsf{V}$ does not appear free in $\mathsf{A}'$.

[3]This means a variable that has not appeared in goals and substitutions until this point of the derivation.

[4]Recall that by the Bound Variable Convention, $\mathsf{V}$ does not appear free in $\mathsf{A}'$.

(c) if $\sim\exists\hat{\mathsf{U}}(\mathsf{E}_1 \approx \mathsf{E}_2)$ is unsatisfiable and either $\mathsf{E}_1$ or $\mathsf{E}_2$ is a variable in $\hat{\mathsf{U}}$, then $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}(\mathsf{A}'\theta)$ where $\theta = \{\mathsf{X}/\mathsf{E}\}$ and $\mathsf{X}$ is the one expression that is a variable in $\hat{\mathsf{U}}$ and $\mathsf{E}$ is the other.

(d) if $\sim \exists\hat{\mathsf{U}}(\mathsf{E}_1 \approx \mathsf{E}_2)$ is primitive and $n > 1$ then $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}_1\ \mathsf{A}_i \vee \exists\hat{\mathsf{U}}_1(\mathsf{A}_i \wedge \sim\exists\hat{\mathsf{U}}_2\ \mathsf{A}')$ where $\hat{\mathsf{U}}_1$ are the variables in $\hat{\mathsf{U}}$ that are free in $\mathsf{A}_i$ and $\hat{\mathsf{U}}_2$ the variables in $\hat{\mathsf{U}}$ not in $\hat{\mathsf{U}}_1$.

7. if $\mathsf{A}_i$ is $(\mathsf{R}\ \hat{\mathsf{E}})$ and $\mathsf{R}$ is a predicate variable then
   (a) if $\mathsf{R} \in \hat{\mathsf{U}}$ then $\mathsf{B}' = \sim \exists\hat{\mathsf{U}}'(\mathsf{A}'\theta)$ where $\theta = \{\mathsf{R}/(\lambda\mathsf{X}.(\mathsf{X} \approx \mathsf{E}) \bigvee_\kappa \mathsf{R}')\}$, $\mathsf{R}'$ is a predicate variable of the same type with $\mathsf{R}$ and $\hat{\mathsf{U}}'$ is the same with $\hat{\mathsf{U}}$ but the variable $\mathsf{R}$ has been replaced with $\mathsf{R}'$.
   (b) if $\mathsf{R} \notin \hat{\mathsf{U}}$ and $n > 1$ then $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}_1\ \mathsf{A}_i \vee \exists\hat{\mathsf{U}}_1(\mathsf{A}_i \wedge \sim \exists\hat{\mathsf{U}}_2\ \mathsf{A}') \wedge \mathsf{B}$ where $\hat{\mathsf{U}}_1$ are the variables in $\hat{\mathsf{U}}$ that are free in $\mathsf{A}_i$ and $\hat{\mathsf{U}}_2$ the variables in $\hat{\mathsf{U}}$ not in $\hat{\mathsf{U}}_1$.
8. if $\mathsf{A}_i$ is $\sim\exists\hat{\mathsf{V}}\ \mathsf{E}$ and $\mathsf{A}_i$ is negatively-reduced to $\mathsf{A}_i'$ then $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_i' \wedge \cdots \wedge \mathsf{A}_n)$
9. if $\mathsf{A}_i$ is a primitive inequality $\sim\exists\hat{\mathsf{V}}(\mathsf{E}_1 \approx \mathsf{E}_2)$ then
   (a) if $\mathsf{A}_i$ contains free variables in $\hat{\mathsf{U}}$ and $\mathsf{A}'$ is a conjunction of primitive inequalities then $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}\ \mathsf{A}'$
   (b) if $\mathsf{A}_i$ does not contain any free variables in $\hat{\mathsf{U}}$ then $\mathsf{B}' = \exists\hat{\mathsf{V}}(\mathsf{E}_1 \approx \mathsf{E}_2) \vee \sim\exists\hat{\mathsf{U}}\ \mathsf{A}'$
10. if $\mathsf{A}_i$ is $\sim\exists\hat{\mathsf{V}}(\mathsf{R}\ \hat{\mathsf{E}})$ and $\mathsf{R}$ is a predicate variable with $\mathsf{R} \notin \hat{\mathsf{V}}$, then.
   (a) if $\mathsf{R} \in \hat{\mathsf{U}}$ then $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}'(\mathsf{A}'\theta)$ where $\theta = \{\mathsf{R}/(\lambda\mathsf{X}. \sim \exists\hat{\mathsf{V}}(\mathsf{X} \approx \mathsf{E}) \bigwedge_\kappa \mathsf{R}')\}$, $\mathsf{R}'$ is a predicate variable of the same type $\kappa$ with $\mathsf{R}$ and $\hat{\mathsf{U}}'$ is the same with $\hat{\mathsf{U}}$ but the variable $\mathsf{R}$ has been replaced with $\mathsf{R}'$.
   (b) if $\mathsf{R} \notin \hat{\mathsf{U}}$ and $n > 1$ then $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}_1\ \mathsf{A}_i \vee \exists\hat{\mathsf{U}}_1(\mathsf{A}_i \wedge \sim \exists\hat{\mathsf{U}}_2\ \mathsf{A}') \wedge \mathsf{B}$ where $\hat{\mathsf{U}}_1$ are the variables in $\hat{\mathsf{U}}$ that are free in $\mathsf{A}_i$ and $\hat{\mathsf{U}}_2$ the variables in $\hat{\mathsf{U}}$ not in $\hat{\mathsf{U}}_1$.
   (c) if $\mathsf{R} \notin \hat{\mathsf{U}}$, $n = 1$ and $\hat{\mathsf{V}}$ is non-empty then $\mathsf{B}' = \exists\hat{\mathsf{V}} \sim \exists\hat{\mathsf{U}}(\sim(\mathsf{R}\ \hat{\mathsf{E}}) \wedge \sim\exists\hat{\mathsf{V}}'(\mathsf{R}\ \mathsf{E}'))$
11. if $\mathsf{A}_i$ is $\sim\exists\hat{\mathsf{V}}\ \sim(\mathsf{R}\ \hat{\mathsf{E}})$ and $\mathsf{R}$ is a predicate variable and $\mathsf{R} \notin \hat{\mathsf{V}}$, then
   (a) if $\mathsf{R} \in \hat{\mathsf{U}}$ then $\mathsf{B}' = \sim \exists\hat{\mathsf{U}}'(\mathsf{A}'\theta)$ where $\theta = \{\mathsf{R}/(\lambda\mathsf{X}.\exists\hat{\mathsf{V}}(\mathsf{X} \approx \mathsf{E}) \bigvee_\kappa \mathsf{R}')\}$, $\mathsf{R}'$ is a predicate variable of the same type $\kappa$ with $\mathsf{R}$ and $\hat{\mathsf{U}}'$ is the same with $\hat{\mathsf{U}}$ but the variable $\mathsf{R}$ has been replaced with $\mathsf{R}'$.
   (b) if $\mathsf{R} \notin \hat{\mathsf{U}}$ and $n > 1$ then $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}_1\ \mathsf{A}_i \vee \exists\hat{\mathsf{U}}_1(\mathsf{A}_i \wedge \sim \exists\hat{\mathsf{U}}_2\ \mathsf{A}') \wedge \mathsf{B}$
   (c) if $\mathsf{R} \notin \hat{\mathsf{U}}$, $n = 1$ and $\hat{\mathsf{V}}$ is non-empty then $\mathsf{B}' = \exists\hat{\mathsf{V}} \sim \exists\hat{\mathsf{U}}((\mathsf{R}\ \hat{\mathsf{E}}) \wedge \sim\exists\hat{\mathsf{V}}' \sim(\mathsf{R}\ \mathsf{E}'))$

