Extensional Higher-Order Logic Programming

A. CHARALAMBIDIS, K. HANDJOPOULOS, P. RONDOGIANNIS, University of Athens
W. W. WADGE, University of Victoria

We propose a purely extensional semantics for higher-order logic programming. In this semantics program predicates denote sets of ordered tuples, and two predicates are equal iff they are equal as sets. Moreover, every program has a unique minimum Herbrand model which is the greatest lower bound of all Herbrand models of the program and the least fixed-point of an immediate consequence operator. We also propose an SLD-resolution proof system which is proven sound and complete with respect to the minimum Herbrand model semantics. In other words, we provide a purely extensional theoretical framework for higher-order logic programming which generalizes the familiar theory of classical (first-order) logic programming.

Categories and Subject Descriptors: D.1.6 [**Software**]: Logic Programming

General Terms: Design, Languages, Theory

ACM Reference Format:
ACM Trans. Comput. Logic V, N, Article A (January YYYY), 40 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The two most prominent declarative paradigms, namely logic and functional programming, differ radically in an important aspect: logic programming is traditionally first-order while functional programming encourages and promotes the use of higher-order functions and constructs. This difference can be partially explained by the fact that higher-order logic fails in terms of vital properties such as completeness and compactness. It would seem, on the face of it, that there would be no hope of finding a complete resolution proof system for higher-order logic programming.

The initial attitude of logic programmers towards higher-order logic programming was somewhat skeptical: it was often argued (see for example [Warren 1982]) that there exist ways of encoding or simulating higher-order programming inside Prolog itself. However, ease of use is a primary criterion for a programming language, and the fact that higher-order features can be simulated or encoded does not mean that it is practical to do so.

Eventually extensions with genuine higher-order capabilities were introduced - roughly speaking, extensions which allow predicates to be applied but also passed as parameters. The existing proposals can be placed in two main categories, namely the

Author's addresses: A. Charalambidis, K. Handjopoulos, P. Rondogiannis, Department of Informatics and Telecommunications, University of Athens, Panepistimiopolis, 15784 Athens, Greece, emails: a.charalambidis@di.uoa.gr, khandj@gmail.com, prondo@di.uoa.gr; W. W. Wadge, PO Box 3055, STN CSC, Victoria, BC, Canada V8W 3P6, email: wwadge@csr.uvic.ca.
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.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

intensional and the *extensional* ones. In the former category, the two most prominent languages are λProlog [Miller and Nadathur 1986; Nadathur 1987] and HiLog [Chen et al. 1989; 1993]. The latter category is much less developed: currently (to our knowledge) there exist two main proposals for extensional higher-order logic programming, namely [Wadge 1991] and [Bezem 1999; 2001], but no actual systems have been built so far.

In an extensional language, two predicates that succeed for the same instances are considered equal. On the other hand, in an intensional language it is possible that predicates that are equal as sets will not be treated as equal. In other words, a predicate in an intensional language is more than just the set of arguments for which it is true. For example, in Hilog, two predicates are not considered equal unless their names are the same. The distinction between extensionality and intensionality has been widely discussed in the literature (see for example the detailed presentation in [Chen et al. 1993])[1] and can be intuitively understood by the following examples:

Example 1.1. Suppose we have a database of professions, both of their membership and their status. We might have rules such as:

```
engineer(tom).
engineer(sally).
programmer(harry).
```

with `engineer` and `programmer` used as predicates. In intensional higher-order logic programming we could also have rules in which these are arguments, eg:

```
profession(engineer).
profession(programmer).
```

Now suppose `tom` and `sally` are also avid users of Twitter. We could have rules:

```
tweeter(tom).
tweeter(sally).
```

Notice now that the predicates `tweeter` and `engineer` are equal as sets (since they are true for the same objects, namely `tom` and `sally`). If we attempted to understand the above program from an extensional point of view, then we would have to accept that `profession(tweeter)` must also hold (since `tweeter` and `engineer` are indistinguishable as sets). It is clear that the extensional interpretation in this case is completely unnatural. The program can however be understood intensionally: the predicate `profession` is true of the *name* `engineer` (which is different than the name `tweeter`). □

On the other hand, there are cases where predicates can be understood from an extensional point of view:

Example 1.2. Consider a program that consists only of the following rule:

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

In an extensional language, predicate `p` above can be intuitively understood in purely set-theoretic terms: `p` is the set of all those sets that contain both `0` and `1`.

It should be noted that the above program is also a syntactically acceptable program of the existing intensional logic programming languages. The difference is that

[1]The notions of "extension" and "intension" have a long history and their roots can be traced back in the study of natural languages and their semantics. The distinction between the two notions is attributed to Frege, although the actual terms were coined later by R. Carnap. An excellent discussion of these notions and their implications in the study of natural languages, can be found in [Dowty et al. 1981].

in an extensional language the above program has a purely set-theoretic semantics. Actually, as we are going to see, this set theoretic interpretation allows us to permit queries of the form:

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

which will get meaningful answers (the answer in this case will express the fact that `R` is any relation which is true of both `0` and `1`). Notice that an intensional language will not in general provide an answer in such a query (since there does not exist any actual predicate defined in the program that is true of both `0` and `1`). □

In this paper we will focus on extensional higher-order logic programming. The first work in this area was [Wadge 1991]. In that paper, W. W. Wadge demonstrated that there exists a modest fragment of higher-order logic programming that can be understood in purely extensional terms. More specifically, Wadge discovered a simple syntactic restriction which ensured that compliant programs have an extensional declarative reading. The restriction forbids user-defined predicates to appear as arguments in the heads of clauses. For example, the two rules cited already:

```
profession(engineer).
profession(programmer).
```

violate the restriction. Roughly speaking, the restriction says that rules about predicates can state general principles but cannot pick out a particular predicate for special treatment. Wadge gave several examples of useful extensional higher-order programs and outlined the proof of a minimum-model result. He also showed that in this model the denotations of program predicates are continuous. Continuity can intuitively be understood as a kind of finitaryness. For example, if `foo(p)` succeeds and `foo` is continuous, it means there is a finite set of arguments $\{\mathtt{a}_1, \ldots, \mathtt{a}_k\}$ for which $\mathtt{p}(\mathtt{a}_1), \ldots, \mathtt{p}(\mathtt{a}_k)$ all succeed, and if $\mathtt{q}(\mathtt{a}_1), \ldots, \mathtt{q}(\mathtt{a}_k)$ also succeed, then `foo(q)` succeeds. Finally, Wadge conjectured that a sound and complete proof system exists for his fragment, but did not further pursue such an investigation. A detailed discussion of Wadge's approach (including its problems and shortcomings) is given in Section 8.

In this paper we extend the study initiated in [Wadge 1991] and derive the first, to our knowledge, complete theoretical framework for extensional higher-order logic programming, both from a semantic as-well-as from a proof theoretic point of view.

Our first contribution is the development of a novel extensional semantics for higher-order logic programming that is based on *algebraic lattices* (see for example [Grätzer 1978]), a subclass of the familiar complete lattices that have traditionally been used in the theory of first-order logic programming. For every predicate type of our language, algebraic lattices single out a subset of "finite" objects of that type. In other words, the proposed semantics reflects in a direct way the finitary nature of continuity that is implicit in [Wadge 1991]. The benefit of the new approach compared to that of [Wadge 1991] is that all basic properties and results of classical logic programming are now transferred in the higher-order setting in a natural way. Moreover, the new semantics leads to a relatively simple, sound and complete proof system even for a language that is genuinely more powerful than the one considered in [Wadge 1991]. More details on the connections between the two semantical approaches will be given in Section 8.

Our second contribution fixes a shortcoming of Wadge's language by allowing clause bodies and program goals to have uninstantiated higher-order variables. To understand the importance of this extension, consider the following rule for `bands` (musical ensembles):

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

This says that a band is a group that has at least a singer, a drummer, and a guitarist. Suppose that we also have a database of musicians:

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

Our extensional higher-order language allows the query `?-band(B)`. At first sight a query like this seems impractical if not impossible to implement. Since a band is a set, bands can be very large and there can be many (possibly uncountably many) of them. In existing intensional systems such queries usually fail since the program does not provide any information about any particular band.

However, in an extensional context the finitary behavior of the predicates of our language, saves us. If the predicate `band` declares a relation to be a band, then (due to the finitaryness described above) it must have examined only finitely many members of the relation. Therefore we can enumerate the bands by enumerating *finite* bands, and this collection is countable (in this particular example it is actually finite). Actually, as we are going to see, this enumeration can be performed in a careful way so as that it avoids producing all finite relations one by one (see the discussion in Section 2 that follows).

Our final contribution is a relatively simple proof system for extensional higher-order logic programming, which extends classical SLD-resolution. We demonstrate that the new proof system is sound and complete with respect to the proposed semantics. In particular, the derived completeness theorems generalize the well-known such theorems for first-order logic programming. This result may, at first sight, also seem paradoxical, given the well-known failure of completeness for even second-order logic. But the paradox is resolved by recalling that we are dealing with a restricted subset of higher-order logic and that the denotations of the types of our language are not arbitrary sets but instead algebraic lattices (which have a much more refined structure).

One very important benefit of the proof system is that it gives us an operational semantics for our language. This means in turn that we could probably extend it with cut, negation and other operational features not easily specified in terms of model theory alone.

The rest of the paper is organized as follows: Section 2 presents in a more detailed manner the basic ideas developed in this paper. Section 3 introduces the syntax of the higher-order logic programming language $\mathcal{H}$. Section 4 introduces the key lattice-theoretic notions that will be needed in the development of the semantics. Sections 5 and 6 develop the semantics and the minimum Herbrand model semantics of $\mathcal{H}$; the main properties of the semantics are also established. Section 7 introduces an SLD-resolution proof system for $\mathcal{H}$ and establishes its soundness and completeness. Section 8 presents a description of related approaches to higher-order logic programming. Section 9 briefly discusses implementation issues and presents certain interesting topics for future work. The lengthiest among the proofs have been moved to corresponding appendices in order to enhance the readability of the paper.

2. THE PROPOSED APPROACH: AN INTUITIVE OVERVIEW

The purpose of this paper is to develop a purely extensional theoretical framework for higher-order logic programming which will generalize the familiar theory of first-order logic programming. The first problem we consider is to bypass one important restriction of [Wadge 1991], namely the inability to handle program clauses or queries that

contain uninstantiated predicate variables. The following example illustrates these ideas:

Example 2.1. Consider the following higher-order logic program written in an extended Prolog-like syntax:

```
p(Q):-Q(0),Q(s(0)).
nat(0).
nat(s(X)):-nat(X).
```

The Herbrand universe of the program is the set of natural numbers in successor notation. According to the semantics of [Wadge 1991], the least Herbrand model of the program assigns to predicate `p` a continuous relation which is true of *all* unary relations that contain at least `0` and `s(0)`. Consider now the query:

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

which asks for all relations that satisfy `p`. Such a query seems completely unreasonable, since there exist uncountably many relations that must be substituted and tested in the place of `R`. □

The above example illustrates why uninstantiated predicate variables in clauses were disallowed in [Wadge 1991]. From a theoretical point of view, one could extend the semantics to cover such cases, but the problem is mainly a practical one: "how can one implement such programs and queries?".

In more formal terms, the least Herbrand model of a higher-order program under the semantics of [Wadge 1991] is in general an *uncountable set*; in our example, this is evidenced by the fact that there exists an uncountable number of unary relations over the natural numbers that contain both `0` and `s(0)`. This observation comes in contrast with the semantics of first-order logic programming in which the least Herbrand model of a program is a countable set. How can one define a proof system that is sound and complete with respect to this semantics? The key idea for bypassing these problems was actually anticipated in the concluding section of [Wadge 1991]:

> *Our higher order predicates, however, are continuous: if a relation satisfies a predicate, then some finite subset satisfies it. This means that we have to examine only finite relations.*

In the above example, despite the fact that there exists an infinite number of relations that satisfy `p`, all these relations are supersets of the finite relation $\{$`0`, `s(0)`$\}$. In some sense, this finite relation *represents* all the relations that satisfy `p`. But how can we make the notion of "finiteness" more explicit? In order to define a sound and complete proof system for an interesting extensional higher-order logic programming language, our semantics must in some sense reflect the above "finitary" concepts more explicitly.

An idea that springs to mind is to define an alternative semantics in which variables (like `Q` in Example 2.1) range over finite relations (and not over arbitrary relations as in [Wadge 1991]). Of course, the notion of "finite" should be appropriately defined for every predicate type. But then an immediate difficulty appears to arise. Given again the program in Example 2.1, and the query

```
?-p(nat).
```

it is not immediately obvious what the meaning of the above is. Since we have assumed that `Q` ranges over finite relations, how can `p` be applied to an infinite one? To overcome this problem, observe that in order for the predicate `p` to succeed for its argument `Q`, it only has to examine a "finite number of facts" about `Q` (namely whether `Q` is true of `0` and `s(0)`). This remark suggests that the meaning of `p(nat)` can be established

following a non-standard interpretation of application: we apply the meaning of p to all the "finite approximations" of the meaning of nat, ie., to all finite subsets of the set $\{0, s(0), s(s(0)), \ldots\}$. In our case p(nat) will be true since there exists a finite subset of the meaning of nat for which the meaning of p is true (namely the set $\{0, s(0)\}$).

Notice that the new semantical approach outlined above, heavily relies on the idea that the meaning of predicates (like nat) can be expressed as the least upper bound of a set of simpler (in this case, finite) relations. Actually, this is an old and well-known assumption in the area of denotational semantics, as the following excerpt from [Stoy 1977][page 98] indicates:

> *So we may reasonably demand of all the value spaces in which we hope to compute that they come equipped with a particular countable subset of elements from which all the other elements may be built up.*

As we are going to demonstrate, the meaning of *every* predicate defined in our language possesses the property just mentioned, and this allows us to use the new non-standard semantics of application. In fact, as we are going to see, for every predicate type of our language, the set of possible meanings of this type forms an *algebraic lattice* [Grätzer 1978]; then, the above property is nothing more than the key property which characterizes algebraic lattices (see Proposition 4.14), namely that "*every element of an algebraic lattice is the least upper bound of the compact elements of the lattice that are below it*". More importantly, for the algebraic lattices we consider, it is relatively easy to identify these compact elements and to enumerate them one by one. Based on the above semantics, we are able to derive for higher-order logic programs many properties that are either identical or generalize the familiar ones from first-order logic programming (see Section 6).

The new semantics allows us to introduce a relatively simple, sound and complete proof system which applies to programs and queries that may contain uninstantiated predicate variables. This is due to the fact that the set of "finite" relations is now countable, and as we are going to see, there exist interesting ways of producing and enumerating them. The key idea can be demonstrated by continuing Example 2.1. Given the query:

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

one (inefficient and tedious) approach would be to enumerate all possible finite relations of the appropriate type over the Herbrand universe. Instead of this, we use an approach which is based on what we call *basic templates*: a basic template for R is (intuitively) a finite set whose elements are individual variables. This saves us from having to enumerate all finite sets consisting of ground terms from the Herbrand universe. For example[2], assume that we instantiate R with the template $\{X, Y\}$. Then, the resolution proceeds as follows:

```
?-p(R)
?-p({X,Y})
?-{X,Y}(0),{X,Y}(s(0))
?-{0,Y}(s(0))
□
```

and the proof system will return the answer $R = \{0, s(0)\}$. The proof system will also return other finite solutions, such as $R = \{0, s(0), Z_1\}$, $R = \{0, s(0), Z_1, Z_2\}$, and so on. However, a slightly optimized implementation (see Section 9) can be created that returns only the answer $R = \{0, s(0)\}$, which represents all the finite relations produced

[2]The notation we use for representing basic templates will slightly change in Section 7.

by the proof system. The intuition behind the above answer is that the given query succeeds for all unary relations that contain at least `0` and `s(0)`. Similarly, for the `band` example of Section 1, the implementation will systematically assemble all the minimal three-member bands from the talents available.

3. THE HIGHER-ORDER LANGUAGE $\mathcal{H}$: SYNTAX

In this section we introduce the higher-order language $\mathcal{H}$, which extends classical first-order logic programming to a higher-order setting. The language $\mathcal{H}$ is based on a simple type system that supports two base types: o, the boolean domain, and ι, the domain of individuals (data objects). The composite types are partitioned into three classes: functional (assigned to function symbols), predicate (assigned to predicate symbols) and argument (assigned to parameters of predicates).

Definition 3.1. A type can either be *functional*, *argument*, or *predicate*, denoted by σ, ρ and π respectively and defined as:

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

We will use τ to denote an arbitrary type (either functional, argument or predicate one).

As usual, the binary operator $\to$ is right-associative. A functional type that is different than ι will often be written in the form $\iota^n \to \iota$, $n \geq 1$. Moreover, it can be easily seen that every predicate type π 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 can now proceed to the definition of $\mathcal{H}$, starting from its alphabet:

Definition 3.2. The *alphabet* of the higher-order language $\mathcal{H}$ consists of the following:

(1) *Predicate variables* of every predicate type π (denoted by capital letters such as $\mathsf{P}, \mathsf{Q}, \mathsf{R}, \ldots$).
(2) *Predicate constants* of every predicate type π (denoted by lowercase letters such as $\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$).
(3) *Individual variables* of type ι (denoted by capital letters such as $\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \ldots$).
(4) *Individual constants* of type ι (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 false and 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 π; the generalized inverse implication constants $\leftarrow_\pi$, of type $\pi \to \pi \to o$, for every predicate type π; the existential quantifier $\exists_\rho$, of type $(\rho \to o) \to o$, for every argument type ρ.
(7) The abstractor λ and the parentheses "(" and ")".

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

The existential quantifier in higher-order logic is usually introduced in a different way than in first-order logic. So, for example, in order to express the quantification of the argument variable V of type ρ over the expression E one writes $(\exists_\rho (\lambda \mathsf{V}.\mathsf{E}))$. For simplicity, we will use in this paper the more familiar notation $(\exists_\rho \mathsf{V}\, \mathsf{E})$.

We proceed by defining the set of *positive expressions* of $\mathcal{H}$:

Definition 3.3. The set of positive expressions of the higher-order language $\mathcal{H}$ is recursively defined as follows:

(1) Every predicate variable (respectively, predicate constant) of type π is a positive expression of type π; every individual variable (respectively, individual constant) of type ι is a positive expression of type ι; the propositional constants false and true are positive expressions of type o.
(2) If f is an n-ary function symbol and $\mathsf{E}_1, \ldots, \mathsf{E}_n$ are positive expressions of type ι, then $(\mathsf{f}\ \mathsf{E}_1 \cdots \mathsf{E}_n)$ is a positive expression of type ι.
(3) If E_1 is a positive expression of type $\rho \to \pi$ and E_2 is a positive expression of type ρ, then $(\mathsf{E}_1\mathsf{E}_2)$ is a positive expression of type π.
(4) If V is an argument variable of type ρ and E is a positive expression of type π, then $(\lambda\mathsf{V}.\mathsf{E})$ is a positive expression of type $\rho \to \pi$.
(5) If $\mathsf{E}_1, \mathsf{E}_2$ are positive expressions of type π, then $(\mathsf{E}_1 \bigwedge_\pi \mathsf{E}_2)$ and $(\mathsf{E}_1 \bigvee_\pi \mathsf{E}_2)$ are positive expressions of type π.
(6) If $\mathsf{E}_1, \mathsf{E}_2$ are positive expressions of type ι, then $(\mathsf{E}_1 \approx \mathsf{E}_2)$ is a positive expression of type o.
(7) If E is an expression of type o and V is an argument variable of type ρ, then $(\exists_\rho \mathsf{V}\ \mathsf{E})$ is a positive expression of type o.

The notions of *free* and *bound* variables of a positive expression are defined as usual. A positive expression is called *closed* if it does not contain any free variables.

The set of *clausal expressions* of $\mathcal{H}$ can now be specified:

Definition 3.4. The set of *clausal expressions* of the higher-order language $\mathcal{H}$ is defined as follows:

(1) If p is a predicate constant of type π and E is a closed positive expression of type π then $\mathsf{p} \leftarrow_\pi \mathsf{E}$ is a clausal expression of $\mathcal{H}$, also called a *program clause*.
(2) If E is a positive expression of type o, then $\mathsf{false} \leftarrow_o \mathsf{E}$ (usually denoted by $\leftarrow_o \mathsf{E}$ or just $\leftarrow \mathsf{E}$) is a clausal expression of $\mathcal{H}$, also called a *goal clause*.

All clausal expressions of $\mathcal{H}$ 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 $\leftarrow \mathsf{true}$, ie., to the propositional constant false.

The union of the sets of positive and clausal expressions of $\mathcal{H}$ will be called the set of *expressions* of $\mathcal{H}$. To denote that an expression E has type τ, we will often write $\mathsf{E} : \tau$; additionally, we write $type(\mathsf{E})$ to denote the type of expression E. Expressions of type ι will be called *terms* and of type o will be called *formulas*. We will write $\leftarrow$, $\wedge$ and $\vee$ instead of $\leftarrow_o$, $\bigwedge_o$ and $\bigvee_o$. Moreover, instead of $\exists_\rho$ we will often write $\exists$. When writing an expression, in order to avoid the excessive use of parentheses, certain usual conventions will be adopted (such as for example the usual priorities between logical constants, the convention that application is left-associative and that lambda abstraction extends as far to the right as possible, and so on). Given an expression E, we denote by $FV(\mathsf{E})$ the set of all free variables of E. By overloading notation, we will also write $FV(S)$, where S is a set of expressions.

Notice that in Definition 3.4 above, a goal clause may contain two types of occurrences of variables that serve a similar purpose, namely free argument variables and argument variables that are existentially quantified. From a semantic point of view, these two types of variables are essentially the same. However, in a later section we will distinguish them from an operational point of view: the free argument variables

that appear in a goal are the ones for which an answer is sought for by the proof system; the argument variables that are existentially quantified are essentially free variables for which an answer is not sought for (something like the underscored variables in Prolog systems). This distinction is not an important one, and we could have proceeded in a different way (eg. by disallowing existentially quantified variables from goals).

Definition 3.5. A program of $\mathcal{H}$ is a finite set of program clauses of $\mathcal{H}$.

Example 3.6. The following is a higher-order program that computes the closure of its input binary relation `R`. The type of `closure` is $\pi = (\iota \to \iota \to o) \to \iota \to \iota \to o$.

$$\texttt{closure} \leftarrow_\pi \lambda\texttt{R}.\lambda\texttt{X}.\lambda\texttt{Y}.(\texttt{R X Y})$$
$$\texttt{closure} \leftarrow_\pi \lambda\texttt{R}.\lambda\texttt{X}.\lambda\texttt{Y}.\exists\texttt{Z}((\texttt{R X Z}) \wedge (\texttt{closure R Z Y}))$$

or even more compactly:

$$\texttt{closure} \leftarrow_\pi (\lambda\texttt{R}.\lambda\texttt{X}.\lambda\texttt{Y}.(\texttt{R X Y})) \bigvee_\pi (\lambda\texttt{R}.\lambda\texttt{X}.\lambda\texttt{Y}.\exists\texttt{Z}((\texttt{R X Z}) \wedge (\texttt{closure R Z Y})))$$

A possible query could be: $\leftarrow$ `(closure R a b)` (which intuitively requests for those binary relations such that the pair $(\texttt{a}, \texttt{b})$ belongs to their transitive closure). In a Prolog-like extended syntax, the above program would have been written as:

```
closure(R, X, Y) :- R(X, Y).
closure(R, X, Y) :- R(X, Z), closure(R, Z, Y).
```

and the corresponding query as $\leftarrow$ `closure(R, a, b).` □

Example 3.7. We define a predicate `ordered` which checks whether its second argument (a list) is ordered according to its first argument (a binary relation). The type of `ordered` is $\pi = (\iota \to \iota \to o) \to \iota \to o$ (notice that the type of a list is also ι since a list is nothing more than a term). In Prolog-like syntax, the program is the following:

```
ordered(R, []).
ordered(R, [X]).
ordered(R, [X,Y|T]) :- R(X, Y), ordered(R, [Y|T]).
```

In the syntax of $\mathcal{H}$ (slightly extended with the standard notation for lists), the above program can be written as follows:

$$\texttt{ordered} \leftarrow_\pi \lambda\texttt{R}.\lambda\texttt{L}.(\texttt{L} \approx \texttt{[]})$$
$$\texttt{ordered} \leftarrow_\pi \lambda\texttt{R}.\lambda\texttt{L}.(\exists\texttt{X}(\texttt{L} \approx \texttt{[X]}))$$
$$\texttt{ordered} \leftarrow_\pi \lambda\texttt{R}.\lambda\texttt{L}.(\exists\texttt{X}\exists\texttt{Y}\exists\texttt{T}((\texttt{L} \approx \texttt{[X,Y|T]}) \wedge (\texttt{R X Y}) \wedge (\texttt{ordered R [Y|T]})))$$

Assume that we have also defined a binary relation `less` which succeeds if its first argument (eg. a natural number) is less than the second one. Then, the query $\leftarrow$ `ordered less [1,4,7,10]` is expected to succeed. On the other hand, the query $\leftarrow$ `ordered R [a,b,c,d]` requests for all binary relations under which the list `[a,b,c,d]` is ordered. As it will become clear in the subsequent sections of the paper, this is a meaningful question which can obtain a reasonable answer. □

4. ALGEBRAIC LATTICES

In order to develop the semantics of $\mathcal{H}$, we first need to introduce certain lattice-theoretic concepts. As it is well-known, the standard semantics of classical (first-order) logic programming, is based on *complete lattices* (see for example [Lloyd 1987]). As we are going to see, the development of the semantics of $\mathcal{H}$ is based on a special class of complete lattices, namely *algebraic lattices* (see for example [Grätzer 1978]). An

algebraic lattice is a complete lattice in which every element can be created by using certain *compact* (intuitively, "simple") elements of the lattice. In our setting, these compact elements will be the ones that the proof procedure will generate in order to answer queries that involve uninstantiated predicate variables. We should mention at this point that algebraic partially ordered sets are widely used in domain theory (see for example [Abramsky and Jung 1994]) and in the semantics of functional languages [Stoy 1977].

We start by introducing some mathematical preliminaries concerning lattice theory. Since the bibliography on partially ordered sets is huge, certain results appear in one form or another in various contexts, and they are often hard to locate in the exact form needed. Propositions 4.7, 4.10 and 4.14 fall into this category; for reasons of completeness, we have included short proofs (or pointers to such proofs) for them. On the other hand, Lemma 4.17 is, to our knowledge, new. We start with some basic definitions:

Definition 4.1. A set P with a binary relation $\sqsubseteq_P$ is called a *partially ordered set* or *poset* if $\sqsubseteq_P$ is reflexive, transitive and antisymmetric.

Usually, the subscript P in $\sqsubseteq_P$ will be omitted when it is obvious from context.

Definition 4.2. Let P be a poset. An element $x \in P$ is called an *upper bound* for a subset $A \subseteq P$, if for every $y \in A$, $y \sqsubseteq x$. If the set of upper bounds of A has a least element, then this element is called the *least upper bound* (or *lub*) of A and is denoted by $\bigsqcup A$. Symmetrically, one can define the notions of *lower bound* and *greatest lower bound* (or *glb*) of A (this last notion denoted by $\bigsqcap A$).

The following proposition (see for example [Abramsky and Jung 1994][Proposition 2.1.4]) will prove useful later in the paper:

PROPOSITION 4.3. *Let P be a poset and let $A, B, (A_i)_{i\in I}$ be subsets of P. Then, the following statements hold (provided the lubs occurring in the formulas exist):*

(1) $A \subseteq B$ *implies* $\bigsqcup A \sqsubseteq \bigsqcup B$.
(2) If $A = \bigcup_{i\in I} A_i$, *then* $\bigsqcup A = \bigsqcup_{i\in I}(\bigsqcup A_i)$.

Definition 4.4. Let P be a poset. A subset A of P is *directed*, if it is nonempty and each pair of elements of A has an upper bound in A.

Definition 4.5. Let P and Q be posets. A function $f : P \to Q$ is called *monotonic* if for all $x, y \in P$ with $x \sqsubseteq_P y$, we have $f(x) \sqsubseteq_Q f(y)$. The set of all monotonic functions from P to Q is denoted by $[P \xrightarrow{m} Q]$.

Notice that monotonicity can be generalized in the obvious way for functions $f : P^n \to Q$, $n > 0$, since P^n is also a poset (where the partial order in this case is defined in a point-wise way).

We are particularly interested in one type of posets, namely *complete lattices*:

Definition 4.6. A poset L in which every subset has a least upper bound and a greatest lower bound, is called a *complete lattice*.

In fact, there is a symmetry here: the existence of all least upper bounds suffices to prove that a poset is indeed a complete lattice, a fact that we will freely use throughout the paper.

PROPOSITION 4.7. *Let P be a poset, L be a complete lattice and let $f : P \times P \to L$ be a monotonic function. Then, $\bigsqcup_{x\in P, y\in P} f(x, y) = \bigsqcup_{x\in P} f(x, x)$.*

PROOF. An easy proof using basic properties of posets (see for example the corresponding proof for *domains* [Tennent 1991][Lemma 5.3, page 92]). □

Definition 4.8. Let L, L' be complete lattices. A function $f : L \to L'$ is called *continuous* if it is monotonic and for every directed subset A of L, we have $f(\bigsqcup A) = \bigsqcup f(A)$.

We will write $\bot_L$ for the greatest lower bound of a complete lattice L (called the *bottom element of* L). A very useful tool in lattice theory, is Kleene's fixpoint theorem:

THEOREM 4.9. *Let L be a complete lattice. Then, every continuous function $f : L \to L$ has a least fixpoint $lfp(f)$ given by $lfp(f) = \bigsqcup_{n<\omega} f^n(\bot_L)$.*

Let A be an arbitrary set and L be a complete lattice. Then, a partial order can be defined on $A \to L$: for all $f, g \in A \to L$, we write $f \sqsubseteq_{A\to L} g$ if for all $a \in A$, $f(a) \sqsubseteq_L g(a)$. We will often use the following proposition:

PROPOSITION 4.10. *Let A be a poset, L be a complete lattice and let $F \subseteq [A \xrightarrow{m} L]$. Then, for all $a \in A$, $(\bigsqcup F)(a) = \bigsqcup_{f\in F} f(a)$ and $(\bigsqcap F)(a) = \bigsqcap_{f\in F} f(a)$. Therefore, $[A \xrightarrow{m} L]$ is a complete lattice.*

PROOF. We give the proof for $\bigsqcup$ (the proof for $\bigsqcap$ is symmetrical). Let $h \in A \to L$ such that $h(a) = \bigsqcup_{f\in F} f(a)$, where the least upper bound is well-defined because L is a complete lattice. Notice that h is obviously an upper bound of F. Now let g be an arbitrary upper bound of F. For each $a \in A$, it holds that $g(a)$ is an upper bound of $\{f(a) \mid f \in F\}$, thus $\bigsqcup_{f\in F} f(a) \sqsubseteq g(a)$ which means that $h \sqsubseteq g$. Therefore, $h = \bigsqcup F$.

It remains to show that h is monotonic. Consider $x, y \in A$ such that $x \sqsubseteq y$. For all $f \in F$ we have $f(x) \sqsubseteq f(y)$ due to the monotonicity of f. Since $\bigsqcup_{f\in F} f(y)$ is an upper bound of $\{f(y) \mid f \in F\}$, it is also an upper bound of $\{f(x) \mid f \in F\}$. Therefore, $\bigsqcup_{f\in F} f(x) \sqsubseteq \bigsqcup_{f\in F} f(y)$ and consequently h is monotonic. □

We will be interested in a certain type of complete lattices in which every element can be "created" by using a set of *compact* (intuitively, "simple") elements of the lattice:

Definition 4.11. Let L be a complete lattice and let $c \in L$. Then c is called *compact* if for every $A \subseteq L$ such that $c \sqsubseteq \bigsqcup A$, there exists finite $A' \subseteq A$ such that $c \sqsubseteq \bigsqcup A'$. The set of all compact elements of L is denoted by $\mathcal{K}(L)$.

We can now define the notion of algebraic lattice (see for example [Grätzer 1978]), which will prove to be the key lattice-theoretic concept applicable to our context.

Definition 4.12. A complete lattice L is called *algebraic* if every element of L is the least upper bound of a set of compact elements of L.

The name "algebraic lattice" is due to G. Birkhoff [Birkhoff 1967] (who did not assume completeness at that time). In the literature, algebraic lattices are also called *compactly generated lattices*.

Example 4.13. Consider the set $L = \{false, true\}$ under the ordering $false \leq true$, $false \leq false$ and $true \leq true$. Then, L is an algebraic lattice with $\mathcal{K}(L) = \{false, true\}$. Actually, every finite complete lattice is algebraic.

Let S be a set. Then, 2^S, the set of all subsets of S, forms a complete lattice under set inclusion. It is easy to see that this is an algebraic lattice whose compact elements are the finite subsets of S. □

Let P be a poset. Given $B \subseteq P$ and $x \in P$, we write $B_{[x]} = \{b \in B \mid b \sqsubseteq_P x\}$. We have the following easy proposition:

PROPOSITION 4.14. *Let L be an algebraic lattice. Then, for every $x \in L$, $x = \bigsqcup \mathcal{K}(L)_{[x]}$.*

PROOF. Obviously it holds that $\bigsqcup \mathcal{K}(L)_{[x]} \sqsubseteq x$. We show that $x \sqsubseteq \bigsqcup \mathcal{K}(L)_{[x]}$. By Definition 4.12, there exists $A \subseteq \mathcal{K}(L)$ such that $x = \bigsqcup A$. Obviously, $A \subseteq \mathcal{K}(L)_{[x]}$. Therefore, by Proposition 4.3, $\bigsqcup A \sqsubseteq \bigsqcup \mathcal{K}(L)_{[x]}$ and consequently $x \sqsubseteq \bigsqcup \mathcal{K}(L)_{[x]}$. □

Given an algebraic lattice L, the set $\mathcal{K}(L)$ will be called *the basis of* L. If additionally, $\mathcal{K}(L)$ is countable, then L will be called an *ω-algebraic lattice*.

In the rest of this section we will define a particular class of algebraic lattices that arise in our semantics of higher-order logic programming. This class will be characterized by Lemma 4.17 that follows. We first need to define the notion of "step functions" (see for example [Abramsky and Jung 1994]) which are used to build the compact elements of our algebraic lattices.

Definition 4.15. Let A be a poset and L be an algebraic lattice. For each $a \in A$ and $c \in \mathcal{K}(L)$, we define the function $(a \searrow c) : A \to L$ as

$$(a \searrow c)(x) = \begin{cases} c, & \text{if } a \sqsubseteq_A x \\ \bot_L, & \text{otherwise} \end{cases}$$

The functions of the above form will be called the *step functions* of $A \to L$.

One can view the step function $(a \searrow c)$ as a c-valued characteristic function (ie., as a function which assigns the c value to all elements of the subset $\{x \in A \mid a \sqsubseteq_A x\}$ of A, and $\bot_L$ to the rest of the elements of A).

Example 4.16. Consider a non-empty set A equipped with the trivial partial order of equality (ie., the partial order that relates every element of A only to itself). Moreover, let $L = \{\mathit{false}, \mathit{true}\}$ (which by Example 4.13 is an algebraic lattice). Then, for every $a \in A$, $(a \searrow \mathit{true})$ is the function that returns true iff its argument is equal to a. In other words $(a \searrow \mathit{true})$ is the characteristic function that corresponds to the singleton set $\{a\}$. On the other hand, for every a, $(a \searrow \mathit{false})$ corresponds to the empty set.

As a second example, assume that A is the set of finite subsets of $\mathbb{N}$ and that $L = \{\mathit{false}, \mathit{true}\}$. Then, for any finite set $a \in A$, $(a \searrow \mathit{true})$ is the function that given any finite set x such that $x \supseteq a$, $(a \searrow \mathit{true})(x) = \mathit{true}$. In other words, $(a \searrow \mathit{true})$ is the characteristic function that corresponds to a set consisting of a and all its (finite) supersets. On the other hand, for every a, $(a \searrow \mathit{false})$ is the function that given any finite set x, $(a \searrow \mathit{false})(x) = \mathit{false}$, ie., it corresponds to the empty set (of sets). □

The following lemma, which we have not seen explicitly stated before, identifies a class of algebraic lattices that will play the central role in the development of the semantics of higher-order logic programming. An important characteristic of these lattices is that they have a simple characterization of their bases. The proof of the lemma is given in the Electronic Appendix, Section A.

LEMMA 4.17. *Let A be a poset and L be an algebraic lattice. Then, $[A \xrightarrow{m} L]$ is an algebraic lattice whose basis is the set of all least upper bounds of finitely many step functions from A to L. If, additionally, A is countable and L is an ω-algebraic lattice then $[A \xrightarrow{m} L]$ is an ω-algebraic lattice.*

We can now outline the reasons why algebraic lattices play such an important role in our context. As we have already mentioned, one of the contributions of this paper is that it allows the treatment of queries with uninstantiated predicate variables. The results of [Wadge 1991] indicate that (due to continuity), if a relation satisfies a predicate, then some "finite representative" of this relation also satisfies it. This gives the idea of defining a semantics which makes these "finite representatives" more explicit.

Intuitively, these finite representatives are the compact elements of an algebraic lattice. From an operational point of view, restricting attention to the compact elements allows us to answer queries with uninstantiated variables: if the set of compact elements is enumerable then we can try them one by one examining in each case whether the query is satisfied.

More formally, since our lattices are algebraic and satisfy the conditions of Lemma 4.17, we have a relatively easy characterization of their sets of compact elements (as suggested by Lemma 4.17). Moreover, as we are going to see, if we restrict attention to Herbrand interpretations (see Section 6), then the lattices that we have to consider are all ω-algebraic and therefore their sets of compact elements are countable. For these lattices it turns out that we can devise an effective procedure for enumerating their compact elements which leads us to an effective proof system for our higher-order language.

5. THE SEMANTICS OF $\mathcal{H}$

The semantics of $\mathcal{H}$ is built upon the notion of algebraic lattice. Recall that an algebraic lattice is a complete lattice L with the additional property that every element x of L is the least upper bound of $\mathcal{K}(L)_{[x]}$.

5.1. The Semantics of Types

Before specifying the semantics of expressions of $\mathcal{H}$ we need to provide the set-theoretic meaning of the types of expressions of $\mathcal{H}$ with respect to a set D (where D is later going to be the domain of our interpretations). The fact that a given type τ denotes a set $[\![\tau]\!]_D$ will mean that an expression of type τ denotes an element of $[\![\tau]\!]_D$. In other words, the semantics of types helps us understand what are the meanings of the expressions of our language. In the following definition we define simultaneously and recursively two things: the semantics $[\![\tau]\!]_D$ of a type τ and the corresponding partial order $\sqsubseteq_\tau$[3].

Definition 5.1. Let D be a non-empty set. Then:

— $[\![\iota]\!]_D = D$, and $\sqsubseteq_\iota$ is the trivial partial order such that $d \sqsubseteq_\iota d$, for all $d \in D$.
— $[\![\iota^n \to \iota]\!]_D = D^n \to D$. A partial order for this case will not be needed.
— $[\![o]\!]_D = \{false, true\}$, and $\sqsubseteq_o$ is the partial order defined by $false \sqsubseteq_o true$, $false \sqsubseteq_o false$ and $true \sqsubseteq_o true$.
— $[\![\iota \to \pi]\!]_D = D \to [\![\pi]\!]_D$, and $\sqsubseteq_{\iota\to\pi}$ is the partial order defined as follows: for all $f, g \in [\![\iota \to \pi]\!]_D$, $f \sqsubseteq_{\iota\to\pi} g$ if and only if $f(d) \sqsubseteq_\pi g(d)$, for all $d \in D$.
— $[\![\pi_1 \to \pi_2]\!]_D = [\mathcal{K}([\![\pi_1]\!]_D) \xrightarrow{m} [\![\pi_2]\!]_D]$, and $\sqsubseteq_{\pi_1\to\pi_2}$ is the partial order defined as follows: for all $f, g \in [\![\pi_1 \to \pi_2]\!]_D$, $f \sqsubseteq_{\pi_1\to\pi_2} g$ if and only if $f(d) \sqsubseteq_{\pi_2} g(d)$, for all $d \in \mathcal{K}([\![\pi_1]\!]_D)$.

It is not immediately obvious that the last case in the above definition is well-defined. More specifically, in order for the quantity $\mathcal{K}([\![\pi_1]\!]_D)$ to make sense, $[\![\pi_1]\!]_D$ must be a complete lattice. This is ensured by the following lemma:

LEMMA 5.2. *Let D be a non-empty set. Then, for every π, $[\![\pi]\!]_D$ is an algebraic lattice (ω-algebraic if D is countable).*

PROOF. The proof is by induction on the structure of π. The basis case is for $\pi = o$ and holds trivially (see Example 4.13). For the induction step, we distinguish two cases. The first case is for $\pi = \iota \to \pi_1$. Then, $[\![\iota \to \pi_1]\!]_D = D \to [\![\pi_1]\!]_D$. Notice now that D is partially ordered by the trivial partial order $\sqsubseteq_\iota$, and it holds that $D \to [\![\pi_1]\!]_D =$

[3]Notice that we are writing $\sqsubseteq_\tau$ instead of the more accurate $\sqsubseteq_{[\![\tau]\!]_D}$. In the following, for brevity reasons we will often use the former (simpler) notation. Similarly, we will often write $\bot_\pi$ instead of $\bot_{[\![\pi]\!]_D}$.

$[D \xrightarrow{m} [\![\pi_1]\!]_D]$ (monotonicity is trivial in this case). By the induction hypothesis and Lemma 4.17 it follows that $[\![\pi]\!]_D$ is an algebraic lattice (ω-algebraic if D is countable). The second case is for $\pi = \pi_1 \to \pi_2$, and the result follows by the induction hypothesis and Lemma 4.17. □

The following definition gives us a convenient shorthand when we want to refer to an object that is either a compact element or a member of the domain D of our interpretations. This shorthand will be used in various places of the paper.

Definition 5.3. Let D be a non-empty set and let ρ be an argument type. Define:

$$\mathcal{F}_D(\rho) = \begin{cases} D, & \text{if } \rho = \iota \\ \mathcal{K}([\![\rho]\!]_D), & \text{otherwise} \end{cases}$$

The set $\mathcal{F}_D(\rho)$ will be called the *set of basic elements of type* ρ (with respect to the set D).

Example 5.4. Consider the type $\iota \to o$ (a first-order predicate with one argument has this type). By Definition 5.1, $[\![\iota \to o]\!]_D$ is the set of all functions from D to $\{false, true\}$ (or equivalently, of all subsets of D).

As a second example, consider the type $(\iota \to o) \to o$. This is the type of a predicate which takes as its only parameter a unary predicate which is first-order; for example, `p` in Example 2.1 has this type. Then, it can be verified using Lemma 4.17 and Example 4.16 that the set $\mathcal{K}([\![\iota \to o]\!]_D)$ is the set of all *finite* functions from D to $\{false, true\}$ (or equivalently, of finite subsets of D). By Definition 5.1, $[\![(\iota \to o) \to o]\!]_D$ is the set of all monotonic functions from finite subsets of D (ie., elements of $\mathcal{K}([\![\iota \to o]\!]_D)$) to $\{false, true\}$. In other words, in the semantics of $\mathcal{H}$, a predicate of type $(\iota \to o) \to o$ will denote a monotonic function from finite subsets of D to $\{false, true\}$. The role that monotonicity plays in this context can be intuitively explained by considering again Example 2.1: if `p` is true of a finite set, then this set must contain both `0` and `s(0)`. But then, `p` will also be true for every superset of this set (since every superset also contains both `0` and `s(0)`). As we are going to see, the meaning of all the higher-order predicates that are defined in a program will possess the monotonicity property. □

Notice now that in our interpretation of types, only monotonicity is required; actually, continuity is not applicable in our interpretation: given a type $\pi_1 \to \pi_2$, it would be meaningless to talk about the continuous functions from $\mathcal{K}([\![\pi_1]\!]_D)$ to $[\![\pi_2]\!]_D$ because $\mathcal{K}([\![\pi_1]\!]_D)$ is not in general a complete lattice[4] as required by the definition of continuity. However, as we are going to see, monotonicity suffices in order to establish that the immediate consequence operator of every program is continuous (Lemma 6.11) and therefore has a least fixed-point.

As a last remark, we should mention that the interpretation of types given in Definition 5.1, does not apply to the inverse implication operator $\leftarrow_\pi$ of $\mathcal{H}$, whose denotation is not monotonic (for example, notice that negation can be implicitly defined with the use of implication). However, since the use of $\leftarrow_\pi$ is not allowed inside positive expressions, the non-monotonicity of $\leftarrow_\pi$ does not create any semantic problems.

5.2. The Semantics of Expressions

We can now proceed to give meaning to the expressions of $\mathcal{H}$. This is performed by first defining the notions of *interpretation* and *state* for $\mathcal{H}$:

[4]To see this, take $\pi_1 = \iota \to o$ and let D be an infinite set. Then, $\mathcal{K}([\![\iota \to o]\!]_D)$ consists of all finite subsets of D and is not a complete lattice (since the least upper bound of a set of finite sets can itself be infinite).

Definition 5.5. An interpretation I of $\mathcal{H}$ consists of:

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

Definition 5.6. Let D be a nonempty set. Then, a *state s of $\mathcal{H}$ over D* is a function that assigns to each argument variable V of type ρ of $\mathcal{H}$ an element $s(\mathsf{V}) \in \mathcal{F}_D(\rho)$.

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

Before we proceed to formally define the semantics of expressions of $\mathcal{H}$, a short discussion on the semantics of *application* is needed. The key technical difficulty we have to confront can be explained by reconsidering Example 2.1 in the more formal context that we have now developed.