One can easily check that all the three above definitions preserve the main property of $\mathcal{H}_{cn}$ goals, namely that all the free variables in them have existential types.

We proceed by defining the notion of a *multi-step derivation* and by characterizing when a multi-step derivation is successful and the form of the computed answer. We first define the notions of terminal and primitive goals:

**Definition 23.** Let $\mathsf{P}$ be a program and $\mathsf{G} = \leftarrow \mathsf{A}$ be a goal clause. Then, $\mathsf{G}$ will be called *terminal* if no goals can be derived in one step from $\mathsf{G}$. The goal $\mathsf{G}$ is called *primitive* if $\mathsf{A}$ is a (possibly empty) conjunction of primitive inequalities.

Notice that every primitive goal is also terminal. Notice also that the empty goal $\Box$ is primitive.

**Definition 24.** Let $\mathsf{G} = \leftarrow \mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n$ be a primitive goal and $S$ be a set of variables. We say that $\mathsf{G}_S$ is a primitive goal obtained by restricting $\mathsf{G}$ to $S$ when for each inequality $\mathsf{A}_i$ in $\mathsf{G}_S$ every free variable of $\mathsf{A}_i$ is in $S$.

**Definition 25.** Let $\mathsf{P}$ be a program and $\mathsf{G}$ be a goal. Assume that $\mathsf{P} \cup \{\mathsf{G}\}$ has a finite derivation $\mathsf{G}_0 = \mathsf{G}, \mathsf{G}_1, \ldots, \mathsf{G}_n$ with basic substitutions $\theta_1, \ldots, \theta_n$, such that the goal $\mathsf{G}_n$ is primitive. Then we will say that $\mathsf{P} \cup \{\mathsf{G}\}$ has a successful derivation of length $n$ with primitive goal $\mathsf{G}_n$ and using basic substitution $\theta = \theta_1 \cdots \theta_n$.

Notice that it can easily be verified that the composition of basic substitutions is indeed a basic substitution as implied by the above definition.

We can now define the important notion of *computed answer*:

**Definition 26.** Let $\mathsf{P}$ be a program, $\mathsf{G}$ be a goal and assume that $\mathsf{P} \cup \{\mathsf{G}\}$ has a successful derivation with primitive goal $\mathsf{G}'$ and basic substitution $\theta$. Then $(\sigma, \mathsf{G}'')$ is a *computed answer* for $\mathsf{P} \cup \{\mathsf{G}\}$ where $\sigma$ is the basic substitution obtained by restricting $\theta$ to the free variables of $\mathsf{G}$ and $\mathsf{G}''$ is the primitive goal $\mathsf{G}'$ restricted to the free variables of $\mathsf{G}$ and the variables in $fv(range(\sigma))$.

**Example 9.** Consider the following simple definition for the predicate $\mathtt{q}$

$$\mathtt{q} \leftarrow \lambda\mathtt{Z}_1.\lambda\mathtt{Z}_2.(\mathtt{Z}_1 \approx \mathtt{a}) \wedge (\mathtt{Z}_2 \approx \mathtt{b})$$

that holds only for the tuple $(\mathtt{a}, \mathtt{b})$. Then, consider the goal

$$\leftarrow (\mathtt{R}\ \mathtt{X}) \wedge \sim(\mathtt{q}\ \mathtt{X}\ \mathtt{Y})$$

that requests bindings for the variables $\mathtt{R}$, $\mathtt{X}$ and $\mathtt{Y}$.

We will underline the selected expression in each step when it is not obvious. We will also omit the substitution in a step if the rule used produces an identity substitution.