Example 5.7. Consider again the program from Example 2.1:

```
p(Q):-Q(0),Q(s(0)).
nat(0).
nat(s(X)):-nat(X).
```

Consider also the query $\leftarrow$ `p(nat)`. The type of `p` is $(\iota \to o) \to o$, while the type of `nat` is $\iota \to o$. Let I be an interpretation with underlying domain D. Then, according to Definition 5.5, $I(\mathtt{p})$ must be a monotonic function from $\mathcal{K}([\![\iota \to o]\!]_D)$ to $\{false, true\}$. Moreover, according to Example 5.4, $\mathcal{K}([\![\iota \to o]\!]_D)$ consists of all the *finite* sets of elements of D. But $I(\mathtt{nat})$ is a member of $[\![\iota \to o]\!]_D$ and can therefore be an infinite set. How can we apply $I(\mathtt{p})$ to $I(\mathtt{nat})$? To overcome this problem, observe that in order for the predicate `p` to succeed for its argument `Q`, it only has to examine a "finite number of facts" about `Q` (namely whether `Q` is true of `0` and `s(0)`). This remark suggests that the meaning of `p(nat)` can be established following a non-standard interpretation of application: we apply $I(\mathtt{p})$ to all the "finite approximations" of $I(\mathtt{nat})$, ie., to all elements of $\mathcal{K}([\![\iota \to o]\!]_D)_{[I(\mathtt{nat})]}$, and then take the least upper bound of the results. Notice that our approach heavily relies on the fact that our semantic domains are algebraic lattices: every element of such a lattice (like $I(\mathtt{nat})$ in our example) is the least upper bound of the compact elements of the lattice that are below it (the finite subsets of $I(\mathtt{nat})$ in our case). □

We can now proceed to present the semantics of $\mathcal{H}$:

Definition 5.8. Let I be an interpretation of $\mathcal{H}$, let D be the domain of I, and let s be a state over D. Then, the semantics of expressions of $\mathcal{H}$ with respect to I and s, is defined as follows:

(1) $[\![\mathsf{false}]\!]_s(I) = false$
(2) $[\![\mathsf{true}]\!]_s(I) = true$
(3) $[\![\mathsf{c}]\!]_s(I) = I(\mathsf{c})$, for every individual constant c
(4) $[\![\mathsf{p}]\!]_s(I) = I(\mathsf{p})$, for every predicate constant p
(5) $[\![\mathsf{V}]\!]_s(I) = s(\mathsf{V})$, for every argument variable 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 f
(7) $[\![(\mathsf{E}_1\mathsf{E}_2)]\!]_s(I) = \bigsqcup_{b \in B}([\![\mathsf{E}_1]\!]_s(I)(b))$, where $B = \mathcal{F}_D(type(\mathsf{E}_2))_{[[\![\mathsf{E}_2]\!]_s(I)]}$
(8) $[\![(\lambda\mathsf{V}.\mathsf{E})]\!]_s(I) = \lambda d.[\![\mathsf{E}]\!]_{s[d/\mathsf{V}]}(I)$, where d ranges over $\mathcal{F}_D(type(\mathsf{V}))$
(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) $[\![(\mathsf{E}_1 \approx \mathsf{E}_2)]\!]_s(I) = \begin{cases} true, & \text{if } [\![\mathsf{E}_1]\!]_s(I) = [\![\mathsf{E}_2]\!]_s(I) \\ false, & \text{otherwise} \end{cases}$

(12) $[\![(\exists \mathsf{V}\, \mathsf{E})]\!]_s(I) = \begin{cases} true, & \text{if there exists } d \in \mathcal{F}_D(type(\mathsf{V})) \text{ such that } [\![\mathsf{E}]\!]_{s[d/\mathsf{V}]}(I) = true \\ false, & \text{otherwise} \end{cases}$

(13) $[\![(\mathsf{p} \leftarrow_\pi \mathsf{E})]\!]_s(I) = \begin{cases} true, & \text{if } [\![\mathsf{E}]\!]_s(I) \sqsubseteq_\pi I(\mathsf{p}) \\ false, & \text{otherwise} \end{cases}$

(14) $[\![(\leftarrow \mathsf{E})]\!]_s(I) = \begin{cases} true, & \text{if } [\![\mathsf{E}]\!]_s(I) = false \\ false, & \text{otherwise} \end{cases}$

For closed expressions E we will often write $[\![\mathsf{E}]\!](I)$ instead of $[\![\mathsf{E}]\!]_s(I)$ (since, in this case, the meaning of E is independent of s).

We need to demonstrate that the semantic valuation function $[\![\cdot]\!]$ assigns to every expression of $\mathcal{H}$ an element of the corresponding semantic domain. More formally, we need to establish that for every interpretation I with domain D, for every state s over D and for all expressions $\mathsf{E} : \rho$, it holds that $[\![\mathsf{E}]\!]_s(I) \in [\![\rho]\!]_D$. In order to prove this, the following definition is needed:

Definition 5.9. Let $\mathcal{S}_{\mathcal{H},D}$ be the set of states of $\mathcal{H}$ over the nonempty set D. We define the following partial order on $\mathcal{S}_{\mathcal{H},D}$: for all $s_1, s_2 \in \mathcal{S}_{\mathcal{H},D}$, $s_1 \sqsubseteq_{\mathcal{S}_{\mathcal{H},D}} s_2$ iff for every argument variable $\mathsf{V} : \rho$ of $\mathcal{H}$, $s_1(\mathsf{V}) \sqsubseteq_\rho s_2(\mathsf{V})$.

The following lemma states that Definition 5.8 assigns to expressions elements of the corresponding semantic domain. Notice that in order to establish this, we must also prove simultaneously that the meaning of positive expressions is monotonic with respect to states.

LEMMA 5.10. *Let $\mathsf{E} : \rho$ be an expression of $\mathcal{H}$ and let D be a nonempty set. Moreover, let s, s_1, s_2 be states over D and let I be an interpretation over D. Then:*

(1) $[\![\mathsf{E}]\!]_s(I) \in [\![\rho]\!]_D$.
(2) If E is positive and $s_1 \sqsubseteq_{\mathcal{S}_{\mathcal{H},D}} s_2$ then $[\![\mathsf{E}]\!]_{s_1}(I) \sqsubseteq_\rho [\![\mathsf{E}]\!]_{s_2}(I)$.

The proof of the lemma is given in the Electronic Appendix, Section B.

We can now define the important notion of a *model* of a set of formulas:

Definition 5.11. Let S be a set of formulas of $\mathcal{H}$ and let I be an interpretation of $\mathcal{H}$. We say that I is a *model* of S if for every $\mathsf{F} \in S$ and for every state s over the domain of I, $[\![\mathsf{F}]\!]_s(I) = true$.

We close this section with the definitions of the notions of unsatisfiability and of logical consequence of a set of formulas.

Definition 5.12. Let S be a set of formulas of $\mathcal{H}$. We say that S is *unsatisfiable* if no interpretation of $\mathcal{H}$ is a model for S.

Definition 5.13. Let S be a set of formulas and F be a formula of $\mathcal{H}$. We say that F is a *logical consequence* of S if, for every interpretation I of $\mathcal{H}$, I is a model of S implies that I is a model of F.

5.3. Discussion on the Semantics

There are certain aspects of the semantics developed in this section, that deserve a further discussion. In this subsection we present at an intuitive level the main reasons that led us to specific decisions regarding the semantics of $\mathcal{H}$.

One first observation is that our interpretation of types in Definition 5.1 is not the standard one: the denotation of $\pi_1 \to \pi_2$ is not the set of *all* relations from the denotation of π_1 to the denotation of π_2. In [Wadge 1991], W. Wadge interpreted types in the standard way and attempted, based on this interpretation, to establish a minimum model property for higher-order logic programs. However, as we explain in detail in Subsection 8, the proof of the minimum model theorem given in [Wadge 1991] based on the standard interpretation, contains a flaw. In other words, the standard interpretation of types does not lead to a minimum Herbrand model semantics for higher-order logic programs. This is one reason that led us to the interpretation of types given in Definition 5.1. Notice that such restrictions in the interpretation of types are quite mainstream in the denotational semantics of programming languages. For example, in the semantics of functional languages, the denotation of a type of the form $\tau_1 \to \tau_2$ is the set of all continuous functions from the denotation of τ_1 to the denotation of τ_2.

A second (related) observation that seems to require further explanation is that in the semantics of types, $[\![\pi_1 \to \pi_2]\!]_D$ is defined as $[\mathcal{K}([\![\pi_1]\!]_D) \stackrel{m}{\to} [\![\pi_2]\!]_D]$ (and not, say, as $[[\![\pi_1]\!]_D \stackrel{m}{\to} [\![\pi_2]\!]_D]$). The reason for this decision has been presented in an intuitive way in Section 2 and we can now re-explain it in a slightly more formal manner. Assume that we want to provide a proof system for programs of $\mathcal{H}$. Consider a goal clause of the form $\leftarrow$`p(R)`, where `p` is defined in our program and has type $(\iota \to o) \to o$. If the type of `p` was interpreted as (say) $[[\![\iota \to o]\!]_D \stackrel{m}{\to} [\![o]\!]_D]$, then the proof system would face the impossible task of examining *all* the uncountably many relations of type $\iota \to o$ for possible solutions. If on the other hand we follow the interpretation of types given in Definition 5.1, then the set $\mathcal{K}([\![\iota \to o]\!]_D)$ is the set of all *finite* relations, which can be enumerated. But then this creates the obvious question: "how do we know that we don't miss anything by restricting attention to the compact elements of a domain?". In our running example, this could be also expressed as follows: "how do we know that by restricting attention to the finite relations we don't miss any interesting infinite ones?". The answer to this question can be given by appealing to our intuitions about programs: when the program predicate `p` succeeds for an infinite relation, then it does so after a finite amount of time and therefore it has only checked a finite amount of information about this relation. In other words, the behavior of `p` can be described by what it does on finite relations.

The above discussion also intuitively explains our non-standard definition of application (see also the discussion in Example 5.7). It also explains why in our semantics the argument variables are permitted to only range over the compact elements of the corresponding domain[5]. For example, consider a predicate constant `p` of type $\pi = (\iota \to o) \to o$, that is defined in a program as `p` $\leftarrow_\pi$ λ`R.(`$\cdots$`)`. Then, according to the interpretation of π, `R` ranges over $\mathcal{K}([\![\iota \to o]\!]_D)$ and this explains why in the case of the semantics of λ-abstraction in Definition 5.8 the bound variable is required to range over compact elements.

Finally, one might wonder why we have insisted on demonstrating that the lattices that show-up in our semantics, are algebraic. In other words, why doesn't it suffice to use the compact elements of the lattice regardless of whether they suffice to describe all elements or not. Intuitively, the algebraicity of our lattices ensures that the new notion of application we introduce is equivalent to the standard one. Alternatively, when we apply a function to an argument, we don't lose anything if we apply the function to all the finite approximations of the argument and take the least upper bound of the results. In other words, the algebraicity ensures that we can decompose the argument

[5]This is evidenced by the fact that a state assigns to a variable of type ρ an element of $\mathcal{F}_D(\rho)$ (and not of $[\![\rho]\!]$); the same happens in the semantics of λ-abstraction and of existential quantification.

of an application without losing any information. A second advantage of the fact that our domains are algebraic lattices is that we know exactly which are the compact elements of our lattice (see Lemma 4.17). In an arbitrary complete lattice we would first have to somehow find and characterize *all* the compact elements of the lattice for all types, and of course prove that they are indeed compact. This would probably require an effort that is comparable to that invested in the present formulation, and wouldn't convey the additional information that the lattices we employ belong to a well-known and broadly studied class of lattices. It is important to stress that algebraic lattices are very familiar structures in domain theory and they have been used extensively since the inception of denotational semantics (see for example [Abramsky and Jung 1994]).

6. MINIMUM HERBRAND MODEL SEMANTICS

Herbrand interpretations constitute a special form of interpretations that have proven to be a cornerstone of first-order logic programming. Analogously, we have:

Definition 6.1. The Herbrand universe $U_{\mathcal{H}}$ of $\mathcal{H}$ is the set of all terms that can be formed out of the individual constants and the function symbols of $\mathcal{H}$.

Definition 6.2. A Herbrand interpretation I of $\mathcal{H}$ is an interpretation such that:

(1) The domain of I is the Herbrand universe $U_{\mathcal{H}}$ of $\mathcal{H}$.
(2) For every individual constant c, $I(\mathsf{c}) = \mathsf{c}$.
(3) For every predicate constant p of type π, $I(\mathsf{p}) \in [\![\pi]\!]_{U_{\mathcal{H}}}$.
(4) For every n-ary function symbol f and for all $\mathsf{t}_1, \ldots, \mathsf{t}_n \in U_{\mathcal{H}}$, $I(\mathsf{f})\,\mathsf{t}_1 \cdots \mathsf{t}_n = \mathsf{f}\,\mathsf{t}_1 \cdots \mathsf{t}_n$.

Notice that the interpretation of terms above is being performed in exactly the same way as in classical (first-order) logic programming. However, the way that predicates are interpreted above is an extension of the way that predicates are treated in classical logic programming (in which the arguments of predicates are just elements of the Herbrand universe).

Since all Herbrand interpretations have the same underlying domain, we will often refer to a "Herbrand state s", meaning a state whose underlying domain is $U_{\mathcal{H}}$. As it is a standard practice in logic programming, we will often refer to an "interpretation of a program P" rather than of the underlying language $\mathcal{H}$. In this case, we will implicitly assume that the set of individual constants and function symbols are those that appear in P. Under this assumption, we will often talk about the "Herbrand universe U_{P} of P.

We should also note that since programs are finite, the Herbrand universe of a program is always a countable set. Then, by Lemma 5.2 we get that for every program P and for every predicate type π, $[\![\pi]\!]_{U_{\mathsf{P}}}$ is an ω-algebraic lattice (ie., it has a countable basis).

The definition of a "Herbrand model" is the usual one:

Definition 6.3. A *Herbrand model* of a program P is a Herbrand interpretation that is a model of P.

We can now proceed to examine properties of Herbrand interpretations. In the following we denote the set of Herbrand interpretations of a program P with $\mathcal{I}_{\mathsf{P}}$.

Definition 6.4. Let P be a program. We define the following partial order on $\mathcal{I}_{\mathsf{P}}$: for all $I, J \in \mathcal{I}_{\mathsf{P}}$, $I \sqsubseteq_{\mathcal{I}_{\mathsf{P}}} J$ iff for every π and for every predicate constant $\mathsf{p} : \pi$ of P, $I(\mathsf{p}) \sqsubseteq_{\pi} J(\mathsf{p})$.

LEMMA 6.5. *Let P be a program and let $\mathcal{I} \subseteq \mathcal{I}_{\mathsf{P}}$. Then, for every predicate p of P, $(\bigsqcup \mathcal{I})(\mathsf{p}) = \bigsqcup_{I \in \mathcal{I}} I(\mathsf{p})$ and $(\bigsqcap \mathcal{I})(\mathsf{p}) = \bigsqcap_{I \in \mathcal{I}} I(\mathsf{p})$. Therefore, $\mathcal{I}_{\mathsf{P}}$ is a complete lattice under $\sqsubseteq_{\mathcal{I}_{\mathsf{P}}}$.*

PROOF. We give the proof for $\bigsqcup$; the proof for $\bigsqcap$ is symmetrical and omitted. Let $J \in \mathcal{I}_{\mathsf{P}}$ such that for every $\mathsf{p} : \pi$ in P, $J(\mathsf{p}) = \bigsqcup_{I \in \mathcal{I}} I(\mathsf{p})$. Notice that $\bigsqcup_{I \in \mathcal{I}} I(\mathsf{p})$ is well-defined since $[\![\pi]\!]_{U_{\mathsf{P}}}$ is a complete lattice. Notice also that J is an upper-bound for $\mathcal{I}$ because for every $I \in \mathcal{I}$, $I \sqsubseteq_{\mathcal{I}_{\mathsf{P}}} J$. Let J' be an arbitrary upper bound of $\mathcal{I}$. Then, for every $\mathsf{p} : \pi$, it holds that $J'(\mathsf{p})$ is an upper bound of $\{I(\mathsf{p}) \mid I \in \mathcal{I}\}$, and therefore $\bigsqcup_{I \in \mathcal{I}} I(\mathsf{p}) \sqsubseteq_{\pi} J'(\mathsf{p})$, which implies that $J \sqsubseteq_{\mathcal{I}_{\mathsf{P}}} J'$. □

In the following we denote with $\bot_{\mathcal{I}_{\mathsf{P}}}$ the greatest lower bound of $\mathcal{I}_{\mathsf{P}}$, ie., the interpretation which for every π, assigns to each predicate $\mathsf{p} : \pi$ of P the element $\bot_{\pi}$.

The properties of monotonicity and continuity of the semantic valuation function will prove vital:

LEMMA 6.6 (MONOTONICITY OF SEMANTICS). *Let* P *be a program and let* $\mathsf{E} : \rho$ *be a positive expression of* P. *Let* I, J *be Herbrand interpretations and* s *be a Herbrand state of* P. *If* $I \sqsubseteq_{\mathcal{I}_{\mathsf{P}}} J$ *then* $[\![\mathsf{E}]\!]_s(I) \sqsubseteq_{\rho} [\![\mathsf{E}]\!]_s(J)$.

The proof of the lemma is given in the Electronic Appendix, Section C.

LEMMA 6.7 (CONTINUITY OF SEMANTICS). *Let* P *be a program and let* E *be a positive expression of* P. *Let* $\mathcal{I}$ *be a directed set of Herbrand interpretations and* s *be a Herbrand state of* P. *Then,* $[\![\mathsf{E}]\!]_s(\bigsqcup \mathcal{I}) = \bigsqcup_{I \in \mathcal{I}} [\![\mathsf{E}]\!]_s(I)$.

The proof of the lemma is given in the Electronic Appendix, Section D.

All the basic properties of first-order logic programming extend naturally to the higher-order case:

THEOREM 6.8 (MODEL INTERSECTION THEOREM). *Let* P *be a program and* $\mathcal{M}$ *be a non-empty set of Herbrand models of* P. *Then,* $\bigsqcap \mathcal{M}$ *is a Herbrand model for* P.

PROOF. By Lemma 6.5, $\bigsqcap \mathcal{M}$ is well-defined. Assume that $\bigsqcap \mathcal{M}$ is not a model for P. Then, there exists a rule $\mathsf{p} \leftarrow_{\pi} \mathsf{E}$ in P and basic elements $b_1, \ldots, b_n$ of the appropriate types such that $(\bigsqcap \mathcal{M})(\mathsf{p})\, b_1 \cdots b_n = \mathit{false}$ while $[\![\mathsf{E}]\!](\bigsqcap \mathcal{M})\, b_1 \cdots b_n = \mathit{true}$. Since for every $M \in \mathcal{M}$ we have $\bigsqcap \mathcal{M} \sqsubseteq M$, using Lemma 6.6 we conclude that for all $M \in \mathcal{M}$, $[\![\mathsf{E}]\!](M)\, b_1 \cdots b_n = 1$. Moreover, since $(\bigsqcap \mathcal{M})(\mathsf{p})\; b_1 \cdots b_n = \mathit{false}$, by Lemma 6.5 we get that $(\bigsqcap \mathcal{M}(\mathsf{p}))\; b_1 \cdots b_n = \mathit{false}$. By Proposition 4.10 we conclude that for some $M \in \mathcal{M}$, $M(\mathsf{p})\; b_1 \cdots b_n = \mathit{false}$. But then there exists $M \in \mathcal{M}$ that does not satisfy the rule $\mathsf{p} \leftarrow_{\pi} \mathsf{E}$, and therefore is not a model of P (contradiction). □

It is straightforward to check that every higher-order program P has at least one Herbrand model I, namely the one which for every predicate constant p of P and for all basic elements $b_1, \ldots, b_n$ of the appropriate types, $I(\mathsf{p})\, b_1 \cdots b_n = \mathit{true}$. Notice that this model generalizes the familiar idea of "Herbrand Base" that is used in the theory of first-order logic programming.

Since the set of models of a higher-order logic program is non-empty, the intersection (glb) of all Herbrand models is well-defined and by the above theorem is a model of the program. We will denote this model by M_{P}.

Definition 6.9. Let P be a program. The mapping $T_{\mathsf{P}} : \mathcal{I}_{\mathsf{P}} \to \mathcal{I}_{\mathsf{P}}$ is defined as follows for every $\mathsf{p} : \pi$ in P and for every $I \in \mathcal{I}_{\mathsf{P}}$:

$$T_{\mathsf{P}}(I)(\mathsf{p}) = \bigsqcup_{(\mathsf{p} \leftarrow_{\pi} \mathsf{E}) \in \mathsf{P}} [\![\mathsf{E}]\!](I)$$

The mapping T_{P} will be called the *immediate consequence operator* for P.

The fact that T_{P} is well-defined is verified by the following lemma:

LEMMA 6.10. *Let* P *be a program and let* $\mathsf{p} : \pi$ *be a predicate constant of* P. *Then, for every* $I \in \mathcal{I}_\mathsf{P}$, $T_\mathsf{P}(I)(\mathsf{p}) \in [\![\pi]\!]_{U_\mathsf{P}}$.

PROOF. The result follows directly by the definition of T_P, Lemma 5.10 and the fact that $[\![\pi]\!]_{U_\mathsf{P}}$ is a complete lattice. □

The key property of T_P is that it is continuous:

LEMMA 6.11. *Let* P *be a program. Then the mapping* T_P *is continuous.*

PROOF. Straightforward using Lemma 6.7. □

The following property of T_P generalizes the corresponding well-known property from first-order logic programming:

LEMMA 6.12. *Let* P *be a program and let* $I \in \mathcal{I}_\mathsf{P}$. *Then* I *is a model of* P *if and only if* $T_\mathsf{P}(I) \sqsubseteq_{\mathcal{I}_\mathsf{P}} I$.

PROOF. An interpretation $I \in \mathcal{I}_\mathsf{P}$ is a model of P iff $[\![\mathsf{E}]\!](I) \sqsubseteq_\pi I(\mathsf{p})$ for every clause $\mathsf{p} \leftarrow_\pi \mathsf{E}$ in P iff $\bigsqcup_{(\mathsf{p} \leftarrow_\pi \mathsf{E}) \in \mathsf{P}} [\![\mathsf{E}]\!](I) \sqsubseteq_\pi I(\mathsf{p})$ iff $T_\mathsf{P}(I)(\mathsf{p}) \sqsubseteq_\pi I(\mathsf{p})$. □

Define now the following sequence of interpretations:

$$\begin{array}{lcl} T_\mathsf{P} \uparrow 0 & = & \bot_{\mathcal{I}_\mathsf{P}} \\ T_\mathsf{P} \uparrow (n+1) & = & T_\mathsf{P}(T_\mathsf{P} \uparrow n) \\ T_\mathsf{P} \uparrow \omega & = & \bigsqcup\{T_\mathsf{P} \uparrow n \mid n < \omega\} \end{array}$$

We have the following theorem (which is entirely analogous to the one for the first-order case):

THEOREM 6.13. *Let* P *be a program. Then* $M_\mathsf{P} = lfp(T_\mathsf{P}) = T_\mathsf{P} \uparrow \omega$.

PROOF. Using exactly the same reasoning as in the first-order case (see for example the corresponding proof in [Lloyd 1987]). □

Closing this section, we feel it is important to discuss certain desirable characteristics of the minimum Herbrand model semantics. First of all, in the proposed semantics, the denotation of a user-defined predicate is literally a set of tuples of compact elements and this fact directly renders the extensionality rule applicable. For example, if p is of type $(\iota \to o) \to o$, then the meaning of p is the set of all finite sets for which p is true. Consider now another predicate of the same type, say q, which is true of the same finite sets as p. Then, obviously, p and q are equal as sets. From a programming point of view, extensionality means that p can be replaced by q in any program without changing the output of the program. For example, if r is of type $((\iota \to o) \to o) \to o$, then, by the semantics of application, r(p) has the same meaning as r(q).

In a slightly different direction, the above discussion implies that under the proposed semantics, we can make changes to a predicate's definition and as long as our transformations preserve the predicate's extension, the enclosing program retains its meaning. Semantics preserving transformations are a vital part of any programming paradigm; the proposed semantics is actually a formal tool for the justification of such transformations. Therefore, all the benefits of extensionality that were highlighted in [Wadge 1991][pages 299-300], continue to hold under the proposed semantics.

Apart from extensionality, the proposed semantics also possesses the potential of being useful in proving properties of programs. In the theory of programming languages, many properties of programs can be proved using induction on the approximations of the least fixed-point of the program (see for example [Stoy 1977][pages 209-217]). Due to the inductive definition of $T_\mathsf{P} \uparrow \omega$, similar proofs can also be performed in our setting. In conclusion, using the proposed semantics for the understanding and analysis

of higher-order logic programs appears to be one of the most promising next steps in our work.

7. A PROOF SYSTEM

In this section we propose a sound and complete proof system for the programs of $\mathcal{H}$. One important aspect we initially have to resolve, is how to represent basic elements (see Definition 5.3) in our source language. In the following subsection we introduce a class of positive expressions, namely *basic expressions*, which are the syntactic analogues of basic elements. Basic expressions will be used in order the formalization of the notion of *answer* (to a given query) as-well-as in the development of the SLD-resolution proof system.

7.1. Basic Expressions

As we have already seen, basic elements have played an important role in the development of the semantics of our higher-order logic programming language. In order to devise a sound and complete proof system for the programs of $\mathcal{H}$, we first need to find a syntactic representation for basic elements. Since the definition of basic elements uses the operator $\sqsubseteq$ (see Lemma 4.17, Definition 4.15 and Definition 5.3), it is not immediately obvious how one can construct a positive expression whose meaning coincides with a given basic element. Basic expressions introduced below, solve this apparent difficulty:

Definition 7.1. The set of *basic expressions of* $\mathcal{H}$ is recursively defined as follows. Every expression of $\mathcal{H}$ of type ι is a basic expression of type ι. Every predicate variable of $\mathcal{H}$ of type π is a basic expression of type π. The propositional constants false and true are basic expressions of type o. A non-empty finite union of expressions each one of which has the following form, is a basic expression of type $\rho_1 \to \cdots \to \rho_n \to o$ (where $\mathsf{V}_1 : \rho_1, \ldots, \mathsf{V}_n : \rho_n$):

(1) $\lambda\mathsf{V}_1.\cdots\lambda\mathsf{V}_n.\mathsf{false}$
(2) $\lambda\mathsf{V}_1.\cdots\lambda\mathsf{V}_n.(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n)$, where each A_i is either
   (a) $(\mathsf{V}_i \approx \mathsf{B}_i)$, if $\mathsf{V}_i : \iota$ and $\mathsf{B}_i : \iota$ is a basic expression where $\mathsf{V}_j \notin FV(\mathsf{B}_i)$ for all j, or
   (b) the constant true or V_i, if $\mathsf{V}_i : o$, or
   (c) the constant true or $\mathsf{V}_i(\mathsf{B}_{11})\cdots(\mathsf{B}_{1r}) \wedge \cdots \wedge \mathsf{V}_i(\mathsf{B}_{m1})\cdots(\mathsf{B}_{mr})$, $m > 0$, if $type(\mathsf{V}_i) = \rho'_1 \to \cdots \to \rho'_r \to o$ and for all k, l, B_{kl} is a basic expression with $type(\mathsf{B}_{kl}) = \rho'_l$ and for all j, $\mathsf{V}_j \notin FV(\mathsf{B}_{kl})$.

The B_i and B_{kl} above will be called the *basic subexpressions* of B.

The following example illustrates the ideas behind the above definition.

Example 7.2. We consider various cases of the above definition:

— The terms `a`, `f(a,b)`, `X` and `f(X,h(Y))`, are basic expressions of type ι.
— Assume `X` : ρ. Then, λ`X.false` is a basic expression of type $\rho \to o$. Intuitively, it corresponds to the basic element $\bot_{\rho\to o}$.
— Assume `X` : ι. Then, λ`X.(X`$\approx$`a)` is a basic expression of type $\iota \to o$. Intuitively, it corresponds to the basic element (`a` $\searrow$ $true$) or more simply to the finite set {`a`}.
— Assume `X` : ι and `Y` : ι. Then, λ`X.`λ`Y.(X`$\approx$`a)`$\wedge$`(Y`$\approx$`b)` is a basic expression of type $\iota \to \iota \to o$. Intuitively, it corresponds to the basic element (`a` $\searrow$ (`b` $\searrow$ $true$)) or more simply to the singleton binary relation {(`a`, `b`)}.
— Assume `X` : ι. Then, `(`λ`X.(X`$\approx$`a))` $\bigvee_{\iota\to o}$`(`λ`X.(X`$\approx$`b))` is a basic expression of type $\iota \to o$. It corresponds to the basic element $\bigsqcup\{$(`a` $\searrow$ $true$), (`b` $\searrow$ $true$)$\}$, or more simply to the finite set {`a`, `b`}.

— Assume $\mathtt{Q} : \iota \to o$. Then, $\lambda\mathtt{Q}.(\mathtt{Q(a)} \wedge \mathtt{Q(b)})$is a basic expression of type $(\iota \to o) \to o$. Intuitively, it corresponds to the basic element $(\bigsqcup\{(\mathtt{a} \searrow \mathit{true}), (\mathtt{b} \searrow \mathit{true})\}) \searrow \mathit{true}$. More simply, it corresponds to the set of all finite sets that contain both a and b.

□

The proof system that will be developed later in this section, relies on a special form of basic expressions:

Definition 7.3. The set of *basic templates of* $\mathcal{H}$ is the subset of the set of basic expressions of $\mathcal{H}$ defined as follows:

— The propositional constants false and true are basic templates.
— Every non-empty finite union of basic expressions (of the form presented in items 1 and 2 of Definition 7.1) in which all the basic subexpressions involved are *distinct* variables, is a basic template.

The distinct variables mentioned above, will be called *template variables*.

Example 7.4. Assume in the following expressions that $\mathtt{X}, \mathtt{Y}, \mathtt{Z}, \mathtt{W} : \iota$, $\mathtt{Q}, \mathtt{Q}_1, \mathtt{Q}_2 : \iota \to o$ and $\mathtt{R} : ((\iota \to o) \to o) \to o$. The expression $\lambda\mathtt{X}.(\mathtt{X} \approx \mathtt{Z})$ is a basic template of type $\iota \to o$. The expression $\lambda\mathtt{X}.\lambda\mathtt{Y}.(\mathtt{X} \approx \mathtt{Z}) \wedge (\mathtt{Y} \approx \mathtt{W})$ is a basic template of type $\iota \to \iota \to o$; the template variables in this case are Z and W. The expression $\lambda\mathtt{Q}.(\mathtt{Q(Z)} \wedge \mathtt{Q(W)})$is a basic template of type $(\iota \to o) \to o$ with template variables Z and W. The expression $\lambda\mathtt{R}.(\mathtt{R}(\mathtt{Q}_1) \wedge \mathtt{R}(\mathtt{Q}_2))$ is a basic template of type $((\iota \to o) \to o) \to o$ with template variables $\mathtt{Q}_1$ and $\mathtt{Q}_2$. □

Notice from the above example that the structure of basic templates is in general much simpler than that of basic expressions (due to the fact that a template variable can represent an arbitrary basic expression of the same type). For this reason, basic templates are much simpler to enumerate than arbitrary basic expressions.

The following two lemmas establish the connections between basic elements and basic expressions.

LEMMA 7.5. *For every basic expression* $\mathsf{B} : \rho$*, for every Herbrand interpretation* I *of* $\mathcal{H}$*, and for every Herbrand state* s*,* $[\![\mathsf{B}]\!]_s(I) \in \mathcal{F}_{U_\mathcal{H}}(\rho)$.

PROOF. The proof is by induction on the type of B. The basis case is for basic expressions of type ι and o and holds trivially. We demonstrate that the lemma holds for basic expressions of type $\rho = \rho_1 \to \cdots \to \rho_n \to o$, assuming that it holds for all basic expressions that have simpler types than ρ. If the basic expression is a predicate variable, the result is immediate; otherwise, we have to distinguish the following cases:

Case 1: $\mathsf{B} = \lambda\mathsf{V}_1.\cdots\lambda\mathsf{V}_n.\mathsf{false}$. Then, the corresponding basic element in $\mathcal{F}_{U_\mathcal{H}}(\rho)$ is the bottom element of type $\rho_1 \to \cdots \to \rho_n \to o$ (ie., $\bot_{\rho_1 \to \cdots \to \rho_n \to o}$).

Case 2: $\mathsf{B} = \lambda\mathsf{V}_1.\cdots\lambda\mathsf{V}_n.(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n)$. Then, the corresponding basic element is the element $b_1 \searrow (b_2 \searrow \cdots \searrow (b_n \searrow \mathit{true})\cdots)$, where the b_i are defined as follows:

— If $\mathsf{V}_i : \iota$, then by Definition 7.1, $\mathsf{A}_i = (\mathsf{V}_i \approx \mathsf{B}_i)$. In this case, $b_i = [\![\mathsf{B}_i]\!]_s(I)$.
— If $\mathsf{V}_i : o$ then A_i is either equal to true or to V_i; in the former case $b_i = \mathit{false}$ and in the latter case $b_i = \mathit{true}$.
— If V_i is of any other type then A_i is either equal to true or to $\mathsf{V}_i(\mathsf{B}_{11})\cdots(\mathsf{B}_{1r}) \wedge \cdots \wedge \mathsf{V}_i(\mathsf{B}_{m1})\cdots(\mathsf{B}_{mr})$, where $m > 0$. In the former case it is $b_i = \bot_{\rho_i}$; in the latter case $b_i = \bigsqcup_{1 \le j \le m}([\![\mathsf{B}_{j1}]\!]_s(I) \searrow ([\![\mathsf{B}_{j2}]\!]_s(I) \searrow \cdots \searrow ([\![\mathsf{B}_{jr}]\!]_s(I) \searrow \mathit{true})\cdots))$.

Case 3: B is a finite union of lambda abstractions. Then, for each term of the finite union we can create (as above) a basic element. By taking the finite union of these elements, we create the basic element that corresponds to B.

It can be easily verified that for every basic expression B, $[\![\mathsf{B}]\!]_s(I)$ coincides with the corresponding basic element defined as above. □

The converse of the above lemma holds, as the following lemma demonstrates.

LEMMA 7.6. *Let ρ be any argument type and let $b \in \mathcal{F}_{U_\mathcal{H}}(\rho)$. Then, there exists a closed basic expression $\mathsf{B} : \rho$ such that for every Herbrand interpretation I, $[\![\mathsf{B}]\!](I) = b$.*

PROOF. The proof is by induction on the structure of argument types. The basis case is for argument types ι and o, and holds trivially. We demonstrate that the lemma holds for type $\rho = \rho_1 \to \cdots \to \rho_n \to o$, assuming that it holds for all subtypes of ρ. Assume now that b is a basic element of type ρ, consisting of a finite union of step functions.

If the union is empty, then $\mathsf{B} = \lambda\mathsf{V}_1.\cdots\lambda\mathsf{V}_n.\mathsf{false}$. Assume now that the union is non-empty. Then, the basic expression corresponding to b will simply be the union of the basic expressions corresponding to the step functions that comprise b.

Let $b_1 \searrow (b_2 \searrow \cdots \searrow (b_n \searrow \mathit{true})\cdots)$ be one of the step functions that constitute b. We create the basic expression: $\mathsf{B} = \lambda\mathsf{V}_1.\cdots\lambda\mathsf{V}_n.(\mathsf{A}_1 \wedge \cdots \mathsf{A}_n)$ where each A_i can be created as follows:

— If b_i is of type ι and $b_i = \mathsf{t} \in U_\mathcal{H}$, then $\mathsf{A}_i = (\mathsf{V}_i \approx \mathsf{t})$.
— If b_i is of type o and $b_i = \mathit{false}$, then $\mathsf{A}_i = \mathsf{true}$.
— If b_i is of type o and $b_i = \mathit{true}$, then $\mathsf{A}_i = \mathsf{V}_i$.
— Otherwise, b_i is a finite union of $m > 0$ basic elements of the form $b_{j1} \searrow (b_{j2} \searrow \cdots \searrow (b_{jr} \searrow \mathit{true})\cdots)$, $1 \le j \le m$. Then, $\mathsf{A}_i = \mathsf{V}_i(\mathsf{B}_{11})\cdots(\mathsf{B}_{1r}) \wedge \cdots \wedge \mathsf{V}_i(\mathsf{B}_{m1})\cdots(\mathsf{B}_{mr})$, where $\mathsf{B}_{j1}, \ldots, \mathsf{B}_{jr}$ are the expressions that correspond (by the induction hypothesis) to $b_{j1}, \ldots, b_{jr}$.

It is easy to verify that the resulting basic expression B satisfies $[\![\mathsf{B}]\!](I) = b$. □

The above two lemmas suggest that basic expressions are the syntactic analogues of basic elements.

7.2. Substitutions and Unifiers

Substitutions are vital in the development of the proof system for $\mathcal{H}$:

Definition 7.7. A *substitution* θ is a finite set of the form $\{\mathsf{V}_1/\mathsf{E}_1, \ldots, \mathsf{V}_n/\mathsf{E}_n\}$, where the V_i's are different argument variables of $\mathcal{H}$ and each E_i is a positive expression of $\mathcal{H}$ having the same type as 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 E_i are basic expressions. A substitution is called *zero-order*, if $type(\mathsf{V}_i) = \iota$, for all $i \in \{1, \ldots, n\}$ (notice that every zero-order substitution is also basic). The substitution corresponding to the empty set will be called the *identity substitution* and will be denoted by ϵ.

We are now ready to define what it means to apply a substitution θ to an expression E. Such definitions are usually complicated by the fact that one has to often rename the bound variable before applying θ to the body of a lambda abstraction. In order to simplify matters, we follow the simple approach suggested in [Barendregt 1984][pages 26-27], which consists of the following two conventions:

— **The α-congruence convention**: Expressions that are *α-congruent* will be considered identical (expression E_1 is α-congruent with expression E_2 if E_2 results from E_1

by a series of changes of bound variables). For example, λ`Q.Q(a)` is α-congruent to λ`R.R(a)`.

— **The variable convention**: If expressions $\mathsf{E}_1, \ldots, \mathsf{E}_n$ occur in a certain mathematical context (eg., definition, proof), then in these expressions all bound variables are chosen to be different from the free variables.

Using the variable convention, we have the following simple definition:

Definition 7.8. Let θ be a substitution and let E be a positive expression. Then, $\mathsf{E}\theta$ is an expression obtained from E as follows:

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

Notice that in the case of lambda abstraction (and similarly in the case of existential quantification), it is not needed to say "provided $\mathsf{V} \notin FV(range(\theta))$ and $\mathsf{V} \notin dom(\theta)$". By the variable convention this is the case.

Definition 7.9. Let $\theta = \{\mathsf{V}_1/\mathsf{E}_1, \ldots, \mathsf{V}_m/\mathsf{E}_m\}$ and $\sigma = \{\mathsf{V}'_1/\mathsf{E}'_1, \ldots, \mathsf{V}'_n/\mathsf{E}'_n\}$ be substitutions. Then the composition $\theta\sigma$ of θ and σ 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 $\mathsf{V}'_j \in \{\mathsf{V}_1, \ldots, \mathsf{V}_m\}$.

The following proposition is easy to establish:

PROPOSITION 7.10. *Let θ, σ and γ be substitutions. Then:*

(1) $\theta\epsilon = \epsilon\theta = \theta$.
(2) For all positive expressions E, $(\mathsf{E}\theta)\sigma = \mathsf{E}(\theta\sigma)$.
(3) $(\theta\sigma)\gamma = \theta(\sigma\gamma)$.

We will now define the notions of "unifier" and "most general unifier", which in our case are the same as in the case of classical first-order logic programming (notice that in the following definition, the expressions to be unified are of type ι and the substitutions involved are all zero-order). Notice that in the proposed proof system there is no higher-order unification process involved.

Definition 7.11. Let S be a set of terms of $\mathcal{H}$ (ie., expressions of type ι). A zero-order substitution θ 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 θ will be called a *most general unifier of* S (denoted by $mgu(S)$), if for every unifier σ of the expressions in S, there exists a zero-order substitution γ such that $\sigma = \theta\gamma$.

We now have the following *Substitution Lemma* (see for example [Tennent 1991] for a corresponding lemma in the case of functional programming). The Substitution Lemma shows that given a basic substitution θ, the meaning of $\mathsf{E}\theta$ is that of E in a

certain state definable from θ. The lemma will be later used in the proof of soundness of the proposed proof system.

LEMMA 7.12 (SUBSTITUTION LEMMA). *Let I be an interpretation of $\mathcal{H}$ and let s be a state over the domain of I. Let θ be a basic substitution and* E *be a positive 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.*

PROOF. By structural induction on E. □

The following lemmas, that also involve the notion of substitution, can be easily demonstrated and will prove useful in the sequel.

LEMMA 7.13. *Let $\theta_1, \ldots, \theta_n$ be basic substitutions. Then, $\theta_1 \cdots \theta_n$ is also a basic substitution.*

PROOF. By induction on n and using Definitions 7.1 and 7.9. □

LEMMA 7.14. *Let I be an interpretation of $\mathcal{H}$ and let s be a state over the domain of I. Let $\lambda\mathsf{V}.\mathsf{E}_1$ and E_2 be positive expressions of type $\rho \to \pi$ and ρ respectively. Then,* $[\![(\lambda\mathsf{V}.\mathsf{E}_1)\mathsf{E}_2]\!]_s(I) = [\![\mathsf{E}_1\{\mathsf{V}/\mathsf{E}_2\}]\!]_s(I)$.

PROOF. By structural induction on E_1. □

LEMMA 7.15. *Let I be a Herbrand interpretation of $\mathcal{H}$ and let s be a Herbrand state. Let* E *be a positive expression. Then, there exists a basic substitution θ such that* $[\![\mathsf{E}]\!]_s(I) = [\![\mathsf{E}\theta]\!]_{s'}(I)$ *for every Herbrand state s'.*

PROOF. Define θ such that if $\mathsf{V} \in FV(\mathsf{E})$, $\theta(\mathsf{V}) = \mathsf{B}$, where B is a closed basic expression such that $[\![\mathsf{B}]\!](I) = s(\mathsf{V})$ (the existence of such a B is ensured by Lemma 7.6). The lemma follows by a structural induction on E. □

It is important to note that in the rest of the paper, the substitutions that we will use will be *basic* ones (unless otherwise stated). Actually, the only place where a non-basic substitution will be needed, is when we perform a *β-reduction* step (see for example the rule for λ in the forthcoming Definition 7.18).

7.3. SLD-Resolution

We now proceed to define the notions of *answer* and *correct answer*.

Definition 7.16. Let P be a program and G be a goal. An *answer* for $\mathsf{P} \cup \{\mathsf{G}\}$ is a basic substitution for (certain of the) free variables of G.

Definition 7.17. Let P be a program, $\mathsf{G} = \leftarrow \mathsf{A}$ be a goal clause and let θ be an answer for $\mathsf{P} \cup \{\mathsf{G}\}$. We say that θ is *a correct answer* for $\mathsf{P} \cup \{\mathsf{G}\}$ if for every model M of P and for every state s over the domain of M, $[\![\mathsf{A}\theta]\!]_s(M) = true$.

Definition 7.18. Let P be a program and let $\mathsf{G} = \leftarrow \mathsf{A}$ and $\mathsf{G}' = \leftarrow \mathsf{A}'$ be goal clauses. We say that A' is derived in one step from A using basic substitution θ (or equivalently that G' is derived in one step from G using θ), and we denote this fact by $\mathsf{A} \xrightarrow{\theta} \mathsf{A}'$ (respectively, $\mathsf{G} \xrightarrow{\theta} \mathsf{G}'$), if one of the following conditions applies:

(1) $\mathsf{p}\ \mathsf{E}_1 \cdots \mathsf{E}_n \xrightarrow{\epsilon} \mathsf{E}\ \mathsf{E}_1 \cdots \mathsf{E}_n$, where $\mathsf{p} \leftarrow_\pi \mathsf{E}$ is a rule in P.
(2) $\mathsf{Q}\ \mathsf{E}_1 \cdots \mathsf{E}_n \xrightarrow{\theta} (\mathsf{Q}\ \mathsf{E}_1 \cdots \mathsf{E}_n)\theta$, where $\theta = \{\mathsf{Q}/\mathsf{B}_t\}$ and B_t a basic template.
(3) $(\lambda\mathsf{V}.\mathsf{E})\ \mathsf{E}_1 \cdots \mathsf{E}_n \xrightarrow{\epsilon} (\mathsf{E}\{\mathsf{V}/\mathsf{E}_1\})\ \mathsf{E}_2 \cdots \mathsf{E}_n$.
(4) $(\mathsf{E}' \bigvee_\pi \mathsf{E}''')\ \mathsf{E}_1 \cdots \mathsf{E}_n \xrightarrow{\epsilon} \mathsf{E}'\ \mathsf{E}_1 \cdots \mathsf{E}_n$.
(5) $(\mathsf{E}' \bigvee_\pi \mathsf{E}''')\ \mathsf{E}_1 \cdots \mathsf{E}_n \xrightarrow{\epsilon} \mathsf{E}''\ \mathsf{E}_1 \cdots \mathsf{E}_n$.

(6) $(\mathsf{E}' \bigwedge_{\pi} \mathsf{E}'')\ \mathsf{E}_1 \cdots \mathsf{E}_n \xrightarrow{\epsilon} (\mathsf{E}'\ \mathsf{E}_1 \cdots \mathsf{E}_n) \wedge (\mathsf{E}''\ \mathsf{E}_1 \cdots \mathsf{E}_n)$, where $\pi \neq o$.
(7) $(\mathsf{E}_1 \wedge \mathsf{E}_2) \xrightarrow{\theta} (\mathsf{E}_1' \wedge (\mathsf{E}_2\theta))$, if $\mathsf{E}_1 \xrightarrow{\theta} \mathsf{E}_1'$.
(8) $(\mathsf{E}_1 \wedge \mathsf{E}_2) \xrightarrow{\theta} ((\mathsf{E}_1\theta) \wedge \mathsf{E}_2')$, if $\mathsf{E}_2 \xrightarrow{\theta} \mathsf{E}_2'$.
(9) $(\mathsf{true} \wedge \mathsf{E}) \xrightarrow{\epsilon} \mathsf{E}$
(10) $(\mathsf{E} \wedge \mathsf{true}) \xrightarrow{\epsilon} \mathsf{E}$
(11) $(\mathsf{E}_1 \approx \mathsf{E}_2) \xrightarrow{\theta} \mathsf{true}$, where θ is an mgu of E_1 and E_2.
(12) $(\exists \mathsf{V}\, \mathsf{E}) \xrightarrow{\epsilon} \mathsf{E}$

Moreover, we write $\mathsf{A} \xrightarrow{\theta}\!\!\!\!\rightarrow \mathsf{A}'$ if $\mathsf{A} = \mathsf{A}_0 \xrightarrow{\theta_1} \mathsf{A}_1 \xrightarrow{\theta_2} \cdots \xrightarrow{\theta_n} \mathsf{A}_n = \mathsf{A}'$, $n \geq 1$, where $\theta = \theta_1 \cdots \theta_n$ (and similarly for $\mathsf{G} \xrightarrow{\theta}\!\!\!\!\rightarrow \mathsf{G}'$).

Definition 7.19. Let P be a program and G be a goal. An *SLD-derivation* of $\mathsf{P} \cup \{\mathsf{G}\}$ is a (finite or infinite) sequence $\mathsf{G}_0 = \mathsf{G}, \mathsf{G}_1, \ldots$ of goals and a sequence $\theta_1, \theta_2, \ldots$ of basic substitutions such that:

(1) each G_{i+1} is derived in one step from G_i using θ_{i+1}, and
(2) for all i, if $\theta_i = \{\mathsf{V}/\mathsf{B}_t\}$ where B_t is a basic template, then the free variables of B_t are disjoint from all the variables that have already appeared in the derivation up to G_{i-1}.

Definition 7.20. Let P be a program and G be a goal. Assume that $\mathsf{P} \cup \{\mathsf{G}\}$ has a finite SLD-derivation $\mathsf{G}_0 = \mathsf{G}, \mathsf{G}_1, \ldots, \mathsf{G}_n$ with basic substitutions $\theta_1, \ldots, \theta_n$, such that $\mathsf{G}_n = \square$. Then, we will say that $\mathsf{P} \cup \{\mathsf{G}\}$ has an *SLD-refutation of length* n *using basic substitution* $\theta = \theta_1 \cdots \theta_n$.

Definition 7.21. Let P be a program, G be a goal and assume that $\mathsf{P} \cup \{\mathsf{G}\}$ has an SLD-refutation using basic substitution θ. Then, a *computed answer* σ for $\mathsf{P} \cup \{\mathsf{G}\}$ is the basic substitution obtained by restricting θ to the free variables of G.