0. $\leftarrow \underline{(\mathtt{R}\ \mathtt{X})} \wedge \sim(\mathtt{q}\ \mathtt{X}\ \mathtt{Y})$
1. $\leftarrow\sim(\underline{\mathtt{q}\ \mathtt{X}\ \mathtt{Y}})$
   using rule 20.6 and $\theta_1 = \{\mathtt{R}/\lambda\mathtt{Z}.(\mathtt{Z} \approx \mathtt{X}) \bigvee \mathtt{R}'\}$
2. $\leftarrow\sim(\underline{(\lambda\mathtt{Z}_1.\lambda\mathtt{Z}_2.(\mathtt{Z}_1 \approx \mathtt{a}) \wedge (\mathtt{Z}_2 \approx \mathtt{b}))\ \mathtt{X}\ \mathtt{Y})}$
   using rules 20.7, 22.5 and then 21.1
3. $\leftarrow \underline{\sim((\mathtt{X} \approx \mathtt{a}) \wedge (\mathtt{Y} \approx \mathtt{b}))}$
   using rules 20.7, 22.5 and then 21.2
4. $\leftarrow\sim(\mathtt{X} \approx \mathtt{a}) \vee (\mathtt{X} \approx \mathtt{a}) \wedge \sim(\mathtt{Y} \approx \mathtt{a})$
   using rules 20.7 and then 22.6(d)
5. $\leftarrow\sim(\mathtt{X} \approx \mathtt{a})$
   using rule 20.2

In step 4 the procedure generates two branches. The first one terminates immediately with a primitive inequality. The computed answer is $\sim(\mathtt{X} \approx \mathtt{a})$ and $\{\mathtt{R}/\lambda\mathtt{Z}.(\mathtt{Z} \approx \mathtt{X}) \bigvee \mathtt{R}'\}$. The second branch will continue as follows:

5. $\leftarrow \underline{(\mathtt{X} \approx \mathtt{a})} \wedge \sim(\mathtt{Y} \approx \mathtt{a})$
   using rule 20.2
6. $\leftarrow \sim(\mathtt{Y} \approx \mathtt{a})$
   using rule 20.5 and $\theta_6 = \{\mathtt{X}/\mathtt{a}\}$

The procedure will now terminate successfully yielding the substitution $\{\mathtt{R}/\lambda\mathtt{Z}.(\mathtt{Z} \approx \mathtt{a}) \bigvee \mathtt{R}', \mathtt{X}/\mathtt{a}\}$ and the primitive goal $\sim(\mathtt{Y} \approx \mathtt{a})$.

## Implementation

In order to verify and experiment with the ideas presented in this paper, an implementation of the proposed proof procedure has been undertaken. For our implementation we extended an existing interpreter for the language $\mathcal{H}$ introduced in (Charalambidis et al. 2010; 2013). The modified interpreter supports the language $\mathcal{H}_{cn}$ and implements exactly the proof procedure presented in this section. The current version of the interpreter can be retrieved from `https://www.github.com/acharal/hopes`.

The source language[5] of the interpreter uses an extended Prolog-like syntax, similar to the one we used for motivation purposes in Section 2. Given a source program, the interpreter first derives the types of the predicates using a specialized but relatively straightforward type-inference algorithm. The well-typed program is then transformed to $\mathcal{H}_{cn}$-like syntax. The interpreter works interactively, ie., in the same way as ordinary Prolog interpreters. Every time the user specifies a goal, the type information that has been derived from the program is used to typecheck the goal. Subsequently, the goal together with the program is passed to a prover that implements the rules of the proposed proof procedure. The interpreter has been coded in Haskell and it implements the proof procedure as-is, namely it does not apply any low level compilation or any specialized optimizations.

## 5 The Soundness Theorem

In this section we state the soundness theorem for the proposed proof procedure. The proofs of the following lemmas can be found in the Appendix.

To specify the soundness theorem for the proof procedure, one has to first define the notion of *correct answer*:

**Definition 27.** Let $\mathsf{P}$ be a program and $\mathsf{G}$ be a goal. An answer $(\theta, \mathsf{Q})$ for $\mathsf{P} \cup \{\mathsf{G}\}$ is a basic substitution $\theta$ and a primitive goal $\mathsf{Q}$.

**Definition 28.** Let $\mathsf{P}$ be a program and let $\mathsf{G} = \leftarrow \mathsf{A}$ be a goal. An answer $(\theta, \leftarrow \mathsf{A}')$ is a *correct answer* for $comp(\mathsf{P}) \cup \{\mathsf{G}\}$ if for every model $M$ of $comp(\mathsf{P})$ and for every state $s$ over the domain of $M$, $[\![\mathsf{A}\theta]\!]_s(M) \sqsupseteq_o [\![\mathsf{A}']\!]_s(M)$.

The soundness theorem (Theorem 1 below) essentially amounts to demonstrating that every computed answer is a correct answer. To establish this fact, we need a lemma for each one of Definitions 21, 20 and 22. In particular, Lemma 2 demonstrates the correctness of Definition 21, Lemma 3 shows the correctness of Definition 22, and Lemma 4 the correctness of Definition 20 regarding single-step derivations. Lemma 5 actually shows the correctness of many-step derivations. The most demanding proof is that of Lemma 3. The proof of Lemma 3 (as-well-as the proof of the auxiliary Lemma 6 that will be given below) utilize one important and common assumption that exists in all previous works on constructive negation: we assume that "the language contains an infinite number of constants and functions" (see (Chan 1989)[page 478]).