Example 7.22. Consider the program of Example 3.6. An SLD-refutation of the goal $\leftarrow$ `closure Q a b` is given below (where we have omitted certain simple steps involving lambda abstractions):

`closure Q a b`	$\theta_1 = \epsilon$
`(`λ`R.`λ`X.`λ`Y.(R X Y)) Q a b`	$\theta_2 = \epsilon$
`Q a b`	$\theta_3 = \{\mathtt{Q}/(\lambda\mathtt{X}.\lambda\mathtt{Y}.(\mathtt{X} \approx \mathtt{X}_0) \wedge (\mathtt{Y} \approx \mathtt{Y}_0))\}$
`(`λ`X.`λ`Y.(X`$\approx$`X`$_0$`)`$\wedge$`(Y`$\approx$`Y`$_0$`)) a b`	$\theta_4 = \epsilon$
`(a`$\approx$`X`$_0$`)`$\wedge$`(b`$\approx$`Y`$_0$`)`	$\theta_5 = \{\mathtt{X}_0/\mathtt{a}\}$
$\square\wedge$`(b`$\approx$`Y`$_0$`)`	$\theta_6 = \epsilon$
`(b`$\approx$`Y`$_0$`)`	$\theta_7 = \{\mathtt{Y}_0/\mathtt{b}\}$
$\square$	

If we restrict the composition $\theta_1 \cdots \theta_7$ to the free variables of the goal, we get the computed answer $\sigma_1 = \{\mathtt{Q}/\lambda\mathtt{X}.\lambda\mathtt{Y}.(\mathtt{X} \approx \mathtt{a}) \wedge (\mathtt{Y} \approx \mathtt{b})\}$. Intuitively, σ_1 assigns to `Q` the relation $\{(\mathtt{a}, \mathtt{b})\}$ (for which the query is obviously true). Notice that by substituting `Q` with different basic templates, one can get answers that are "similar" to the above one, such as for example $\{(\mathtt{a}, \mathtt{b}), (\mathtt{Z1}, \mathtt{Z2})\}$ or $\{(\mathtt{a}, \mathtt{b}), (\mathtt{Z1}, \mathtt{Z2}), (\mathtt{Z3}, \mathtt{Z4})\}$, and so on. Answers of this type are in some sense "represented" by the answer $\{(\mathtt{a}, \mathtt{b})\}$. Actually, one can easily optimize the proof system so as to avoid enumerating such superfluous answers (see the discussion in Section 9).

However, there exist other answers to our original query that are genuinely different from $\{(\mathtt{a}, \mathtt{b})\}$ and can be obtained by making different clause choices. For example, another answer to our query is σ_2 =

$\{\mathtt{Q}/(\lambda\mathtt{X}.\lambda\mathtt{Y}.(\mathtt{X}\approx\mathtt{a})\wedge(\mathtt{Y}\approx\mathtt{Z}))\bigvee_{\pi}(\lambda\mathtt{X}.\lambda\mathtt{Y}.(\mathtt{X}\approx\mathtt{Z})\wedge(\mathtt{Y}\approx\mathtt{b}))\}$, which corresponds to the relations of the form $\{(\mathtt{a},\mathtt{Z}),(\mathtt{Z},\mathtt{b})\}$, for every $\mathtt{Z}$ in the Herbrand universe. Similarly, one can get the answer $\{(\mathtt{a},\mathtt{Z1}),(\mathtt{Z1},\mathtt{Z2}),(\mathtt{Z2},\mathtt{b})\}$, and so on.

In other words, we observe that by performing different choices in the selection of a basic template for $\mathtt{Q}$ and making an appropriate use of the two rules of the program for `closure`, we get an infinite (but countable) number of computed answers to our original query. □

7.4. Soundness of SLD-resolution

In this subsection we establish the soundness of the SLD-resolution proof system. The following lemmas are very useful in the proof of the soundness theorem:

LEMMA 7.23. *Let* P *be a program, let* I *be an interpretation of* P *and let* s *be a state over the domain of* I*. Let* E_1 *and* E_2 *be positive expressions of type* $\rho\to\pi$ *and let* E *be a positive expression of type* ρ*. If* $[\![\mathsf{E}_1]\!]_s(I)\sqsubseteq_{\rho\to\pi}[\![\mathsf{E}_2]\!]_s(I)$*, then* $[\![(\mathsf{E}_1\,\mathsf{E})]\!]_s(I)\sqsubseteq_{\pi}[\![(\mathsf{E}_2\,\mathsf{E})]\!]_s(I)$*.*

PROOF. Straightforward using the definition of application. □

LEMMA 7.24. *Let* P *be a program, let* $\mathsf{G}=\!\leftarrow\mathsf{A}$ *and* $\mathsf{G}'=\!\leftarrow\mathsf{A}'$ *be goals and let* θ *be a basic substitution such that* $\mathsf{A}\xrightarrow{\theta}\mathsf{A}'$*. Then, for every model* M *of* P *and for every state* s *over the domain of* M*, it holds that* $[\![\mathsf{A}\theta]\!]_s(M)\sqsupseteq[\![\mathsf{A}']\!]_s(M)$*.*

PROOF. First, observe that in all cases A is of the form $\mathsf{E}\,\mathsf{E}_1\cdots\mathsf{E}_k$, $k\geq 0$, where E is an expression of predicate type. We perform a structural induction on E.

Induction Basis: We distinguish three cases, namely $\mathsf{E}=(\mathsf{E}'\approx\mathsf{E}'')$, $\mathsf{E}=\mathsf{p}$ and $\mathsf{E}=\mathsf{Q}$. For the first case it suffices to show that $[\![(\mathsf{E}'\approx\mathsf{E}'')\theta]\!]_s(M)\sqsupseteq[\![\mathsf{true}]\!]_s(M)$, where θ is an mgu of E' and E''. This holds trivially since both sides are equal to $true$. For the second case it suffices to show that $[\![(\mathsf{p}\,\mathsf{E}_1\cdots\mathsf{E}_k)\theta]\!]_s(M)\sqsupseteq[\![(\mathsf{E}_\mathsf{p}\,\mathsf{E}_1\cdots\mathsf{E}_k)\theta]\!]_s(M)$, where $\theta=\epsilon$ and $\mathsf{p}\leftarrow_\pi\mathsf{E}_\mathsf{p}$ is a clause in P. This follows easily by the fact that M is a model of P and using Lemma 7.23. The third case is trivial.

Induction Step: We examine the two most interesting cases (the rest are straightforward):

Case 1: $\mathsf{E}=(\lambda\mathsf{V}.\mathsf{E}')$. In this case θ is the empty substitution, and therefore it suffices to show that $[\![(\lambda\mathsf{V}.\mathsf{E}')\,\mathsf{E}_1\cdots\mathsf{E}_k]\!]_s(M)\sqsupseteq[\![\mathsf{E}'\{\mathsf{V}/\mathsf{E}_1\}\,\mathsf{E}_2\cdots\mathsf{E}_k]\!]_s(M)$. By Lemma 7.14 we have that $[\![(\lambda\mathsf{V}.\mathsf{E}')\,\mathsf{E}_1]\!]_s(M)=[\![\mathsf{E}'\{\mathsf{V}/\mathsf{E}_1\}]\!]_s(M)$, and the result follows by Lemma 7.23.

Case 2: $\mathsf{E}=(\mathsf{E}'\wedge\mathsf{E}'')$. Moreover, assume that $\mathsf{E}'\xrightarrow{\theta}\mathsf{E}'_1$. Then, $(\mathsf{E}'\wedge\mathsf{E}'')$ derives in one step the expression $(\mathsf{E}'_1\wedge(\mathsf{E}''\theta))$. It suffices to show that $[\![(\mathsf{E}'\wedge\mathsf{E}'')\theta]\!]_s(M)\sqsupseteq[\![\mathsf{E}'_1\wedge(\mathsf{E}''\theta)]\!]_s(M)$, or equivalently that $[\![(\mathsf{E}'\theta)\wedge(\mathsf{E}''\theta)]\!]_s(M)\sqsupseteq[\![\mathsf{E}'_1\wedge(\mathsf{E}''\theta)]\!]_s(M)$. But this holds since by the induction hypothesis we have that $[\![\mathsf{E}'\theta]\!]_s(M)\sqsupseteq[\![\mathsf{E}'_1]\!]_s(M)$. □

LEMMA 7.25. *Let* 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 an SLD-refutation of length* n *using basic substitutions* $\theta_1,\ldots,\theta_n$*. Then, for every model* M *of* 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 7.24, Lemma 7.12 and induction on n. □

THEOREM 7.26 (SOUNDNESS). *Let* P *be a program and* $\mathsf{G}=\!\leftarrow\mathsf{A}$ *be a goal. Then, every computed answer for* $\mathsf{P}\cup\{\mathsf{G}\}$ *is a correct answer for* $\mathsf{P}\cup\{\mathsf{G}\}$*.*

PROOF. The result is a direct consequence of Lemma 7.25 for $\mathsf{G}_n=\Box$ (ie., for $\mathsf{G}_n=\!\leftarrow\mathsf{true}$). □

7.5. Completeness of SLD-resolution

In order to establish the completeness of the proposed SLD-resolution, we need to first demonstrate a result that is analogous to the *lifting lemma* of the first-order case (see [Lloyd 1987]). We first state (and prove in the appendix) a more technical lemma, which has as a special case the desired lifting lemma.

In the rest of this subsection, whenever we refer to a "substitution" we mean a "basic substitution".

LEMMA 7.27. *Let* P *be a program,* G *be a goal and let* θ *be a substitution. Suppose that there exists an SLD-refutation of* $\mathsf{P} \cup \{\mathsf{G}\theta\}$ *using substitution* σ*. Then, there exists an SLD-refutation of* $\mathsf{P} \cup \{\mathsf{G}\}$ *using a substitution* δ*, where for some substitution* γ *it holds that* $\delta\gamma \supseteq \theta\sigma$ *and* $dom(\delta\gamma - \theta\sigma)$ *is a (possibly empty) set of template variables that are introduced during the refutation of* $\mathsf{P} \cup \{\mathsf{G}\}$.

The proof of the above lemma is by induction on the length of the SLD-refutation of $\mathsf{P} \cup \{\mathsf{G}\theta\}$, and is given in the Electronic Appendix, Section E.

LEMMA 7.28 (LIFTING LEMMA). *Let* P *be a program,* G *be a goal and let* θ *be a substitution. Suppose that there exists an SLD-refutation of* $\mathsf{P} \cup \{\mathsf{G}\theta\}$ *using substitution* σ*. Then, there exists an SLD-refutation of* $\mathsf{P} \cup \{\mathsf{G}\}$ *using a substitution* δ*, where for some substitution* γ *it holds that* $\mathsf{G}\delta\gamma = \mathsf{G}\theta\sigma$.

PROOF. By Lemma 7.27, $\delta\gamma$ and $\theta\sigma$ differ only in template variables that are introduced during the refutation. By the second restriction mentioned in Definition 7.19, these variables are different from the variables in the goal G. Therefore, $\delta\gamma$ and $\theta\sigma$ agree on the expressions they assign to the free variables of G. □

Notice that the above lifting lemma differs slightly from the corresponding lemma for classical logic programming, where we actually have the equality $\delta\gamma = \theta\sigma$. This difference is due to the existence of template variables in the higher-order resolution proof system. Of course, if we restrict the higher-order proof system to apply to first-order logic programs, then it behaves like classical SLD-resolution and the usual lifting lemma holds.

Example 7.29. Consider any program P of our higher-order language and consider the goal clause $\mathsf{G} = \leftarrow \mathtt{R(Z)}$, where $\mathtt{Z}$ is of type ι and $\mathtt{R}$ of type $\iota \to o$. Let $\theta = \{\mathtt{R}/\lambda\mathtt{X}.(\mathtt{X} \approx \mathtt{a}), \mathtt{Z}/\mathtt{a}\}$. Then, $\mathsf{G}\theta = \leftarrow (\lambda\mathtt{X}.(\mathtt{X} \approx \mathtt{a}))(\mathtt{a})$. We have the following SLD-refutation:

$$(\lambda\mathtt{X}.(\mathtt{X} \approx \mathtt{a}))(\mathtt{a}) \xrightarrow{\epsilon} (\mathtt{a} \approx \mathtt{a}) \xrightarrow{\epsilon} \mathsf{true}$$

Therefore, $\mathsf{G}\theta$ has an SLD-refutation with substitution $\sigma = \epsilon$. On the other hand, we have the following SLD-refutation of G:

$$\mathtt{R(Z)} \overset{\{\mathtt{R}/\lambda\mathtt{X}.(\mathtt{X} \approx \mathtt{X}_0)\}}{\rightarrow} (\lambda\mathtt{X}.(\mathtt{X} \approx \mathtt{X}_0))(\mathtt{Z}) \xrightarrow{\epsilon} (\mathtt{Z} \approx \mathtt{X}_0) \overset{\{\mathtt{X}_0/\mathtt{Z}\}}{\rightarrow} \mathsf{true}$$

Therefore, G has an SLD-refutation with substitution δ which is equal to the composition of the substitutions $\{\mathtt{R}/\lambda\mathtt{X}.(\mathtt{X} \approx \mathtt{X}_0)\}$, ϵ and $\{\mathtt{X}_0/\mathtt{Z}\}$, ie., $\delta = \{\mathtt{R}/\lambda\mathtt{X}.(\mathtt{X} \approx \mathtt{Z}), \mathtt{X}_0/\mathtt{Z}\}$. Let $\gamma = \{\mathtt{Z}/\mathtt{a}\}$. Then, $\delta\gamma = \{\mathtt{R}/\lambda\mathtt{X}.(\mathtt{X} \approx \mathtt{a}), \mathtt{X}_0/\mathtt{a}, \mathtt{Z}/\mathtt{a}\}$ while $\theta = \{\mathtt{R}/\lambda\mathtt{X}.(\mathtt{X} \approx \mathtt{a}), \mathtt{Z}/\mathtt{a}\}$. We see that $\delta\gamma \supseteq \theta\sigma$ and $dom(\delta\gamma - \theta\sigma) = \{\mathtt{X}_0\}$ (which is a template variable). Moreover, it holds $\mathsf{G}\theta\sigma = \mathsf{G}\delta\gamma$. □

Before we derive the first completeness result, we need certain definitions and lemmas.

Definition 7.30. Let P be a program and let E be a positive expression or a goal clause. We define S_{E} to be the set of all expressions that can be obtained from E by

substituting zero or more occurrences of every predicate constant p in E with the expression $\mathsf{E}_1 \bigvee_\pi \cdots \bigvee_\pi \mathsf{E}_k$, where $\mathsf{p} \leftarrow_\pi \mathsf{E}_i$ are all the clauses[6] for p in P. Moreover, $\widehat{\mathsf{E}} \in S_{\mathsf{E}}$ is the expression obtained from E by substituting *every* predicate symbol occurrence with the corresponding expression.

LEMMA 7.31. *Let* P *be a program,* E *be a positive expression or a goal clause,* I *be a Herbrand interpretation of* P *and let* s *be a Herbrand state. Then,* $[\![\mathsf{E}]\!]_s(T_{\mathsf{P}}(I)) = [\![\widehat{\mathsf{E}}]\!]_s(I)$.

PROOF. The proof is by structural induction on E. Assume that E is a positive expression (the proof for the case of goal clause is similar). For the induction basis we need to consider the cases where E is an argument variable V, an individual constant c, a propositional constant (false, true), or a predicate constant p. Except for the last one, all other cases are straightforward because the meaning of E is independent of $T_{\mathsf{P}}(I)$ and I. For the last case assume that $\mathsf{E}_1, \ldots, \mathsf{E}_k$ are all the bodies of the rules defining p in P. By definition of the T_{P} operator, it holds that $[\![\mathsf{p}]\!]_s(T_{\mathsf{P}}(I)) = \bigsqcup_{(\mathsf{p} \leftarrow_\pi \mathsf{E}_i) \in \mathsf{P}} [\![\mathsf{E}_i]\!]_s(I)$. Moreover, $[\![\widehat{\mathsf{E}}]\!]_s(I) = [\![\mathsf{E}_1 \bigvee_\pi \cdots \bigvee_\pi \mathsf{E}_k]\!]_s(I) = \bigsqcup_{(\mathsf{p} \leftarrow_\pi \mathsf{E}_i) \in \mathsf{P}} [\![\mathsf{E}_i]\!]_s(I)$. This completes the basis case. For the induction step, all cases are immediate. □

LEMMA 7.32. *Let* P *be a program,* G,G' *be goals and* $\mathsf{G}' \in S_{\mathsf{G}}$. *If* $\mathsf{G}' \xrightarrow{\theta} \mathsf{H}'$ *then* $\mathsf{G} \overset{\theta}{\twoheadrightarrow} \mathsf{H}$, *where* $\mathsf{H}' \in S_{\mathsf{H}}$.

PROOF. The proof is by induction on the number m of top-level subexpressions of the goal G that are connected with the logical constant $\wedge$. The basis case is for $m = 1$, ie., it applies to goal clauses G that do not contain a top-level $\wedge$. Assume that $\mathsf{G} = \leftarrow \mathsf{A}$. The cases we need to examine for A for the induction basis are the following: $(\mathsf{p}\ \mathsf{A}_1 \cdots \mathsf{A}_n)$, $(\mathsf{Q}\ \mathsf{A}_1, \ldots \mathsf{A}_n)$, $((\lambda \mathsf{V}.\mathsf{A}')\ \mathsf{A}_1 \cdots \mathsf{A}_n)$, $((\mathsf{A}' \bigvee_\pi \mathsf{A}'')\ \mathsf{A}_1 \cdots \mathsf{A}_n)$, $((\mathsf{A}' \bigwedge_\pi \mathsf{A}'')\ \mathsf{A}_1 \cdots \mathsf{A}_n)$, $(\mathsf{A}_1 \approx \mathsf{A}_2)$, and $(\exists \mathsf{V}\, \mathsf{A})$. The only non-trivial case is $\mathsf{A} = (\mathsf{p}\ \mathsf{A}_1 \cdots \mathsf{A}_n)$, which we demonstrate. Assume that p is defined in P with a set of k rules with right-hand sides $\mathsf{E}_1, \ldots, \mathsf{E}_k$. Let $\mathsf{E}_{\mathsf{p}} = \mathsf{E}_1 \bigvee_\pi \cdots \bigvee_\pi \mathsf{E}_k$. Since $\mathsf{G} = \leftarrow (\mathsf{p}\, \mathsf{A}_1 \cdots \mathsf{A}_n)$, we have that $\mathsf{G}' = \leftarrow (\mathsf{A}'\, \mathsf{A}'_1 \cdots \mathsf{A}'_n)$, with $\mathsf{A}' \in S_{\mathsf{p}}, \mathsf{A}'_1 \in S_{\mathsf{A}_1}, \ldots, \mathsf{A}'_n \in S_{\mathsf{A}_n}$. We distinguish three cases for A':

— $\mathsf{A}' = \mathsf{p}$. Then $\mathsf{G}' \xrightarrow{\epsilon} \mathsf{H}'$, where $\mathsf{H}' = \leftarrow (\mathsf{E}_j\, \mathsf{A}'_1 \cdots \mathsf{A}'_n)$ for some j. We also have that $\mathsf{G} \xrightarrow{\epsilon} \mathsf{H}$, where $\mathsf{H} = \leftarrow (\mathsf{E}_j\, \mathsf{A}_1 \cdots \mathsf{A}_n)$. Obviously, it holds that $\mathsf{H}' \in S_{\mathsf{H}}$.
— $\mathsf{A}' = \mathsf{E}_{\mathsf{p}}$ and E_{p} contains more than one disjunct. Then $\mathsf{G}' \xrightarrow{\epsilon} \mathsf{H}'$, where $\mathsf{H}' = \leftarrow (\mathsf{E}_j\, \mathsf{A}'_1 \cdots \mathsf{A}'_n)$. We also have that $\mathsf{G} \xrightarrow{\epsilon} \mathsf{H}$, where $\mathsf{H} = \leftarrow (\mathsf{E}_j\, \mathsf{A}_1 \cdots \mathsf{A}_n)$. Again, it holds that $\mathsf{H}' \in S_{\mathsf{H}}$.
— $\mathsf{A}' = \mathsf{E}_{\mathsf{p}}$ and E_{p} contains exactly one disjunct. Then this disjunct must be a lambda abstraction of the form $(\lambda \mathsf{V}.\mathsf{A}'')$. This implies that $\mathsf{G}' = \leftarrow ((\lambda \mathsf{V}.\mathsf{A}'')\, \mathsf{A}'_1 \cdots \mathsf{A}'_n)$ and $\mathsf{G}' \xrightarrow{\epsilon} \mathsf{H}'$, where $\mathsf{H}' = \leftarrow (\mathsf{A}''\{\mathsf{V}/\mathsf{A}'_1\}\, \mathsf{A}'_2 \cdots \mathsf{A}'_n)$. On the other hand, $\mathsf{G} \xrightarrow{\epsilon} \mathsf{H}_1$, where $\mathsf{H}_1 = \leftarrow ((\lambda \mathsf{V}.\mathsf{A}'')\, \mathsf{A}_1 \cdots \mathsf{A}_n)$, and $\mathsf{H}_1 \xrightarrow{\epsilon} \mathsf{H}$, where $\mathsf{H} = \leftarrow (\mathsf{A}''\{\mathsf{V}/\mathsf{A}_1\}\, \mathsf{A}_2 \cdots \mathsf{A}_n)$. Therefore, $\mathsf{G} \overset{\epsilon}{\twoheadrightarrow} \mathsf{H}$ where $\mathsf{H}' \in S_{\mathsf{H}}$.

The above completes the proof for the basis case. For the induction step, the goal must be of the form $\mathsf{G} = \leftarrow (\mathsf{A}_1 \wedge \mathsf{A}_2)$. Then, $\mathsf{G}' = \leftarrow (\mathsf{A}'_1 \wedge \mathsf{A}'_2)$ where $\mathsf{A}'_1 \in S_{\mathsf{A}_1}$ and $\mathsf{A}'_2 \in S_{\mathsf{A}_2}$. Since $\mathsf{G}' \xrightarrow{\theta} \mathsf{H}'$, we conclude without loss of generality that $\mathsf{A}'_1 \xrightarrow{\theta} \mathsf{H}'_1$ and $\mathsf{H}' = \leftarrow (\mathsf{H}'_1 \wedge \mathsf{A}'_2\theta)$. By the induction hypothesis, since $\mathsf{A}'_1 \xrightarrow{\theta} \mathsf{H}'_1$, we get that $\mathsf{A}_1 \overset{\theta}{\twoheadrightarrow} \mathsf{H}_1$, where $\mathsf{H}'_1 \in S_{\mathsf{H}_1}$. But

[6]We may assume without loss of generality that each predicate symbol p that is used in P, has a definition in P: if no such definition exists, we can add to the program the clause $\mathsf{p} \leftarrow_\pi \mathsf{E}$, where E is a basic expression corresponding to the basic element $\bot_\pi$.

then this easily implies that $(\mathsf{A}_1 \wedge \mathsf{A}_2) \overset{\theta}{\twoheadrightarrow} (\mathsf{H}_1 \wedge \mathsf{A}_2\theta)$, ie., $\mathsf{G} \overset{\theta}{\twoheadrightarrow} \mathsf{H}$, where $\mathsf{H}' \in S_{\mathsf{H}}$. This completes the proof for the induction step and the lemma. □

LEMMA 7.33. *Let* P *be a program,* G,G' *be goals and* $\mathsf{G}' \in S_{\mathsf{G}}$. *If there exists an SLD-refutation for* $\mathsf{P} \cup \{\mathsf{G}'\}$ *using substitution* θ, *then there also exists an SLD-refutation for* $\mathsf{P} \cup \{\mathsf{G}\}$ *using the same substitution* θ.