**Lemma 2.** *Let $\mathsf{P}$ be a program, let $\mathsf{E}, \mathsf{E}'$ be body expressions of type $o$ and assume that $\mathsf{E}$ is reduced to $\mathsf{E}'$. Then, for every model $M$ of $comp(\mathsf{P})$ and for every state $s$ over the domain of $M$, it holds $[\![\mathsf{E}]\!]_s(M) = [\![\mathsf{E}']\!]_s(M)$.*

*Proof.* Straightforward using the semantics of $\mathcal{H}_{cn}$. □

**Lemma 3.** *Let $\mathsf{P}$ be a program, let $\mathsf{E}, \mathsf{E}'$ be body expressions of type $o$ and assume that $\mathsf{E}$ is negatively-reduced to $\mathsf{E}'$. Then, for every model $M$ of $comp(\mathsf{P})$ and for every state $s$ over the domain of $M$, it holds $[\![\mathsf{E}]\!]_s(M) = [\![\mathsf{E}']\!]_s(M)$.*

**Lemma 4.** *Let $\mathsf{P}$ be a program, let $\mathsf{G} = \leftarrow \mathsf{A}$ and $\mathsf{G}' = \leftarrow \mathsf{A}'$ be goals and let $\theta$ be a basic substitution such that $\mathsf{G} \xrightarrow{\theta} \mathsf{G}'$. Then, for every model $M$ of $comp(\mathsf{P})$ and for every state $s$ over the domain of $M$, it holds $[\![\mathsf{A}\theta]\!]_s(M) \sqsupseteq_o [\![\mathsf{A}']\!]_s(M)$.*

*Proof.* By case analysis and standard logical arguments. □

**Lemma 5.** *Let $\mathsf{P}$ be a program and $\mathsf{G} = \leftarrow \mathsf{A}$ be a goal. Let $\mathsf{G}_0 = \mathsf{G}, \mathsf{G}_1 = \leftarrow \mathsf{A}_1, \ldots, \mathsf{G}_n = \leftarrow \mathsf{A}_n$ be a derivation of length $n$ using basic substitutions $\theta_1, \ldots, \theta_n$. Then, for every model $M$ of $comp(\mathsf{P})$ and for every state $s$ over the domain of $M$, $[\![\mathsf{A}\theta_1 \cdots \theta_n]\!]_s(M) \sqsupseteq [\![\mathsf{A}_n]\!]_s(M)$.*

*Proof.* Using Lemma 4, Lemma 1 and induction on $n$. □

Finally, we will need another auxiliary lemma, which will be used to obtain the Soundness theorem:

**Lemma 6.** *Let $\mathsf{E}$ be a body expression of type $o$ and $\mathsf{G}$ a conjunction of primitive inequalities. Then, for all states $s$ and interpretations $I$ if $[\![\mathsf{E}]\!]_s(I) \sqsupseteq_o [\![\mathsf{G}]\!]_s(I)$ then $[\![\mathsf{E}]\!]_s(I) \sqsupseteq_o [\![\mathsf{G}']\!]_s(I)$ where $\mathsf{G}'$ is a conjunction of primitive inequalities that is obtained from $\mathsf{G}$ by restricting it to the free variables of $\mathsf{E}$.*

Using the above, we obtain the soundness theorem:

**Theorem 1** (Soundness Theorem)**.** *Let $\mathsf{P}$ be a program and $\mathsf{G}$ be a goal. Then, every computed answer for $\mathsf{P} \cup \{\mathsf{G}\}$ is a correct answer for $comp(\mathsf{P}) \cup \{\mathsf{G}\}$.*

*Proof.* A direct consequence of Lemma 5 using the auxiliary Lemma 6. □

[5]The language is called *HOPES* standing for Higher-Order Prolog with Extensional Semantics.

## 6 Future Work

We have introduced the higher-order language $\mathcal{H}_{cn}$ which supports constructive negation. There have been various proposals for higher-order logic programming languages, the most notable among them being Hilog (Chen, Kifer, and Warren 1989; 1993) and $\lambda$-Prolog (Miller and Nadathur 1986; Nadathur and Miller 1990); to our knowledge however, none of these languages has ever before been extended to support constructive negation.

One possible direction for future work is the investigation of completeness results for the proposed proof-procedure. This is an interesting venue for research but appears to be quite demanding due to the non-trivial nature of the proof procedure (note that, to our knowledge, no completeness results have been reported for the work of (Chan 1989)). Possibly a completeness result could be obtained for the case of finite-tree queries. An easier goal for future research would be to extend the language with other useful programming features that would enhance its higher-order style of programming (such as for example, type polymorphism).

## References

Barendregt, H. P. 1984. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland.

Chan, D. 1988. Constructive negation based on the completed database. In *ICLP/SLP*, 111–125.

Chan, D. 1989. An extension of constructive negation and its application in coroutining. In *NACLP*, 477–493.

Charalambidis, A.; Handjopoulos, K.; Rondogiannis, P.; and Wadge, W. W. 2010. Extensional higher-order logic programming. In Janhunen, T., and Niemelä, I., eds., *JELIA*, volume 6341 of *Lecture Notes in Computer Science*, 91–103. Springer.

Charalambidis, A.; Handjopoulos, K.; Rondogiannis, P.; and Wadge, W. W. 2013. Extensional higher-order logic programming. *ACM Trans. Comput. Log.* 14(3).

Chen, W.; Kifer, M.; and Warren, D. S. 1989. Hilog as a platform for database languages. *IEEE Data Eng. Bull.* 12(3):37–44.

Chen, W.; Kifer, M.; and Warren, D. S. 1993. Hilog: A foundation for higher-order logic programming. *J. Log. Program.* 15(3):187–230.

Clark, K. L. 1977. Negation as failure. In *Logic and Data Bases*, 293–322.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In Kowalski, R. A., and Bowen, K. A., eds., *ICLP/SLP*, 1070–1080. MIT Press.

Lifschitz, V. 2008. What is answer set programming? In Fox, D., and Gomes, C. P., eds., *AAAI*, 1594–1597. AAAI Press.

Lloyd, J. W. 1987. *Foundations of Logic Programming, 2nd Edition*. Springer.

Miller, D., and Nadathur, G. 1986. Higher-order logic programming. In Shapiro, E. Y., ed., *ICLP*, volume 225 of *Lecture Notes in Computer Science*, 448–462. Springer.

Nadathur, G., and Miller, D. 1990. Higher-order horn clauses. *J. ACM* 37(4):777–814.

## A Proof of Lemma 3

In this Appendix we establish Lemma 3. The proof requires the following three lemmas which can easily be established using the properties of Clark's Equality Theory. In particular, the proof of Lemma 9 requires our assumption regarding the infinite number of constants and functions of the language.

**Lemma 7.** *Let* $\mathsf{E}_1, \mathsf{E}_2$ *be body expressions of type* $\iota$. *Then, for every interpretation I that satisfies Clark's Equality Theory and for all states s over the domain of I, it holds:*

1. *if* $\mathsf{E}_1, \mathsf{E}_2$ *are unifiable with* $\theta = \{\mathsf{V}_1/\mathsf{A}_1, \ldots, \mathsf{V}_n/\mathsf{A}_n\}$ *then* $[\![\mathsf{E}_1 \approx \mathsf{E}_2]\!]_s(I) = [\![(\mathsf{V}_1 \approx \mathsf{A}_1) \wedge \cdots \wedge (\mathsf{V}_n \approx \mathsf{A}_n)]\!]_s(I)$
2. *if* $\mathsf{E}_1, \mathsf{E}_2$ *are not unifiable then* $[\![\mathsf{E}_1 \approx \mathsf{E}_2]\!]_s(I) = \mathit{false}$.

**Lemma 8.** *Let* $\mathsf{E}$ *be a body expression of type* $\iota$. *Then, for every interpretation I that satisfies Clark's Equality Theory and for all states* $s_1, s_2$ *over the domain of I, if* $[\![\mathsf{E}]\!]_{s_1}(I) = [\![\mathsf{E}]\!]_{s_2}(I)$ *then* $s_1(\mathsf{V}) = s_2(\mathsf{V})$ *for all the free variables* $\mathsf{V}$ *of* $\mathsf{E}$.

**Lemma 9.** *Let* $\mathsf{A}$ *be a conjunction of primitive inequalities. and let* $S_i$ *be a non empty subset of the free variables of* $\mathsf{A}_i$ *and* $U = \{\mathsf{U}_1, \ldots, \mathsf{U}_k\}$ *be the union of all* $S_i$. *Then, for all states s and interpretations I there exist* $d_1, \ldots d_k$ *such that* $[\![\mathsf{A}]\!]_{s[\mathsf{U}_1/d_1,\ldots \mathsf{U}_k/d_k]}(I) = \mathit{true}$.

We recall Lemma 3 and demonstrate its proof:

**Lemma 3.** *Let* $\mathsf{P}$ *be a program, let* $\mathsf{B}, \mathsf{B}'$ *be body expressions of type o and assume that* $\mathsf{B}$ *is negatively-reduced to* $\mathsf{B}'$. *Then, for every model M of comp*$(\mathsf{P})$ *and for every state s over the domain of M, it holds* $[\![\mathsf{B}]\!]_s(M) = [\![\mathsf{B}']\!]_s(M)$.

*Proof.* The proof is by a case analysis on the selected expression $\mathsf{A}_i$; the analysis follows the cases in Definition 22. When $\mathsf{A}_i$ has the structure implied by Cases 1-5 of Definition 22, it is direct to prove that $[\![\mathsf{B}]\!]_s(M) = [\![\mathsf{B}']\!]_s(M)$. Moreover, Case 8 follows easily by an inductive argument. The interesting Cases are 6, 7 and 9; Cases 10 and 11 can be obtained in an analogous way as Case 9. We give the proofs for Cases 6, 7 and 9.

<u>Case 6:</u> We examine subcases (a)-(d):
Subcase (a) follows from the fact that since $\sim\exists\hat{\mathsf{U}}(\mathsf{E}_1 \approx \mathsf{E}_2)$ is valid then for all $M$ and all $s$, $[\![\mathsf{E}_1 \approx \mathsf{E}_2]\!]_s(M) = \mathit{false}$ and therefore $[\![\mathsf{B}']\!]_s(M) = \mathit{true}$.

Subcase (b) follows easily by Lemma 7.