PROOF. The proof is by induction on the length n of the refutation of $\mathsf{P} \cup \{\mathsf{G}'\}$. The induction basis is for $n = 1$ and includes the following cases for G: $(\mathsf{true} \wedge \mathsf{true})$, $(\mathsf{true} \vee \mathsf{E}_1)$, $(\mathsf{E}_1 \vee \mathsf{true})$, $(\mathsf{E}_1 \approx \mathsf{E}_2)$, $((\lambda \mathsf{V}.\mathsf{true})\,\mathsf{E})$ and $(\exists \mathsf{V}\,\mathsf{true})$. It can be easily verified that the lemma holds for all these cases.

Suppose now that the result holds for $n - 1$. We demonstrate that it also holds for n. Let $\mathsf{G}' = \mathsf{G}'_0, \mathsf{G}'_1, \ldots, \mathsf{G}'_n$ be the derived goals of the SLD-refutation of G' using the sequence of substitutions $\theta_1, \ldots, \theta_n$. Since $\mathsf{G}' \overset{\theta_1}{\to} \mathsf{G}'_1$, by Lemma 7.32 there exists a goal G_1 such that $\mathsf{G} \overset{\theta_1}{\twoheadrightarrow} \mathsf{G}_1$ and $\mathsf{G}'_1 \in S_{\mathsf{G}_1}$. By the induction hypothesis, $\mathsf{P} \cup \{\mathsf{G}_1\}$ has an SLD-refutation using $\theta_2 \cdots \theta_n$. It follows that $\mathsf{P} \cup \{\mathsf{G}\}$ also has an SLD-refutation using $\theta = \theta_1 \cdots \theta_n$. □

COROLLARY 7.34. *Let* P *be a program and* G *be a goal. If there exists an SLD-refutation for* $\mathsf{P} \cup \{\widehat{\mathsf{G}}\}$ *using substitution* θ, *then there also exists an SLD-refutation for* $\mathsf{P} \cup \{\mathsf{G}\}$ *using the same substitution* θ. □

LEMMA 7.35. *Let* P *be a program and* $\mathsf{G} = \leftarrow \mathsf{A}$ *be a goal such that* $[\![\mathsf{A}]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = \mathit{true}$ *for all Herbrand states* s. *Then, there exists an SLD-refutation for* $\mathsf{P} \cup \{\mathsf{G}\}$ *with computed answer equal to the identity substitution.*

The proof of the lemma can be found in the Electronic Appendix, Section F.

As in the first-order case, we have various forms of completeness. We can now prove the analogue of a theorem due to Apt and van Emden (see [Apt 1990][Lemma 3.17] or [Lloyd 1987][Theorem 8.3]).

THEOREM 7.36. *Let* P *be a program,* $\mathsf{G} = \leftarrow \mathsf{A}$ *be a goal and assume that* $[\![\mathsf{A}]\!]_s(M_\mathsf{P}) = \mathit{true}$ *for all Herbrand states* s. *Then, there exists an SLD-refutation for* $\mathsf{P} \cup \{\mathsf{G}\}$ *with computed answer equal to the identity substitution.*

PROOF. We prove by induction on n that if $[\![\mathsf{A}]\!]_s(T_\mathsf{P} \uparrow n) = \mathit{true}$ for all Herbrand states s, then $\mathsf{P} \cup \{\mathsf{G}\}$ has an SLD-refutation with computed answer equal to the identity substitution. For $n = 0$ the proof is a direct consequence of Lemma 7.35.

Now suppose that the result holds for $n - 1$. For the induction step assume that $[\![\mathsf{A}]\!]_s(T_\mathsf{P} \uparrow n) = \mathit{true}$ for all s. By Lemma 7.31, $[\![\widehat{\mathsf{A}}]\!]_s(T_\mathsf{P} \uparrow (n-1)) = \mathit{true}$. By the induction hypothesis there exists an SLD-refutation for $\mathsf{P} \cup \{\widehat{\mathsf{G}}\}$ with computed answer equal to the identity substitution. Let θ be the composition of the substitutions that are used during the SLD-refutation of $\mathsf{P} \cup \{\widehat{\mathsf{G}}\}$. By Corollary 7.34, $\mathsf{P} \cup \{\mathsf{G}\}$ also has an SLD-refutation using the same substitution θ. The restriction of θ to the free variables of G is equal to the restriction of θ to the free variables of $\widehat{\mathsf{G}}$ which is equal to the empty substitution. Therefore, $\mathsf{P} \cup \{\mathsf{G}\}$ has an SLD-refutation with computed answer equal to the identity substitution. □

The following theorem generalizes a result of Hill [Hill 1974] (see also [Apt 1990][Theorem 3.15]):

THEOREM 7.37. *Let* P *be a program and* $\mathsf{G} = \leftarrow \mathsf{A}$ *be a goal. Suppose that* $\mathsf{P} \cup \{\mathsf{G}\}$ *is unsatisfiable. Then, there exists an SLD-refutation of* $\mathsf{P} \cup \{\mathsf{G}\}$.

PROOF. Since $\mathsf{P} \cup \{\mathsf{G}\}$ is unsatisfiable and since M_{P} is a model of P, we conclude that $[\![\mathsf{G}]\!]_s(M_{\mathsf{P}}) = \mathit{false}$, for some state s. Therefore, $[\![\mathsf{A}]\!]_s(M_{\mathsf{P}}) = \mathit{true}$. By Lemma 7.15 we can construct a substitution θ such that $[\![\mathsf{A}\theta]\!]_{s'}(M_{\mathsf{P}}) = \mathit{true}$ for all Herbrand states s'. By Theorem 7.36, there exists an SLD-refutation for $\mathsf{P} \cup \{\mathsf{G}\theta\}$. By Lemma 7.28 there exists an SLD-refutation for $\mathsf{P} \cup \{\mathsf{G}\}$. □

Finally, the following theorem is a generalization of Clark's theorem [Clark 1979] (see also the more accessible [Apt 1990][Theorem 3.18]) for the higher-order case:

THEOREM 7.38 (COMPLETENESS). *Let* P *be a program and* $\mathsf{G} = \leftarrow \mathsf{A}$ *be a goal. For every correct answer* θ *for* $\mathsf{P} \cup \{\mathsf{G}\}$*, there exists an SLD-refutation for* $\mathsf{P} \cup \{\mathsf{G}\}$ *with computed answer* δ *and a substitution* γ *such that* $\mathsf{G}\theta = \mathsf{G}\delta\gamma$.

PROOF. Since θ is a correct answer for $\mathsf{P} \cup \{\leftarrow \mathsf{A}\}$, it follows that $[\![\mathsf{A}\theta]\!]_s(M_{\mathsf{P}}) = \mathit{true}$ for all Herbrand states s. By Theorem 7.36, $\mathsf{P} \cup \{\mathsf{G}\theta\}$ has an SLD-refutation with computed answer equal to the identity substitution. This means that if σ is the composition of the substitutions used in the refutation of $\mathsf{P} \cup \{\mathsf{G}\theta\}$, then $\mathsf{G}\theta\sigma = \mathsf{G}\theta$. By Lemma 7.28 there exists an SLD-refutation for $\mathsf{P} \cup \{\mathsf{G}\}$ using substitution δ' such that for some substitution γ, $\mathsf{G}\delta'\gamma = \mathsf{G}\theta\sigma$. Let δ be δ' restricted to the variables in G. Then, it also holds that $\mathsf{G}\delta'\gamma = \mathsf{G}\delta\gamma$, and therefore $\mathsf{G}\delta\gamma = \mathsf{G}\theta\sigma = \mathsf{G}\theta$. □

8. RELATED WORK

In this section we present various approaches to higher-order logic programming that are directly or indirectly connected to the work reported in this paper.

8.1. The Origins

The basic notions underlying our work can be traced back to certain classical ideas from computability theory as-well-as to ideas from domain theory. Starting back in 1959, Kleene and Kreisel introduced in [Kleene 1959] and [Kreisel 1959] respectively, what are now known as the *Kleene-Kreisel continuous functionals*. The key idea in both of these works was to construct a hierarchy of functionals where the behavior of functional Φ on input Ψ can be determined through finite "approximations" to Φ and Ψ. There is an obvious conceptual link to our work in which the behavior of a program predicate is described in terms of its behavior on the compact elements of the domain corresponding to its argument type.

The work of Kleene and Kreisel has very close connections to the development of *domain theory* in Theoretical Computer Science (see for example [Normann 2006] for a more detailed discussion on the relationship between the two areas). Roughly speaking, domain theory was developed in order to solve recursive definitions (of functions, data-structures, etc.) that often appear in Computer Science. The problem we have considered in this paper is exactly of this type: we are given a set of recursively defined higher-order predicates and we are seeking the least solution of this set of definitions. Our solution to the problem has actually used many mainstream tools from domain theory (eg., algebraic lattices, least fixed-point theorem, and so on). What is new in our approach is that our constructions are performed in a logic programming setting (and not in a functional or an imperative one); moreover, our language has the novel characteristic of supporting uninstantiated higher-order variables, a fact that led us to a novel way of interpreting types and to a new, non-standard interpretation of application.

8.2. Extensional Higher-Order Logic Programming

Work on extensional higher-order logic programming is rather limited. Apart from the results of [Wadge 1991][7], the only other work that has come to our attention is that of M. Bezem [Bezem 1999; 2001], who considers higher-order logic programming languages with syntax similar to that of [Wadge 1991].

Wadge's approach:. In [Wadge 1991], W. W. Wadge identifies a fragment of higher-order logic programming that is argued to have an extensional semantics. In this fragment, (i) predicate constants cannot appear as parameters in the heads of clauses, (ii) predicate variables that appear in the body of a clause must also appear in its head, and (iii) predicate variables cannot appear in goal clauses. Wadge provides a standard semantics for this language [Wadge 1991][page 292] in which the denotations of types are not restricted, ie., the denotation of $\pi_1 \to \pi_2$ is the set of *all* functions from the denotation of π_1 to the denotation of π_2. Under this semantics, it is claimed in [Wadge 1991] that every program has a unique minimum model that assigns to predicates continuous relations. The ideas in [Wadge 1991] are indeed intriguing, but the main argument contains a flaw (which however can be corrected by altering the semantics). More specifically, a careful examination of [Wadge 1991] reveals that if the denotations of types are not somehow restricted, then the main theorem of the paper (the second theorem in page 296 of [Wadge 1991]) does not hold.

Example 8.1. We give an example which demonstrates that if in the fragment of [Wadge 1991] one allows the universally quantified variables to range over arbitrary relations, then the intended model of the program will not in general be continuous. This contradicts the second theorem in page 296 of [Wadge 1991] which states that the minimum Herbrand model of a higher-order program is continuous. Actually, the construction of the minimum Herbrand model itself sketched in [Wadge 1991] does not seem to work either, if the domains contain arbitrary relations.

Consider the program:

```
apply(R,S):-R(S).
```

where the type of `apply` is $((\iota \to o) \to o) \to (\iota \to o) \to o$. The intended model of the program assigns to `apply` the relation $apply$ which contains all pairs (r, s) such that $s \in r$. We demonstrate that $apply$ is not continuous if we allow the implicit universal quantifiers to range over all relations of the corresponding types.

Consider the relation $nat = \{0, 1, 2, \ldots\}$ of type $\iota \to o$ and the relation $allnat = \{nat\}$ of type $(\iota \to o) \to o$. It is easy to see that $allnat$ is not continuous. Consider now the following directed subset of the domain of $apply$: $\{(allnat, \emptyset), (allnat, \{0\}), (allnat, \{0, 1\}), \ldots\}$. Obviously, it holds $apply(allnat, \emptyset) = apply(allnat, \{0\}) = \cdots = \mathit{false}$ while $apply(allnat, nat) = \mathit{true}$. In other words, $apply$ is not continuous.

Based on the same lines of reasoning, if our domains are arbitrary then the operators K_i used in the sketch of the proof of [Wadge 1991][pages 296-297] are not continuous and therefore Kleene's fixpoint theorem cannot be applied to get the minimum model result. □

As we realized in the first steps of our research for the present paper, if we restrict the domains to contain only *continuous functions*, then *all theorems of [Wadge 1991] hold exactly as stated* and the flaw is corrected. More specifically, given a non-empty

[7]The work in [Wadge 1991] has also been used in order to define a higher-order extension of Datalog [Kountouriotis et al. 2005].

set D, the definition of the semantics of types in [Wadge 1991][page 292] can be altered as follows (we use $[\![\cdot]\!]^*_D$ to denote the new semantics):

— $[\![\iota]\!]^*_D = D$.
— $[\![\iota^n \to \iota]\!]^*_D = D^n \to D$.
— $[\![o]\!]^*_D = \{false, true\}$.
— $[\![\iota \to \pi]\!]^*_D = D \to [\![\pi]\!]^*_D$.
— $[\![\pi_1 \to \pi_2]\!]^*_D = [[\![\pi_1]\!]^*_D \stackrel{c}{\to} [\![\pi_2]\!]^*_D]$, ie., the set of continuous functions from $[\![\pi_1]\!]^*_D$ to $[\![\pi_2]\!]^*_D$.

The corresponding partial orders can be easily defined as in Definition 5.1. The semantics of expressions can be defined in an analogous way as in Definition 5.8, the main difference being that the semantics of application is the standard one. In the following, we will refer to the above semantics as the "*continuous semantics*".

It is easy to check that under the continuous semantics every higher-order logic program (of the fragment considered in [Wadge 1991]) has a unique minimum Herbrand model and this model assigns continuous denotations to all predicates of the program. Moreover, it is straightforward to see that the continuous semantics also applies to the richer language we consider in this paper, ie., a language allowing uninstantiated predicate variables in clauses.

The continuous semantics was the first semantics we considered when we started the research reported in this paper. However, we soon also realized that although the continuous semantics is quite appropriate for the fragment considered in [Wadge 1991], it is not very convenient when the language allows occurrences of uninstantiated predicate variables in clauses. Consider a goal clause of the form $\leftarrow$`p(R)`, where `p` is defined in our program and has type $(\iota \to o) \to o$. Then, the continuous semantics dictates that we should examine *all* possible relations of type $\iota \to o$ for possible solutions. This set of relations is uncountable. Of course we know that since the denotation of `p` in the least model of the program is continuous, there must be some finite set that satisfies `p`. This finiteness property is of vital importance in order to devise any sound and complete proof system for our fragment. The problem, as we understood it, was that the continuous semantics does not reflect the finiteness property *directly*. In other words, *we needed a semantics which would make the finiteness more explicit*.

Actually, there appear to be close connections between the semantics proposed in this paper and the continuous semantics. It is relatively easy to show that for every argument type ρ of $\mathcal{H}$ there is a bijection between the sets $[\![\rho]\!]_D$ and $[\![\rho]\!]^*_D$. Similarly, there is a bijection between the set of interpretations of $\mathcal{H}$ under the proposed semantics and the set of interpretations of $\mathcal{H}$ under the continuous semantics. Then, the following proposition can be established:

PROPOSITION 8.2. *Let* `P` *be a program and let* `F` *be a formula of* $\mathcal{H}$. *Then,* `F` *is a logical consequence of* `P` *under the proposed semantics iff* `F` *is a logical consequence of* `P` *under the continuous semantics.*

An outline of the proof of the above proposition is given in the Electronic Appendix, Section G.

What the above proposition suggests is that the two semantics, despite their differences, are closely related. The key advantage of the proposed semantics is that it is much closer to the SLD-resolution proof system that is introduced in Section 7. More specifically:

— The compact elements of our algebraic lattices correspond to the *basic expressions* that are a vital characteristic of the proposed proof system (see Subsection 7.1).

— The notion of *answer* and *correct answer* for a query (see Definitions 7.16 and 7.17) can now be accurately defined. Notice that the notion of correct answer must be quite close to that of *computed answer* in order to be able to state the main completeness theorem.

In conclusion, the proposed semantics allows us to define an SLD-resolution proof system and it helps us formalize and prove its completeness. It is unclear to us whether (and how) this could have been accomplished by relying on the continuous semantics.

Bezem's approach:. In [Bezem 2001] a notion of extensionality is defined (called the *extensional collapse*) and it is demonstrated that many logic programs are extensional under this notion; however, this notion appears to differ from classical extensionality and has a more proof-theoretical flavor.

The source language considered by Bezem is quite close to the one we adopt in this paper. The *"hoapata"* programs defined in [Bezem 2001] are higher-order logic programs in which the higher-order arguments (hoa) can only be passed (p), applied (a) and/or thrown away (ta). In particular, all higher-order arguments in the head of a clause, are distinct variables. Roughly speaking, these restrictions define a language that is syntactically similar to ours.

However, the semantics of hoapata programs defined in [Bezem 2001] appears to differ significantly from the semantics of $\mathcal{H}$. As it was pointed to us by one of the reviewers, one important difference can be seen when considering queries with uninstantiated predicate variables. More specifically, given the program:

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

the query $\leftarrow$`p(R)` will not return any answer under the approach of [Bezem 2001]. Under our semantics, we will get basic expressions (representing sets), as discussed in the previous sections. However, if the two facts:

```
nat(0).
nat(s(0)).
```

are added to the program, then Bezem's approach will return the answer `R=nat`, while our approach will continue to return the same answers (basic expressions). This difference in behavior when evaluating queries with uninstantiated higher-order arguments, can also be observed when comparing our approach with that of λProlog and HiLog. A more careful discussion of this issue is given at the end of the following subsection.

8.3. Intensional Higher-Order Logic Programming

Research on intensional higher-order logic programming is much more extended. The two main existing approaches in this area are represented by the languages λProlog and HiLog. Both systems have mature implementations and have been tested in various application domains. It should be noted that both λProlog and HiLog encourage a form of higher-order programming that extends in various ways the higher-order programming capabilities that are supported by functional programming languages. For a more detailed discussion on this issue, see [Nadathur and Miller 1998][section 7.4].

In the rest of this section, we give a brief presentation of certain characteristics of these two systems that are related to their intensional behavior (ie., characteristics that will help the reader further clarify the differences between the intensional and extensional approaches). A detailed discussion on the syntax, semantics, implementation and applications of the two languages, is outside the scope of this paper (and the interested reader can consult the relevant bibliography).

λProlog:. The language was initially designed in the late 1980s [Miller and Nadathur 1986; Nadathur 1987; Nadathur and Miller 1990] in order to provide a proof theoretic basis for logic programming. The syntax of λProlog is based on the intuitionistic theory of higher-order hereditary Harrop formulas. The resulting language is a powerful one, that allows the programmer to quantify over function and predicate variables, to use λ-abstractions in terms, and so on. The semantics of λProlog is not extensional (see for example the discussion in [Nadathur and Miller 1998]). The following simple example illustrates this idea.

Example 8.3. Consider the λProlog program (we omit type declarations):

```
r p.
p X :- q X.
q X :- p X.
```

The goal ←`(r q)` fails for the above program. □

Notice that due to the unit clause `r p.`, the above program is not a valid $\mathcal{H}$ program. Actually, as it was first remarked in [Wadge 1991][Example in pages 292-293], if predicate constants are used as parameters in the heads of clauses in a higher-order language, then there does not appear to exist any straightforward way to achieve an extensional minimum-model semantics. In conclusion, λProlog has a very general syntax; this makes the language very powerful from an expressive point of view, but more demanding from a semantic point of view.

Since the syntax of $\mathcal{H}$ is restricted with respect to that of λProlog, in order to compare the two systems, we restrict attention to the syntax of $\mathcal{H}$. In general, we believe that programs of $\mathcal{H}$ that do not involve uninstantiated higher-order variables will have a similar behavior in λProlog and under our proof system (see also the discussion and conjecture at the end of this subsection). For programs or queries that contain uninstantiated higher-order variables, the two systems have in general a different behavior. Consider for example the `band` example given in the introductory section, and the query ←`band(B)`. This goal is not generally a meaningful one for λProlog because there exist too many suitable answer substitutions (ie., predicate terms) for `B` that one could think of (see the relevant discussion in [Nadathur and Miller 1998][page 50]). Notice that there exists at least one suitable answer for `B`, namely the top relation λ`x.`$\top$ that is true of everything, which can be taken as the desired answer in such queries (see again [Nadathur and Miller 1998][page 50]). Our approach in such queries is quite different: the SLD-resolution will enumerate the possible bands (sets) that can be formed given the information in the program. For example, `B` $= \{$`sally, dave, grace`$\}$ is one such band. We believe that such queries that return sets as answers add a nice new characteristic to higher-order logic programming that deserves to be better investigated.

HiLog:. The language possesses a higher-order syntax and a first-order semantics [Chen et al. 1989; 1993]. It extends classical logic programming quite naturally, and allows the programmer to write in a concise way programs that would be rather awkward to code in Prolog. It has been used in various application domains (eg. deductive and object-oriented databases, modular logic programming, and so on).

As it is the case with λProlog, HiLog also differs from $\mathcal{H}$ in the way that it handles queries with higher-order uninstantiated arguments. For example, the ←`band(R)` query will fail in HiLog since there is no actual band defined in the program. However, consider the program:

```
married(john,mary).
```

Then, the query:

```
?-R(john,mary).
```

is a meaningful one for HiLog, and the interpreter will respond with `R = married`. Intuitively, the interpreter searches the program for possible candidate relations and tests them one by one. Of course, if there is no binary relation defined in the program, the above query will fail.

The above program behavior can be best explained by the following comment from [Chen et al. 1993]: "in HiLog predicates and other higher-order syntactic objects are not equal unless they (ie., their names) are equated explicitly".

Some remarks:. From the above discussion, it turns out that the four languages, namely $\mathcal{H}$, the "hoapata" language, λProlog and HiLog, *seem* to have the same behavior when restricted to the fragment proposed in [Wadge 1991]. This is something we had observed during the course of this research and which was also pointed to us by two of the reviewers of the paper. It would be interesting (and certainly nontrivial) to establish this behavioral equivalence in a formal way (ie., by using the semantics of each language in order to demonstrate that for the particular fragment all four languages have the same logical consequences). Of course, as we have seen above, the addition of uninstantiated predicate variables, differentiates $\mathcal{H}$ from the other three proposals.

Notice that despite the fact that the four languages seem to have the same behavior when restricted to the fragment of [Wadge 1991], the proposed approach appears to have an important distinguishing characteristic: since it is based on an extension of classical logic programming, one can use well-established techniques in order to demonstrate program equivalence, the correctness of transformations, and so on (eg., by using induction on the approximations to the least fixpoint of the immediate consequence operator).

8.4. Other Approaches

There exist other approaches in the study of higher-order logic that appear to be connected to higher-order logic programming. One such approach is reported in [Benzmüller et al. 2004] in which the semantics of classical higher-order logic is reexamined with the goal of characterizing the deductive power of existing higher-order theorem provers. More specifically, Church's simply typed λ-calculus is considered as the source language and nine classes of models are identified. Roughly speaking, the classes are distinguished based on "the amount of extensionality" that is present in each one of them.

The work of [Benzmüller et al. 2004] is interesting because the classes of models can be used in order to study and classify existing higher-order theorem provers of varying capabilities. Moreover, as the authors point out (Section 8.1, page 1085 of [Benzmüller et al. 2004]), it is possible that their results may be relevant to higher-order logic programming and in particular to the semantic study of λProlog.

9. FUTURE WORK

There are several aspects of this work, both theoretical and practical, that can be further investigated. In the following, we briefly discuss some of them.

Semantics:. The discussion of the related approaches in Section 8 leaves open some interesting questions that deserve to be further investigated.

First of all, it would be interesting to investigate whether the four different languages mentioned in Section 8 (namely $\mathcal{H}$, "hoapata" programs, λProlog and HiLog) coincide semantically when restricted to the fragment of "definitional programs" in-

troduced in [Wadge 1991]. If such an equivalence holds, then its demonstration must be non-trivial (mainly due to the fact that all four formalisms use different semantic approaches).

A second important question stems from the continuous semantics outlined in Subsection 8.2. It would be interesting to examine whether a proof system can be defined for $\mathcal{H}$ programs which will be sound and complete *with respect to the continuous semantics*. Such an approach would seem more "mainstream" since it would (for example) avoid the non-standard semantics of application given in Definition 5.8.

Implementation:. A prototype implementation of the proposed proof system has been performed in Haskell[8]. A detailed description of the implementation is outside the scope of this paper. However, in the following we outline certain points that we feel are important.

The main difference in comparison to a first-order implementation, is that the proof system has to generate an infinite (yet enumerable) number of basic templates. In order to make more efficient the production of the basic templates, one main optimization has been adopted. As we have already mentioned in Definition 7.3, a basic template is a non-empty finite union of basic expressions of a particularly simple form. In the implementation, the members of this union are generated in a "demand-driven way", as the following examples illustrate.

Example 9.1. Consider the query $\leftarrow$ `(R a b),(R c d)`. The proposed proof system would try some basic templates until it finds one that satisfies the query. However, if it first tries the basic template $(\lambda$`X.`λ`Y.(X`$\approx$`Z)`$\wedge$`(Y`$\approx$`W))` then this will obviously not lead to an answer (since a relation that satisfies the above query must contain at least two pairs of elements). In order to avoid such cases, our implementation initially produces a basic expression that consists of the union of a basic template with an uninstantiated variable (say `L`) of the same type as the template; intuitively, `L` represents a (yet undetermined) set of basic templates that may be needed later during resolution and which need not yet be explicitly generated. In our example, the implementation starts with the production of an expression of the form $(\lambda$`X.`λ`Y.(X`$\approx$`Z)`$\wedge$`(Y`$\approx$`W))` $\bigvee$ `L`. When the second application in the goal is reached, then a second basic template will be generated together with a new uninstantiated variable (say `L1`). The final answer to the query will be an expression of the form: $(\lambda$`X.`λ`Y.(X`$\approx$`a)`$\wedge$`(Y`$\approx$`b))` $\bigvee$ $(\lambda$`X.`λ`Y.(X`$\approx$`c)`$\wedge$`(Y`$\approx$`d))` $\bigvee$ `L1`. The intuitive meaning of the above answer is that the query is satisfied by all relations that contain at least the pairs $(\mathtt{a},\mathtt{b})$ and $(\mathtt{c},\mathtt{d})$.

Notice that an important practical advantage of the above optimization is that a unique answer to the given query is generated. Notice also that if the formal proof system of the previous sections was followed faithfully in the implementation, then an infinite number of answers would be generated: an answer representing the two-element relation $\{(\mathtt{a},\mathtt{b}),(\mathtt{c},\mathtt{d})\}$, an answer representing all three-element relations $\{(\mathtt{a},\mathtt{b}),(\mathtt{c},\mathtt{d}),(\mathtt{X1},\mathtt{X2})\}$, an answer representing the four-element relations $\{(\mathtt{a},\mathtt{b}),(\mathtt{c},\mathtt{d}),(\mathtt{X1},\mathtt{X2}),(\mathtt{X3},\mathtt{X4})\}$, and so on. □

Example 9.2. Consider the `ordered` predicate of Example 3.7 and let $\leftarrow$ `ordered R [1,2,3]` be a query. Following the same ideas as in the previous example, the implementation will produce the unique answer $(\lambda$`X.`λ`Y.(X`$\approx$`1)`$\wedge$`(Y`$\approx$ `2))` $\bigvee$ $(\lambda$`X.`λ`Y.(X`$\approx$`2)`$\wedge$`(Y`$\approx$`3))` $\bigvee$ `L`. Intuitively, this answer states that the list `[1,2,3]` is ordered under any relation of the form $\{(1,2),(2,3)\} \cup L$.

Finally, consider Example 3.6 defining the `closure` predicate. Consider also the query $\leftarrow$ `closure Q a b`. Then, the implementation will enumerate the following (infi-

[8]The code can be retrieved from `http://code.haskell.org/hopes`

nite set of) answers:

$$\begin{array}{rcl} \mathtt{Q} & = & (\lambda \mathtt{X}.\lambda \mathtt{Y}.(\mathtt{X} \approx \mathtt{a}) \wedge (\mathtt{Y} \approx \mathtt{b})) \bigvee \mathtt{L} \\ \mathtt{Q} & = & (\lambda \mathtt{X}.\lambda \mathtt{Y}.(\mathtt{X} \approx \mathtt{a}) \wedge (\mathtt{Y} \approx \mathtt{Z})) \bigvee (\lambda \mathtt{X}.\lambda \mathtt{Y}.(\mathtt{X} \approx \mathtt{Z}) \wedge (\mathtt{Y} \approx \mathtt{b})) \bigvee \mathtt{L} \\ & \ldots & \end{array}$$

which intuitively correspond to relations of the following forms:

$$\begin{array}{rcl} \mathtt{Q} & = & \{(\mathtt{a}, \mathtt{b})\} \cup L \\ \mathtt{Q} & = & \{(\mathtt{a}, \mathtt{Z}), (\mathtt{Z}, \mathtt{b})\} \cup L \\ & \ldots & \end{array}$$

Intuitively, the above answers state that the pair $(\mathtt{a}, \mathtt{b})$ belongs to the transitive closure of all relations that contain at least the pair $(\mathtt{a}, \mathtt{b})$; moreover, it also belongs to the transitive closure of all relations that contain at least two pairs of the form $(\mathtt{a}, \mathtt{Z})$ and $(\mathtt{Z}, \mathtt{b})$ for any $\mathtt{Z}$, and so on. □

We conjecture that the above ideas can be used in order to define a new, more efficient proof procedure for $\mathcal{H}$ which will also be sound and complete with respect to the semantics introduced in this paper. Of course, an attempt in this direction would mean that the notions of *basic expression* and *basic template* would have to be altered: the present definitions do not allow a basic expression to be the union of a predicate variable (like L in the above examples) with another basic expression. We believe that all the proofs of Section 7 can be adapted in order to work under the new definitions and the new proof procedure, but this has to be further investigated.

We are currently considering issues regarding an extended WAM-based implementation of the ideas presented in the paper. We believe that ideas originating from graph-reduction [Field and Harrison 1988] will also prove vital in the development of this extended implementation.

Other Extensions:. Another interesting direction for future research is the extension of our higher-order fragment with negation-as-failure. The semantics of negation in a higher-order setting could probably be captured model-theoretically using the purely logical characterization of the well-founded semantics through an appropriate infinite-valued logic given in [Rondogiannis and Wadge 2005].

Also, it would be interesting to investigate whether it is possible to build a higher-order logic programming language that would combine both extensional and intensional characteristics. Designing such a language would obviously be important because such a system could embody all the benefits from both the extensional and the intensional approaches.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their intriguing remarks and suggestions. We would also like to thank Costas Koutras for valuable discussions regarding algebraic lattices and the reading group on programming languages at the University of Athens for many insightful comments.

REFERENCES

ABRAMSKY, S. AND JUNG, A. 1994. Domain theory. In *Handbook of Logic in Computer Science III*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Clarendon Press.

APT, K. R. 1990. Logic programming. In *Handbook of Theoretical Computer Science (Vol. B)*, J. van Leeuwen, Ed. MIT Press, Cambridge, MA, USA, 493–574.

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

BENZMÜLLER, C., BROWN, C. E., AND KOHLHASE, M. 2004. Higher-order semantics and extensionality. *Journal of Symbolic Logic 69,* 4, 1027–1088.

BEZEM, M. 1999. Extensionality of simply typed logic programs. In *International Conference on Logic Programming (ICLP)*. The MIT Press, 395–410.

BEZEM, M. 2001. An improved extensionality criterion for higher-order logic programs. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL)*. Springer-Verlag, London, UK, 203–216.

BIRKHOFF, G. 1967. *Lattice theory*. American Mathematical Society.

CHEN, W., KIFER, M., AND WARREN, D. 1989. Hilog as a platform for database languages. *IEEE Data Engineering Bulletin 12,* 3, 37–44.

CHEN, W., KIFER, M., AND WARREN, D. S. 1993. Hilog: A foundation for higher-order logic programming. *Journal of Logic Programming 15,* 3, 187–230.

CLARK, K. L. 1979. *Predicate Logic as a Computational Formalism*. Research Report DOC 79/59. Department of Computing, Imperial College.

DOWTY, D., WALL, R., AND PETERS, S. 1981. *Introduction to Montague Semantics*. Studies in Linguistics and Philosophy. Kluwer Academic Publishers.

FIELD, A. AND HARRISON, P. 1988. *Functional Programming*. International computer science series. Addison-Wesley.

GRÄTZER, G. 1978. *General Lattice Theory*. Academic Press.

HILL, R. 1974. *LUSH-resolution and Its Completeness*. DCL Memo 78. University of Edinburgh, Department of Artificial Intelligence.

KLEENE, S. 1959. Countable functionals. In *Constructivity in Mathematics*, A. Heyting, Ed. North Holland, 81–100.

KOUNTOURIOTIS, V., RONDOGIANNIS, P., AND WADGE, W. W. 2005. Extensional higher-order datalog. In *Short Paper Proceeding of the 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. 1–5.

KREISEL, G. 1959. Interpretation of analysis by means of functionals of finite type. In *Constructivity in Mathematics*, A. Heyting, Ed. North Holland, 101–128.

LLOYD, J. W. 1987. *Foundations of Logic Programming*. Springer Verlag.

MILLER, D. AND NADATHUR, G. 1986. Higher-order logic programming. In *Proceedings of the Third International Conference on Logic Programming (ICLP)*. 448–462.

NADATHUR, G. 1987. A higher-order logic as the basis for logic programming. Ph.D. thesis, University of Pennsylvania.

NADATHUR, G. AND MILLER, D. 1990. Higher-order horn clauses. *J. ACM 37,* 4, 777–814.

NADATHUR, G. AND MILLER, D. 1998. Higher-order logic programming. In *Handbook of Logics for Artificial Intelligence and Logic Programming*. Vol. 5. Claredon Press, 499–590.

NORMANN, D. 2006. Computing with functionals - computability theory or computer science. *Bulletin of Symbolic Logic 12,* 1, 43–59.

RONDOGIANNIS, P. AND WADGE, W. W. 2005. Minimum model semantics for logic programs with negation-as-failure. *ACM Trans. Comput. Logic 6,* 2, 441–467.

STOY, J. E. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA.

TENNENT, R. D. 1991. *Semantics of Programming Languages*. Prentice-Hall.

WADGE, W. W. 1991. Higher-order horn logic programming. In *Proceedings of the International Symposium on Logic Programming (ISLP)*. 289–303.

WARREN, D. H. 1982. Higher-order extensions to prolog: are they needed? *Machine Intelligence*, 441–454.

Online Appendix to: Extensional Higher-Order Logic Programming

A. CHARALAMBIDIS, K. HANDJOPOULOS, P. RONDOGIANNIS, University of Athens
W. W. WADGE, University of Victoria

A. PROOF OF LEMMA 4.17

In order to establish Lemma 4.17, we first demonstrate the following auxiliary propositions:

PROPOSITION A.1. *Let A be a poset and L be an algebraic lattice. Then, for each step function $(a \searrow c)$ and for every $f : [A \xrightarrow{m} L]$ it holds that $(a \searrow c) \sqsubseteq f$ if and only if $c \sqsubseteq f(a)$.*

PROOF. If $(a \searrow c) \sqsubseteq f$, by applying both functions to a we get $c \sqsubseteq f(a)$. Now suppose that $c \sqsubseteq f(a)$ and consider an arbitrary $x \in A$. In case $a \sqsubseteq x$, we have $(a \searrow c)(x) = c$ thus, since $c \sqsubseteq f(a)$ and f is monotonic, $(a \searrow c)(x) \sqsubseteq f(x)$. Otherwise, $(a \searrow c)(x) = \bot_L$ thus $(a \searrow c)(x) \sqsubseteq f(x)$. It follows that $(a \searrow c) \sqsubseteq f$. □

PROPOSITION A.2. *Let L be a complete lattice and assume there exists $B \subseteq \mathcal{K}(L)$ such that for every $x \in L$, $x = \bigsqcup B_{[x]}$. Then L is an algebraic lattice (ω-algebraic if B is countable) with basis $\mathcal{K}(L) = \{\bigsqcup M \mid M$ is a finite subset of $B\}$.*

PROOF. It is immediate that L is algebraic, since by assumption every element of L can be written as the least upper bound of a set of compact elements of L. The nontrivial part is establishing the relation between $\mathcal{K}(L)$ and B.

Given $x \in L$, we let $\Delta(x)$ be the set $\{\bigsqcup M \mid M$ is a finite subset of $B_{[x]}\}$. Notice that $B_{[x]} = \bigcup\{M \mid M$ is a finite subset of $B_{[x]}\}$. Using Proposition 4.3(2), we have that $\bigsqcup B_{[x]} = \bigsqcup \Delta(x)$ and thus $\bigsqcup \Delta(x) = x$. We show that for each $x \in L$ it holds that $\mathcal{K}(L)_{[x]} = \Delta(x)$ by proving that each set is a subset of the other one.

First consider an arbitrary $c \in \mathcal{K}(L)_{[x]}$ and recall that $c = \bigsqcup \Delta(c)$. By the compactness of c, there exists a finite $A \subseteq \Delta(c)$ such that $c \sqsubseteq \bigsqcup A$. But then $\bigsqcup A \sqsubseteq c$ because c is an upper bound of $\Delta(c)$, and therefore $c = \bigsqcup A$. By the definition of $\Delta(c)$ and the fact that $A \subseteq \Delta(c)$, we get that $c = \bigsqcup\{\bigsqcup M_1, \ldots, \bigsqcup M_r\}$, where $M_1, \ldots, M_r$ are finite subsets of $B_{[c]}$. By Proposition 4.3(2), $c = \bigsqcup (M_1 \cup \cdots \cup M_r)$. In other words there exists a finite set $M = M_1 \cup \cdots \cup M_r$ such that $M \subseteq B_{[c]} \subseteq B_{[x]}$ and $c = \bigsqcup M$, which means that $c \in \Delta(x)$.

On the other hand, consider a finite set $M = \{c_1, \ldots, c_n\} \subseteq B_{[x]}$ such that $\bigsqcup M \in \Delta(x)$. Let A be a subset of L such that $\bigsqcup M \sqsubseteq \bigsqcup A$. Due to the compactness of each c_i, by $c_i \sqsubseteq \bigsqcup A$ we get $c_i \sqsubseteq \bigsqcup A_i$ for some finite $A_i \subseteq A$. But then, for every i, $c_i \sqsubseteq \bigsqcup A_i \sqsubseteq \bigsqcup\{\bigsqcup A_1, \ldots, \bigsqcup A_n\} = \bigsqcup(A_1 \cup \cdots \cup A_n)$. In other words, $\bigsqcup M \sqsubseteq \bigsqcup(A_1 \cup \cdots \cup A_n)$, which implies that $\bigsqcup M$ is compact. Moreover, since x is an upper bound of M, we have that $\bigsqcup M \in \mathcal{K}(L)_{[x]}$. Hence, $\mathcal{K}(L)_{[x]} = \Delta(x)$.

To complete the proof, simply take $x = \bigsqcup L$ in the equality $\mathcal{K}(L)_{[x]} = \Delta(x)$. If, additionally, B is countable, the cardinality of $\mathcal{K}(L)$ is bounded by the number of finite subsets of a countable set, which is countable. Hence, L is an ω-algebraic lattice in this case. □

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

We can now proceed to the proof of Lemma 4.17:

Lemma 4.17 Let A be a poset and L be an algebraic lattice. Then, $[A \xrightarrow{m} L]$ is an algebraic lattice whose basis is the set of all least upper bounds of finitely many step functions from A to L. If, additionally, A is countable and L is an ω-algebraic lattice then $[A \xrightarrow{m} L]$ is an ω-algebraic lattice.

PROOF. Let B denote the set of all step functions from A to L. Recall that $[A \xrightarrow{m} L]$ forms a complete lattice by Proposition 4.10. Let $(a \searrow c) \in B$ be an arbitrary step function. We show that $(a \searrow c)$ is compact. Consider a set F of monotonic functions from A to L such that $(a \searrow c) \sqsubseteq \bigsqcup F$. By Propositions A.1 and 4.10 we get that $c \sqsubseteq \bigsqcup_{f \in F} f(a)$. By the compactness of c, there exists a finite $F' \subseteq F$ such that $c \sqsubseteq \bigsqcup_{f \in F'} f(a)$. Let $f' = \bigsqcup F'$. Then, $c \sqsubseteq f'(a)$, or equivalently by Proposition A.1, $(a \searrow c) \sqsubseteq f' = \bigsqcup F'$. Hence, $(a \searrow c)$ is compact.

We now show that every monotonic function $f \in [A \xrightarrow{m} L]$ is the least upper bound of $B_{[f]}$. Since f is an upper bound of this set, we let g be an upper bound of $B_{[f]}$ and prove that $f \sqsubseteq g$. In fact, we consider an arbitrary $x \in A$ and prove that $f(x) \sqsubseteq g(x)$. Suppose S_x is the set of all step functions $h_c = (x \searrow c)$ for every compact element $c \in \mathcal{K}(L)_{[f(x)]}$. By Proposition A.1, we have that for all step functions $h_c \in S_x$, $h_c \sqsubseteq f$; thus S_x is a subset of $B_{[f]}$. Since g is an upper bound of $B_{[f]}$, it must also be an upper bound of S_x, therefore it holds that $h_c \sqsubseteq g$ for each $h_c \in S_x$. Applying this inequality for x we get that $c \sqsubseteq g(x)$ for each $c \in \mathcal{K}(L)_{[f(x)]}$, therefore $\bigsqcup \mathcal{K}(L)_{[f(x)]} \sqsubseteq g(x)$. Since L is an algebraic lattice, $f(x)$ is the least upper bound of $\mathcal{K}(L)_{[f(x)]}$, thus $f(x) \sqsubseteq g(x)$. Hence, f is the least upper bound of $B_{[f]}$.

On the whole, we have shown that B is a subset of $\mathcal{K}([A \xrightarrow{m} L])$ such that each monotonic function f from A to L is the least upper bound of $B_{[f]}$. Notice that if, additionally, A is countable and L is an ω-algebraic lattice, then B is countable because its cardinality is equal to that of the cartesian product of two countable sets. Now apply Proposition A.2. □

B. PROOF OF LEMMA 5.10

Lemma 5.10 Let $\mathsf{E} : \rho$ be an expression of $\mathcal{H}$ and let D be a nonempty set. Moreover, let s, s_1, s_2 be states over D and let I be an interpretation over D. Then:

(1) $[\![\mathsf{E}]\!]_s(I) \in [\![\rho]\!]_D$.
(2) If E is positive and $s_1 \sqsubseteq_{\mathcal{S}_{\mathcal{H},D}} s_2$ then $[\![\mathsf{E}]\!]_{s_1}(I) \sqsubseteq_\rho [\![\mathsf{E}]\!]_{s_2}(I)$.

PROOF. The two statements are established simultaneously by a structural induction on E.

Induction Basis: The cases for E being false, true, c, p or V, are all straightforward.

Induction Step: The interesting cases are $\mathsf{E} = (\mathsf{E}_1\mathsf{E}_2)$ and $\mathsf{E} = (\lambda\mathsf{V}.\mathsf{E}_1)$. The other cases are easier and omitted.

Case 1: $\mathsf{E} = (\mathsf{E}_1\mathsf{E}_2)$. We examine the two statements of the lemma:

Statement 1: Assume that $\mathsf{E}_1 : \rho_1 \to \pi_2$ and $\mathsf{E}_2 : \rho_1$. Then, it suffices to demonstrate that $[\![(\mathsf{E}_1\mathsf{E}_2)]\!]_s(I) \in [\![\pi_2]\!]_D$, or equivalently that $\bigsqcup_{b \in B}([\![\mathsf{E}_1]\!]_s(I)(b)) \in [\![\pi_2]\!]_D$, where $B = \mathcal{F}_D(type(\mathsf{E}_2))_{[[\![\mathsf{E}_2]\!]_s(I)]} = \{b \in \mathcal{F}_D(type(\mathsf{E}_2)) \mid b \sqsubseteq [\![\mathsf{E}_2]\!]_s(I)\}$. By the induction hypothesis, $[\![\mathsf{E}_1]\!]_s(I) \in [\![\rho_1 \to \pi_2]\!]_D$ and $[\![\mathsf{E}_2]\!]_s(I) \in [\![\rho_1]\!]_D$. But then, for every $b \in B$, $[\![\mathsf{E}_1]\!]_s(I)(b) \in [\![\pi_2]\!]_D$ and since $[\![\pi_2]\!]_D$ is a complete lattice, we get that $\bigsqcup_{b \in B}([\![\mathsf{E}_1]\!]_s(I)(b)) \in [\![\pi_2]\!]_D$.

Statement 2: It suffices to demonstrate that $[\![(\mathsf{E}_1\mathsf{E}_2)]\!]_{s_1}(I) \sqsubseteq [\![(\mathsf{E}_1\mathsf{E}_2)]\!]_{s_2}(I)$, or equivalently that $\bigsqcup_{b_2\in B_2}([\![\mathsf{E}_1]\!]_{s_1}(I)(b_2)) \sqsubseteq \bigsqcup_{b'_2\in B'_2}([\![\mathsf{E}_1]\!]_{s_2}(I)(b'_2))$, where $B_2 = \mathcal{F}_D(type(\mathsf{E}_2))_{[[\![\mathsf{E}_2]\!]_{s_1}(I)]}$ and $B'_2 = \mathcal{F}_D(type(\mathsf{E}_2))_{[[\![\mathsf{E}_2]\!]_{s_2}(I)]}$. Notice that by definition, $B_2 = \{b \in \mathcal{F}_D(type(\mathsf{E}_2)) \mid b \sqsubseteq [\![\mathsf{E}_2]\!]_{s_1}(I)\}$ and $B'_2 = \{b \in \mathcal{F}_D(type(\mathsf{E}_2)) \mid b \sqsubseteq [\![\mathsf{E}_2]\!]_{s_2}(I)\}$. By the induction hypothesis we have $[\![\mathsf{E}_2]\!]_{s_1}(I) \sqsubseteq [\![\mathsf{E}_2]\!]_{s_2}(I)$, and therefore $B_2 \subseteq B'_2$. By the induction hypothesis we also have that $[\![\mathsf{E}_1]\!]_{s_1}(I) \sqsubseteq [\![\mathsf{E}_1]\!]_{s_2}(I)$. By the induction hypothesis for the first statement of the lemma, both $[\![\mathsf{E}_1]\!]_{s_1}(I)$ and $[\![\mathsf{E}_1]\!]_{s_2}(I)$ are monotonic functions since they belong to $[\![\rho_1 \to \pi_2]\!]_D$. Therefore, $\bigsqcup_{b_2\in B_2}([\![\mathsf{E}_1]\!]_{s_1}(I)(b_2)) \sqsubseteq \bigsqcup_{b'_2\in B'_2}([\![\mathsf{E}_1]\!]_{s_2}(I)(b'_2))$, or equivalently $[\![(\mathsf{E}_1\mathsf{E}_2)]\!]_{s_1}(I) \sqsubseteq [\![(\mathsf{E}_1\mathsf{E}_2)]\!]_{s_2}(I)$.

Case 2: $\mathsf{E} = (\lambda\mathsf{V}.\mathsf{E}_1)$. We examine the two statements of the lemma:

Statement 1: Assume that $\mathsf{V} : \rho_1$ and $\mathsf{E}_1 : \pi_1$. We show that $[\![(\lambda\mathsf{V}.\mathsf{E}_1)]\!]_s(I) \in [\![\rho_1 \to \pi_1]\!]_D$. We distinguish two cases, namely $\rho_1 = \iota$ and $\rho_1 = \pi$. If $\rho_1 = \iota$ then the result follows easily using the induction hypothesis for the first statement of the lemma. If $\rho_1 = \pi$, then we must demonstrate that $[\![(\lambda\mathsf{V}.\mathsf{E}_1)]\!]_s(I) \in [\![\pi \to \pi_1]\!]_D = [\mathcal{K}([\![\pi]\!]_D) \stackrel{m}{\to} [\![\pi_1]\!]_D]$. In other words, we need to show that the function $\lambda d.[\![\mathsf{E}_1]\!]_{s[d/\mathsf{V}]}(I)$ is monotonic. But this follows directly from the induction hypothesis for the second statement of the lemma.