Consider now Subcase (c). Assume without loss of generality that $\mathsf{E}_1$ is a variable, say $\mathsf{X}$. We show that for all models $M$ of $\mathit{comp}(\mathsf{P})$ and for every state $s$ over the domain of $M$, it holds $[\![\mathsf{B}]\!]_s(M) = \mathit{false}$ if and only if $[\![\mathsf{B}']\!]_s(M) = \mathit{false}$. This is direct when $\mathsf{E}_2 = \mathsf{X}$. Assume that $\mathsf{E}_2 \neq \mathsf{X}$. Then $\mathsf{E}_2$ can not contain occurrences of $\mathsf{X}$ because otherwise the formula $\sim\exists\hat{\mathsf{U}}(\mathsf{X} \approx \mathsf{E}_2)$ would not be unsatisfiable. Assume now that $[\![\mathsf{B}]\!]_s(M) = \mathit{false}$. This implies that for some $\hat{d}$, $[\![\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{true}$, ie., that for all $j$, $[\![\mathsf{A}_j]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{true}$. In particular, $[\![\mathsf{X} \approx \mathsf{E}_2]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{true}$ and therefore $(s[\hat{\mathsf{U}}/\hat{d}])(\mathsf{X}) = [\![\mathsf{E}_2]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M)$. We show that $[\![\mathsf{B}']\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{false}$ by showing that $[\![\mathsf{A}'\theta]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{true}$, where $\theta = \{\mathsf{X}/\mathsf{E}_2\}$. By the Substitution Lemma (Lemma 1) we have that $[\![\mathsf{A}'\theta]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M)$ is equal to $[\![\mathsf{A}']\!]_{s[\hat{\mathsf{U}}/\hat{d}][\mathsf{X}/[\![\mathsf{E}_2]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M)]}(M)$ which by our previous remarks is equal to $[\![\mathsf{A}']\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M)$ and therefore equal to $\mathit{true}$. Consequently, $[\![\mathsf{B}']\!]_s(M) = \mathit{false}$. Conversely, assume that $[\![\mathsf{B}']\!]_s(M) = \mathit{false}$. This implies that for some $\hat{d}$, $[\![\mathsf{A}'\theta]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{true}$ which by the Substitution Lemma gives that $[\![\mathsf{A}']\!]_{s[\hat{\mathsf{U}}/\hat{d}][\mathsf{X}/[\![\mathsf{E}_2]\!]_{s[\hat{\mathsf{U}}/\hat{d}](M)}]}(M) = \mathit{true}$. But then, consider $\hat{d}'$ identical to $\hat{d}$ except that at the component corresponding to the variable $\mathsf{X}$ it has the value $[\![\mathsf{E}_2]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M)$. It is then immediate that $[\![\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n]\!]_{s[\hat{\mathsf{U}}/\hat{d}']}(M) = \mathit{true}$. In particular, $[\![\mathsf{X} \approx \mathsf{E}_2]\!]_{s[\hat{\mathsf{U}}/\hat{d}']}(M) = \mathit{true}$ because $(s[\hat{\mathsf{U}}/\hat{d}'])(\mathsf{X}) = [\![\mathsf{E}_2]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = [\![\mathsf{E}_2]\!]_{s[\hat{\mathsf{U}}/\hat{d}']}(M)$ because $\mathsf{E}_2$ does not contain occurrences of $\mathsf{X}$.

Consider Subcase (d). Assume without loss of generality that $\mathsf{E}_1$ is a variable, say $\mathsf{X}$. We show that for all models $M$ of $\mathit{comp}(\mathsf{P})$ and for every state $s$ over the domain of $M$, it holds $[\![\mathsf{B}]\!]_s(M) = \mathit{true}$ if and only if $[\![\mathsf{B}']\!]_s(M) = \mathit{true}$. Assume first that $[\![\mathsf{B}]\!]_s(M) = \mathit{true}$. This implies that for every $\hat{d}$, either $[\![\mathsf{X} \approx \mathsf{E}_2]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{false}$ or for some $j \neq i$, $[\![\mathsf{A}_j]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{false}$. Now, if the former condition holds then $[\![\sim\exists\hat{\mathsf{U}}_1(\mathsf{X} \approx \mathsf{E}_2)]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{true}$ and we are done. Otherwise, there exists some $\hat{d}$ such that $[\![\mathsf{X} \approx \mathsf{E}_2]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{true}$ and for some $j \neq i$, $[\![\mathsf{A}_j]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{false}$. Actually, it is $[\![\mathsf{X} \approx \mathsf{E}_2]\!]_{s[\hat{\mathsf{U}}'/\hat{d}']}(M) = \mathit{true}$ for all $\hat{\mathsf{U}}'$ that agree with $\hat{\mathsf{U}}$ on the variables $\hat{\mathsf{U}}_1$, and for every such $\hat{d}'$ there exists $j \neq i$ such that $[\![\mathsf{A}_j]\!]_{s[\hat{\mathsf{U}}'/\hat{d}']}(M) = \mathit{false}$. But then, $[\![(\mathsf{X} \approx \mathsf{E}_2) \wedge \sim\exists\hat{\mathsf{U}}_2\mathsf{A}']\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{true}$, ie., $[\![\mathsf{B}']\!]_s(M) = \mathit{true}$. Conversely, assume that $[\![\mathsf{B}']\!]_s(M) = \mathit{true}$. If $[\![\sim\exists\hat{\mathsf{U}}_1\mathsf{A}_i]\!]_s(M) = \mathit{true}$ then this easily implies that $[\![\mathsf{B}]\!]_s(M) = \mathit{true}$. Assume now that $[\![\sim\exists\hat{\mathsf{U}}_1\mathsf{A}_i]\!]_s(M) = \mathit{false}$ and that $[\![\exists\hat{\mathsf{U}}_1(\mathsf{A}_i \wedge \sim\exists\hat{\mathsf{U}}_2\mathsf{A}')]\!]_s(M) = \mathit{true}$. This means that there exists $\hat{d}_1$ such that it holds $[\![\mathsf{A}_i \wedge \sim\exists\hat{\mathsf{U}}_2\mathsf{A}']\!]_{s[\hat{\mathsf{U}}_1/\hat{d}_1]}(M) = \mathit{true}$. But then, recalling that $\mathsf{X} \notin \hat{\mathsf{U}}_1$ and using Lemma 8 on expression $\mathsf{E}_2$, one can easily see that for all $\hat{d}_1'$ such that $[\![\mathsf{A}_i]\!]_{s[\hat{\mathsf{U}}_1/\hat{d}_1']}(M) = \mathit{true}$ it also holds $[\![\sim\exists\hat{\mathsf{U}}_2\mathsf{A}']\!]_{s[\hat{\mathsf{U}}_1/\hat{d}_1']}(M) = \mathit{true}$. This easily implies that $[\![\mathsf{B}]\!]_s(M) = \mathit{true}$.

<u>Case 7:</u> We examine subcases (a) and (b):

For Subcase (a), assume first that $[\![\mathsf{B}]\!]_s(M) = \mathit{false}$, ie., that there exists $\hat{d}$ such that $[\![\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{true}$. This means that $[\![(\mathsf{R}\,\mathsf{E})]\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{true}$ and $[\![\mathsf{A}']\!]_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{true}$. We claim that $[\![\mathsf{B}']\!]_s(M) = \mathit{false}$, ie., $\llbracket \sim\exists\hat{\mathsf{U}}'(\mathsf{A}'\theta)\rrbracket_s(M) = \mathit{false}$. It suffices to find a $\hat{d}'$ such that $\llbracket\mathsf{A}'\theta\rrbracket_{s[\hat{\mathsf{U}}'/\hat{d}']}(M) = \mathit{true}$. Let $r$ be the relation that $\hat{\mathsf{U}}$ assigns to $\mathsf{R}$. Since $\llbracket(\mathsf{R}\,\mathsf{E})\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{true}$ we have that $\llbracket\mathsf{E}\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}]}(M) \in r$. Let $r' = r - \{\llbracket\mathsf{E}\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}]}(M)\}$ and assume that $[\hat{\mathsf{U}}'/\hat{d}']$ is identical to $[\hat{\mathsf{U}}/\hat{d}]$ the only exception being that the former assigns to $\mathsf{R}'$ the relation $r'$ while the latter assigns to $\mathsf{R}$ the relation $r$. Then, we have that $\llbracket\mathsf{A}'\theta\rrbracket_{s[\hat{\mathsf{U}}'/\hat{d}']}(M)$ is equal (using the Substitution Lemma) to $\llbracket\mathsf{A}'\rrbracket_{s[\hat{\mathsf{U}}'/\hat{d}'][\mathsf{R}/\llbracket\theta(\mathsf{R})\rrbracket_{s[\hat{\mathsf{U}}'/\hat{d}']}(M)]}(M)$, which according to the above discussion is equal to $\llbracket\mathsf{A}'\theta\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}]}(M)$ and therefore equal to $\mathit{true}$. Consequently, $\llbracket\mathsf{B}'\rrbracket_s(M) = \mathit{false}$. Conversely, assume that $\llbracket\mathsf{B}'\rrbracket_s(M) = \mathit{false}$, that is $\llbracket \sim\exists\hat{\mathsf{U}}'(\mathsf{A}'\theta)\rrbracket_s(M) = \mathit{false}$. This means that there exists some $\hat{d}'$ such that $\llbracket\mathsf{A}'\theta\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}']}(M) = \mathit{true}$. Using the Substitution Lemma we get that $\llbracket\mathsf{A}'\rrbracket_{s[\hat{\mathsf{U}}'/\hat{d}'][\mathsf{R}/\llbracket\theta(\mathsf{R})\rrbracket_{s[\hat{\mathsf{U}}'/\hat{d}']}(M)]}(M) = \mathit{true}$. We claim that $\llbracket\mathsf{B}\rrbracket_s(M) = \mathit{false}$, or equivalently that $\llbracket \sim\exists\hat{\mathsf{U}}(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n)\rrbracket_s(M) = \mathit{false}$, ie., that there exists $\hat{d}$ such that $\llbracket\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{true}$. Take the assignment $[\hat{\mathsf{U}}/\hat{d}]$ which is identical to $[\hat{\mathsf{U}}'/\hat{d}']$ the only difference being that $\hat{\mathsf{U}}$ assigns to $\mathsf{R}$ the value $\llbracket\lambda\mathsf{X}.(\mathsf{X}\approx\mathsf{E})\bigvee\mathsf{R}'\rrbracket_{s[\hat{\mathsf{U}}'/\hat{d}']}(M)$. This satisfies the statement $\llbracket\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{true}$, which gives the desired result, namely $\llbracket\mathsf{B}\rrbracket_s(M) = \mathit{false}$.

For Subcase (b), assume first that $\llbracket\mathsf{B}\rrbracket_s(M) = \mathit{true}$, ie., that for all $\hat{d}$, $\llbracket\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{false}$. Assume first that for all $\hat{d}$, $\llbracket(\mathsf{R}\,\hat{\mathsf{E}})\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{false}$. Then, we have $\llbracket \sim\exists\hat{\mathsf{U}}_1(\mathsf{R}\,\hat{\mathsf{E}})\rrbracket_s(M) = \mathit{true}$ and we are done. If on the other hand there exists $\hat{d}$ such that $\llbracket(\mathsf{R}\,\hat{\mathsf{E}})\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{true}$ then $\llbracket\mathsf{A}'\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{false}$; actually, $\llbracket\mathsf{A}'\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}']}(M) = \mathit{false}$ for all $\hat{d}'$ that agree with $\hat{d}$ on the variables $\hat{\mathsf{U}}_2$ (that occur only in $\mathsf{A}'$). But then, $\llbracket\exists\hat{\mathsf{U}}_1((\mathsf{R}\,\hat{\mathsf{E}})\wedge \sim\exists\hat{\mathsf{U}}_2\mathsf{A}')\rrbracket_s(M) = \mathit{true}$. Conversely, assume that $\llbracket\mathsf{B}'\rrbracket_s(M) = \mathit{true}$. Then, if we have that $\llbracket \sim\exists\hat{\mathsf{U}}_1(\mathsf{R}\,\hat{\mathsf{E}})\rrbracket_s(M) = \mathit{true}$ it obviously holds $\llbracket\mathsf{B}\rrbracket_s(M) = \mathit{true}$. Otherwise, $\llbracket\exists\hat{\mathsf{U}}_1(\mathsf{A}_i \wedge \sim\exists\hat{\mathsf{U}}_2\mathsf{A}') \wedge \mathsf{B}\rrbracket_s(M) = \mathit{true}$ and since one of the conjuncts is $\mathsf{B}$, we immediately get $\llbracket\mathsf{B}\rrbracket_s(M) = \mathit{true}$.

<u>Case 9:</u> We examine subcases (a) and (b):
For Subcase (a), assume first that $\llbracket\mathsf{B}\rrbracket_s(M) = \mathit{true}$, ie., that for all $\hat{d}$, $\llbracket\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{false}$. Since $\mathsf{A}_i$ has some free variables in $\hat{\mathsf{U}}$ it follows from Lemma 9 that there exists $\hat{d}'$ such that $\llbracket\mathsf{A}_i\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}']}(M) = \mathit{true}$; actually consider the inequalities in $\mathsf{A}'$ that have variables in $\hat{\mathsf{U}}$: there exists $\hat{d}'$ such that the conjunction of $\mathsf{A}_i$ and these inequalities is also equal to $\mathit{true}$. For that $\hat{d}'$, it follows that $\llbracket\mathsf{A}'\rrbracket_{s[\mathsf{U}/\hat{d}']}(M) = \mathit{false}$. We will show then that this implies that for all $\hat{d}$ it holds $\llbracket\mathsf{A}'\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}]}(M) = \mathit{false}$. Since $\mathsf{A}'$ is a conjunction of inequalities we distinguish two cases. First, if the inequality (say $\mathsf{A}_j$) contains variables in $\hat{\mathsf{U}}$ we know that for that $\hat{d}'$, $\llbracket\mathsf{A}_j\rrbracket_s[\hat{\mathsf{U}}/\hat{d}'] = \mathit{true}$. In order for the conjunction to be equal to $\mathit{false}$, some of the remaining inequalities has to be equal to $\mathit{false}$ for the specific $\hat{d}'$. If the inequality (say $\mathsf{A}_j$) does not contain any variables in $\hat{\mathsf{U}}$ it easily follows that for all $\hat{d}$, $\llbracket\mathsf{A}_j\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}]} = \llbracket\mathsf{A}_j\rrbracket_{s[\hat{\mathsf{U}}/\hat{d}']}$. The other direction of the proof is straightforward.

For Subcase (b) consider the formula $\sim\exists\hat{\mathsf{U}}(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n)$ which is logically equivalent to $\sim\exists\hat{\mathsf{U}}((\sim\exists\mathsf{V}(\mathsf{E}_1 \approx \mathsf{E}_2) \wedge \mathsf{A}')$. Since the formula $\sim\exists\mathsf{V}(\mathsf{E}_1 \approx \mathsf{E}_2)$ does not contain free variables in $\hat{\mathsf{U}}$, we can move inside the existential quantifier getting $\sim((\sim\exists\mathsf{V}(\mathsf{E}_1 \approx \mathsf{E}_2) \wedge \exists\hat{\mathsf{U}}\mathsf{A}')$. This latter formula is logically equivalent to $\exists\mathsf{V}(\mathsf{E}_1 \approx \mathsf{E}_2) \vee \sim\exists\hat{\mathsf{U}}\mathsf{A}'$. □

## B Proof of Lemma 6

**Lemma 6.** *Let* $\mathsf{E}$ *be a body expression of type* $o$ *and* $\mathsf{G}$ *a conjunction of primitive inequalities. Then, for all states* $s$ *and interpretations* $I$ *if* $\llbracket\mathsf{E}\rrbracket_s(I) \sqsupseteq_o \llbracket\mathsf{G}\rrbracket_s(I)$ *then* $\llbracket\mathsf{E}\rrbracket_s(I) \sqsupseteq_o \llbracket\mathsf{G}'\rrbracket_s(I)$ *where* $\mathsf{G}'$ *is a conjunction of primitive inequalities that is obtained from* $\mathsf{G}$ *by restricting it to the free variables of* $\mathsf{E}$.

*Proof.* We will write $\mathsf{G}$ as $\mathsf{G}' \wedge \mathsf{A}$ where $\mathsf{A} = \mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n$ is a conjunction of primitive inequalities that have been removed from $\mathsf{G}$. Let $S_i$ be set of the free variables of $\mathsf{A}_i$ that do not occur in $\mathsf{E}$. Let $U$ be the union of $S_i$ and $\mathsf{U}_j \in U$ where $j \in \{1, \ldots, k\}$. Then, by Lemma 9 it follows that there exist $d_1, \ldots, d_k$ such that $\llbracket\mathsf{A}\rrbracket_{s'}(I) = \mathit{true}$ where $s' = s[\mathsf{U}_1/d_1, \ldots, \mathsf{U}_k/d_k]$. It suffices to show that if $\llbracket\mathsf{E}\rrbracket_s(I) = \mathit{false}$ then $\llbracket\mathsf{G}'\rrbracket_s(I) = \mathit{false}$. If $\llbracket\mathsf{E}\rrbracket_s(I) = \mathit{false}$ it is implied that $\llbracket\mathsf{G}\rrbracket_s(I) = \mathit{false}$. It follows that $\bigsqcap\{\llbracket\mathsf{G}'\rrbracket_s(I), \llbracket\mathsf{A}\rrbracket_s(I)\} = \mathit{false}$ and since we know that $\llbracket\mathsf{A}\rrbracket_{s'}(I) = \mathit{true}$ we have that $\llbracket\mathsf{G}'\rrbracket_{s'}(I) = \mathit{false}$. By the construction of $\mathsf{G}'$ we know that it does not contain any free variable in $U$, so it implies that $\llbracket\mathsf{G}'\rrbracket_{s'}(I) = \llbracket\mathsf{G}'\rrbracket_s(I)$. □