Statement 2: It suffices to show that $[\![(\lambda\mathsf{V}.\mathsf{E}_1)]\!]_{s_1}(I) \sqsubseteq [\![(\lambda\mathsf{V}.\mathsf{E}_1)]\!]_{s_2}(I)$. By the semantics of lambda abstraction, it suffices to show that $\lambda d.[\![\mathsf{E}_1]\!]_{s_1[d/\mathsf{V}]}(I) \sqsubseteq \lambda d.[\![\mathsf{E}_1]\!]_{s_2[d/\mathsf{V}]}(I)$, or that for every d, $[\![\mathsf{E}_1]\!]_{s_1[d/\mathsf{V}]}(I) \sqsubseteq [\![\mathsf{E}_1]\!]_{s_2[d/\mathsf{V}]}(I)$, which holds by the induction hypothesis. □

C. PROOF OF LEMMA 6.6

Lemma 6.6 Let P be a program and let $\mathsf{E} : \rho$ be a positive expression of P. Let I, J be Herbrand interpretations and s be a Herbrand state of P . If $I \sqsubseteq_{\mathcal{I}_\mathsf{P}} J$ then $[\![\mathsf{E}]\!]_s(I) \sqsubseteq_\rho [\![\mathsf{E}]\!]_s(J)$.

PROOF. The proof is by a structural induction on E.

Induction Basis: The cases for E being false, true, c, p or V, are all straightforward.

Induction Step: The interesting cases are $\mathsf{E} = (\mathsf{E}_1\mathsf{E}_2)$ and $\mathsf{E} = (\lambda\mathsf{V}.\mathsf{E}_1)$. The other cases are easier and omitted.

Case 1: $\mathsf{E} = (\mathsf{E}_1\mathsf{E}_2)$. It suffices to demonstrate that $[\![(\mathsf{E}_1\mathsf{E}_2)]\!]_s(I) \sqsubseteq [\![(\mathsf{E}_1\mathsf{E}_2)]\!]_s(J)$, or equivalently that $\bigsqcup_{b_2\in B_2}([\![\mathsf{E}_1]\!]_s(I)(b_2)) \sqsubseteq \bigsqcup_{b'_2\in B'_2}([\![\mathsf{E}_1]\!]_s(J)(b'_2))$, where $B_2 = \mathcal{F}_D(type(\mathsf{E}_2))_{[[\![\mathsf{E}_2]\!]_s(I)]}$ and $B'_2 = \mathcal{F}_D(type(\mathsf{E}_2))_{[[\![\mathsf{E}_2]\!]_s(J)]}$. Notice that by definition, $B_2 = \{b \in \mathcal{F}_D(type(\mathsf{E}_2)) \mid b \sqsubseteq [\![\mathsf{E}_2]\!]_s(I)\}$ and $B'_2 = \{b \in \mathcal{F}_D(type(\mathsf{E}_2)) \mid b \sqsubseteq [\![\mathsf{E}_2]\!]_s(J)\}$. By the induction hypothesis we have $[\![\mathsf{E}_2]\!]_s(I) \sqsubseteq [\![\mathsf{E}_2]\!]_s(J)$, and therefore $B_2 \subseteq B'_2$. By the induction hypothesis we also have that $[\![\mathsf{E}_1]\!]_s(I) \sqsubseteq [\![\mathsf{E}_1]\!]_s(J)$. Therefore, $\bigsqcup_{b_2\in B_2}([\![\mathsf{E}_1]\!]_s(I)(b_2)) \sqsubseteq \bigsqcup_{b'_2\in B'_2}([\![\mathsf{E}_1]\!]_s(J)(b'_2))$, or equivalently $[\![(\mathsf{E}_1\mathsf{E}_2)]\!]_s(I) \sqsubseteq [\![(\mathsf{E}_1\mathsf{E}_2)]\!]_s(J)$.

Case 2: $\mathsf{E} = (\lambda\mathsf{V}.\mathsf{E}_1)$. It suffices to show that $[\![(\lambda\mathsf{V}.\mathsf{E}_1)]\!]_s(I) \sqsubseteq [\![(\lambda\mathsf{V}.\mathsf{E}_1)]\!]_s(J)$. By the semantics of lambda abstraction, it suffices to show that $\lambda d.[\![\mathsf{E}_1]\!]_{s[d/\mathsf{V}]}(I) \sqsubseteq \lambda d.[\![\mathsf{E}_1]\!]_{s[d/\mathsf{V}]}(J)$, or that for every d, $[\![\mathsf{E}_1]\!]_{s[d/\mathsf{V}]}(I) \sqsubseteq [\![\mathsf{E}_1]\!]_{s[d/\mathsf{V}]}(J)$, which holds by the induction hypothesis. □

D. PROOF OF LEMMA 6.7

Lemma 6.7 Let P be a program and let E be any positive expression of P. Let $\mathcal{I}$ be a directed set of Herbrand interpretations and s be a Herbrand state of P. Then, $[\![\mathsf{E}]\!]_s(\bigsqcup \mathcal{I}) = \bigsqcup_{I\in\mathcal{I}} [\![\mathsf{E}]\!]_s(I)$.

PROOF. The proof can be performed in two steps: we first show that $[\![\mathsf{E}]\!]_s(\bigsqcup \mathcal{I}) \sqsupseteq \bigsqcup_{I\in\mathcal{I}} [\![\mathsf{E}]\!]_s(I)$ and then that $[\![\mathsf{E}]\!]_s(\bigsqcup \mathcal{I}) \sqsubseteq \bigsqcup_{I\in\mathcal{I}} [\![\mathsf{E}]\!]_s(I)$.

For the first of these two statements observe that by Lemma 6.6, we have that $[\![\mathsf{E}]\!]_s(\bigsqcup \mathcal{I}) \sqsupseteq [\![\mathsf{E}]\!]_s(I)$, for all $I \in \mathcal{I}$. But then $[\![\mathsf{E}]\!]_s(\bigsqcup \mathcal{I})$ is an upper bound of the set $\{[\![\mathsf{E}]\!]_s(I) \mid I \in \mathcal{I}\}$, and therefore $[\![\mathsf{E}]\!]_s(\bigsqcup \mathcal{I}) \sqsupseteq \bigsqcup_{I\in\mathcal{I}} [\![\mathsf{E}]\!]_s(I)$. It remains to show that $[\![\mathsf{E}]\!]_s(\bigsqcup \mathcal{I}) \sqsubseteq \bigsqcup_{I\in\mathcal{I}} [\![\mathsf{E}]\!]_s(I)$. The proof is by a structural induction on E.

Induction Basis: The cases for E being false, true, c, p or V, are all straightforward.

Induction Hypothesis: Assume that for given expressions $\mathsf{E}_1, \mathsf{E}_2$ it holds that $[\![\mathsf{E}_i]\!]_s(\bigsqcup \mathcal{I}) = \bigsqcup_{I\in\mathcal{I}} [\![\mathsf{E}_i]\!]_s(I)$, $i \in \{1, 2\}$. Notice that we assume equality. This is due to the fact that the one direction has already been established for all expressions while the other direction is assumed.

Induction Step: We distinguish the following cases:

Case 1: $\mathsf{E} = \mathsf{f}\ \mathsf{E}_1 \cdots \mathsf{E}_n$. This case is straightforward since for every interpretation I and for every state s, the value of $[\![\mathsf{f}\ \mathsf{E}_1 \cdots \mathsf{E}_n]\!]_s(I)$ only depends on s (since the expressions $\mathsf{E}_1, \ldots, \mathsf{E}_n$ are of type ι and do not contain predicate symbols).

Case 2: $\mathsf{E} = (\mathsf{E}_1\mathsf{E}_2)$. Assume that $\mathsf{E}_2 : \rho$. Then:

$$
\begin{aligned}
& [\![(\mathsf{E}_1\mathsf{E}_2)]\!]_s(\bigsqcup \mathcal{I}) = \\
&= \bigsqcup_{b\in B}([\![\mathsf{E}_1]\!]_s(\bigsqcup \mathcal{I})(b)), \text{ where } B = \{b \in \mathcal{F}_D(\rho) \mid b \sqsubseteq [\![\mathsf{E}_2]\!]_s(\bigsqcup \mathcal{I})\} \\
&\quad \text{(Semantics of application)} \\
&= \bigsqcup_{b\in B}((\bigsqcup_{I\in\mathcal{I}} [\![\mathsf{E}_1]\!]_s(I))(b)), \text{ where } B = \{b \in \mathcal{F}_D(\rho) \mid b \sqsubseteq [\![\mathsf{E}_2]\!]_s(\bigsqcup \mathcal{I})\} \\
&\quad \text{(Induction hypothesis)} \\
&= \bigsqcup_{b\in B}(\bigsqcup_{I\in\mathcal{I}} [\![\mathsf{E}_1]\!]_s(I)(b)), \text{ where } B = \{b \in \mathcal{F}_D(\rho) \mid b \sqsubseteq [\![\mathsf{E}_2]\!]_s(\bigsqcup \mathcal{I})\} \\
&\quad \text{(Proposition 4.10)} \\
&= \bigsqcup\{[\![\mathsf{E}_1]\!]_s(I)(b) \mid I \in \mathcal{I},\ b \in \mathcal{F}_D(\rho),\ b \sqsubseteq [\![\mathsf{E}_2]\!]_s(\bigsqcup \mathcal{I})\} \\
&\quad \text{(Proposition 4.3(2))} \\
&= \bigsqcup\{[\![\mathsf{E}_1]\!]_s(I)(b) \mid I \in \mathcal{I},\ b \in \mathcal{F}_D(\rho),\ b \sqsubseteq \bigsqcup_{I\in\mathcal{I}} [\![\mathsf{E}_2]\!]_s(I)\} \\
&\quad \text{(Induction hypothesis)} \\
&= \bigsqcup\{[\![\mathsf{E}_1]\!]_s(I)(b) \mid I \in \mathcal{I},\ b \in \mathcal{F}_D(\rho),\ b \sqsubseteq \bigsqcup_{J\in F} [\![\mathsf{E}_2]\!]_s(J),\ F \text{ finite subset of } \mathcal{I}\} \\
&\quad \text{(Since } b \text{ is either a compact element or a member of } D\text{)} \\
&\sqsubseteq \bigsqcup\{[\![\mathsf{E}_1]\!]_s(I)(b) \mid I \in \mathcal{I},\ b \in \mathcal{F}_D(\rho),\ b \sqsubseteq [\![\mathsf{E}_2]\!]_s(J)\},\ \text{for some } J \in \mathcal{I} \\
&\quad \text{(Because } \mathcal{I} \text{ is directed and } [\![\mathsf{E}_2]\!]_s \text{ is monotonic by Lemma 6.6)} \\
&\sqsubseteq \bigsqcup\{[\![\mathsf{E}_1]\!]_s(I)(b) \mid I \in \mathcal{I},\ J \in \mathcal{I},\ b \in \mathcal{F}_D(\rho),\ b \sqsubseteq [\![\mathsf{E}_2]\!]_s(J)\} \\
&\quad \text{(Proposition 4.3(1))} \\
&\sqsubseteq \bigsqcup_{I\in\mathcal{I},J\in\mathcal{I}} \bigsqcup\{[\![\mathsf{E}_1]\!]_s(I)(b) \mid b \in \mathcal{F}_D(\rho),\ b \sqsubseteq [\![\mathsf{E}_2]\!]_s(J)\} \\
&\quad \text{(Proposition 4.3(2))} \\
&\sqsubseteq \bigsqcup_{I\in\mathcal{I}} \bigsqcup\{[\![\mathsf{E}_1]\!]_s(I)(b) \mid b \in \mathcal{F}_D(\rho),\ b \sqsubseteq [\![\mathsf{E}_2]\!]_s(I)\} \\
&\quad \text{(Proposition 4.7)} \\
&= \bigsqcup_{I\in\mathcal{I}} [\![(\mathsf{E}_1\mathsf{E}_2)]\!]_s(I) \\
&\quad \text{(Semantics of application)}
\end{aligned}
$$

Case 3: $\mathsf{E} = (\lambda\mathsf{V}.\mathsf{E}_1)$. We show that $[\![(\lambda\mathsf{V}.\mathsf{E}_1)]\!]_s(\bigsqcup \mathcal{I}) \sqsubseteq \bigsqcup_{I\in\mathcal{I}} [\![(\lambda\mathsf{V}.\mathsf{E}_1)]\!]_s(I)$. Consider $b \in \mathcal{F}_D(type(\mathsf{V}))$. By the semantics of lambda abstraction we get that $[\![(\lambda\mathsf{V}.\mathsf{E}_1)]\!]_s(\bigsqcup \mathcal{I})(b) = [\![\mathsf{E}_1]\!]_{s[b/\mathsf{V}]}(\bigsqcup \mathcal{I})$; by the induction hypothesis this is equal to $\bigsqcup_{I\in\mathcal{I}} [\![\mathsf{E}_1]\!]_{s[b/\mathsf{V}]}(I)$, which by Proposition 4.10 is equal to $(\bigsqcup_{I\in\mathcal{I}} [\![(\lambda\mathsf{V}.\mathsf{E}_1)]\!]_s(I))(b)$.

Case 4: $\mathsf{E} = (\mathsf{E}_1 \bigvee_\pi \mathsf{E}_2)$. We show that $[\![(\mathsf{E}_1 \bigvee_\pi \mathsf{E}_2)]\!]_s(\bigsqcup \mathcal{I}) \sqsubseteq \bigsqcup_{I\in\mathcal{I}} [\![(\mathsf{E}_1 \bigvee_\pi \mathsf{E}_2)]\!]_s(I)$, ie., that for all $b_1, \ldots, b_n$, if $[\![(\mathsf{E}_1 \bigvee_\pi \mathsf{E}_2)]\!]_s(\bigsqcup \mathcal{I})\, b_1 \cdots b_n = true$ then $(\bigsqcup_{I\in\mathcal{I}} [\![(\mathsf{E}_1 \bigvee_\pi \mathsf{E}_2)]\!]_s(I))\, b_1 \cdots b_n = true$. By the semantics of $\bigvee_\pi$ we get that if $[\![(\mathsf{E}_1 \bigvee_\pi \mathsf{E}_2)]\!]_s(\bigsqcup \mathcal{I})\, b_1 \cdots b_n = true$ then it either holds that $[\![\mathsf{E}_1]\!]_s(\bigsqcup \mathcal{I})\, b_1 \cdots b_n = true$ or $[\![\mathsf{E}_2]\!]_s(\bigsqcup \mathcal{I})\, b_1 \cdots b_n = true$. By the induction hypothesis and Proposition 4.10 we get that either $\bigsqcup_{I\in\mathcal{I}}([\![\mathsf{E}_1]\!]_s(I)\, b_1 \cdots b_n) = true$ or $\bigsqcup_{I\in\mathcal{I}}([\![\mathsf{E}_2]\!]_s(I)\, b_1 \cdots b_n) = true$. Then there must exist $I \in \mathcal{I}$ such that either $[\![\mathsf{E}_1]\!]_s(I)\, b_1 \cdots b_n = true$ or $[\![\mathsf{E}_2]\!]_s(I)\, b_1 \cdots b_n = true$. By the semantics of $\bigvee_\pi$ we get that $[\![(\mathsf{E}_1 \bigvee_\pi \mathsf{E}_2)]\!]_s(I)\, b_1 \cdots b_n = true$ and therefore $(\bigsqcup_{I\in\mathcal{I}} [\![(\mathsf{E}_1 \bigvee_\pi \mathsf{E}_2)]\!]_s(I))\, b_1 \cdots b_n = true$.

Case 5: $\mathsf{E} = (\mathsf{E}_1 \bigwedge_\pi \mathsf{E}_2)$. We show that $[\![(\mathsf{E}_1 \bigwedge_\pi \mathsf{E}_2)]\!]_s(\bigsqcup \mathcal{I}) \sqsubseteq \bigsqcup_{I\in\mathcal{I}} [\![(\mathsf{E}_1 \bigwedge_\pi \mathsf{E}_2)]\!]_s(I)$. In other words, it suffices to show that for all $b_1, \ldots, b_n$, if $[\![(\mathsf{E}_1 \bigwedge_\pi \mathsf{E}_2)]\!]_s(\bigsqcup \mathcal{I})\, b_1 \cdots b_n = \mathit{true}$ then $(\bigsqcup_{I\in\mathcal{I}} [\![(\mathsf{E}_1 \bigwedge_\pi \mathsf{E}_2)]\!]_s(I))\, b_1 \cdots b_n = \mathit{true}$. But if $[\![(\mathsf{E}_1 \bigwedge_\pi \mathsf{E}_2)]\!]_s(\bigsqcup \mathcal{I})\, b_1 \cdots b_n = \mathit{true}$, then by the semantics of $\bigwedge_\pi$ we get that $[\![\mathsf{E}_1]\!]_s(\bigsqcup \mathcal{I})\, b_1 \cdots b_n = \mathit{true}$ and $[\![\mathsf{E}_2]\!]_s(\bigsqcup \mathcal{I})\, b_1 \cdots b_n = \mathit{true}$. By the induction hypothesis and Proposition 4.10 this implies that $\bigsqcup_{I\in\mathcal{I}}([\![\mathsf{E}_1]\!]_s(I)\, b_1 \cdots b_n) = \mathit{true}$ and $\bigsqcup_{I\in\mathcal{I}}([\![\mathsf{E}_2]\!]_s(I)\, b_1 \cdots b_n) = \mathit{true}$. This means that there must exist $I_1, I_2 \in \mathcal{I}$ such that $[\![\mathsf{E}_1]\!]_s(I_1)\, b_1 \cdots b_n = \mathit{true}$ and $[\![\mathsf{E}_2]\!]_s(I_2)\, b_1 \cdots b_n = \mathit{true}$. Since $\mathcal{I}$ is directed, we get that $I = \bigsqcup\{I_1, I_2\}$ exists in $\mathcal{I}$ and it holds that $[\![\mathsf{E}_1]\!]_s(I)\, b_1 \cdots b_n = \mathit{true}$ and $[\![\mathsf{E}_2]\!]_s(I)\, b_1 \cdots b_n = \mathit{true}$. By the semantics of $\bigwedge_\pi$, $[\![(\mathsf{E}_1 \bigwedge_\pi \mathsf{E}_2)]\!]_s(I)\, b_1 \cdots b_n = \mathit{true}$ and thus $(\bigsqcup_{I\in\mathcal{I}} [\![(\mathsf{E}_1 \bigwedge_\pi \mathsf{E}_2)]\!]_s(I))\, b_1 \cdots b_n = \mathit{true}$.

Case 6: $\mathsf{E} = (\mathsf{E}_1 \approx \mathsf{E}_2)$. It suffices to show that $[\![(\mathsf{E}_1 \approx \mathsf{E}_2)]\!]_s(\bigsqcup \mathcal{I}) \sqsubseteq \bigsqcup_{I\in\mathcal{I}} [\![(\mathsf{E}_1 \approx \mathsf{E}_2)]\!]_s(I)$. This is straightforward since the value of $[\![(\mathsf{E}_1 \approx \mathsf{E}_2)]\!]$ only depends on s (since the expressions $\mathsf{E}_1, \mathsf{E}_2$ do not contain predicate symbols).

Case 7: $\mathsf{E} = (\exists \mathsf{V}\, \mathsf{E}_1)$. We show that $[\![(\exists\, \mathsf{E}_1)]\!]_s(\bigsqcup \mathcal{I}) \sqsubseteq \bigsqcup_{I\in\mathcal{I}} [\![(\exists \mathsf{V}\, \mathsf{E}_1)]\!]_s(I)$ or equivalently that if $[\![(\exists \mathsf{V}\, \mathsf{E}_1)]\!]_s(\bigsqcup \mathcal{I}) = \mathit{true}$ then $\bigsqcup_{I\in\mathcal{I}} [\![(\exists \mathsf{V}\, \mathsf{E}_1)]\!]_s(I) = \mathit{true}$. Notice now that if $[\![(\exists \mathsf{V}\, \mathsf{E}_1)]\!]_s(\bigsqcup \mathcal{I}) = \mathit{true}$ then there exists b such that $[\![\mathsf{E}_1]\!]_{s[b/\mathsf{V}]}(\bigsqcup \mathcal{I}) = \mathit{true}$, which by the induction hypothesis gives $\bigsqcup_{I\in\mathcal{I}} [\![\mathsf{E}_1]\!]_{s[b/\mathsf{V}]}(I) = \mathit{true}$. But this last statement implies that $\bigsqcup_{I\in\mathcal{I}} [\![(\exists \mathsf{V}\, \mathsf{E}_1)]\!]_s(I) = \mathit{true}$. □

E. PROOF OF LEMMA 7.27

Lemma 7.27 Let P be a program, G be a goal and let θ be a substitution. Suppose that there exists an SLD-refutation of $\mathsf{P} \cup \{\mathsf{G}\theta\}$ using substitution σ. Then, there exists an SLD-refutation of $\mathsf{P} \cup \{\mathsf{G}\}$ using a substitution δ, where for some substitution γ it holds that $\delta\gamma \supseteq \theta\sigma$ and $dom(\delta\gamma - \theta\sigma)$ is a (possibly empty) set of template variables that are introduced during the refutation of $\mathsf{P} \cup \{\mathsf{G}\}$.

PROOF. The proof is by induction on the length n of the SLD-refutation of $\mathsf{P} \cup \{\mathsf{G}\theta\}$.

Induction Basis: The basis case is for $n = 1$. We need to distinguish cases based on the structure of G. The most interesting case is $\mathsf{G} = (\mathsf{E}_1 \approx \mathsf{E}_2)$ (the rest are simpler and omitted). By assumption, it holds that $(\mathsf{E}_1\theta \approx \mathsf{E}_2\theta) \xrightarrow{\sigma} \mathsf{true}$, where σ is an mgu of $\mathsf{E}_1\theta$ and $\mathsf{E}_2\theta$. But then we also have that $(\mathsf{E}_1 \approx \mathsf{E}_2) \xrightarrow{\delta} \mathsf{true}$, where δ is an mgu of E_1 and E_2. Since $\theta\sigma$ is a unifier of $\mathsf{E}_1, \mathsf{E}_2$, there exists substitution γ such that $\theta\sigma = \delta\gamma$.

Induction Step: We demonstrate the statement for SLD-refutations of length $n+1$. We distinguish cases based on the structure of G.

Case 1: $\mathsf{G} = \leftarrow (\mathsf{p}\, \mathsf{E}_1 \cdots \mathsf{E}_k)$. Then, $\mathsf{G}\theta = \leftarrow (\mathsf{p}\, (\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta))$. By Definition 7.18 we get that $\mathsf{p}\, (\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta) \xrightarrow{\epsilon} \mathsf{E}\, (\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta)$, where $\mathsf{p} \leftarrow \mathsf{E}$ is a rule in P. By assumption, $\mathsf{E}\, (\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta)$ has an SLD-refutation of length n using σ. Consider now the goal G. By Definition 7.18, we get that $(\mathsf{p}\, \mathsf{E}_1 \cdots \mathsf{E}_k) \xrightarrow{\epsilon} (\mathsf{E}\, \mathsf{E}_1 \cdots \mathsf{E}_k)$. Notice now that since E is a closed lambda expression, it holds that $(\mathsf{E}\, \mathsf{E}_1 \cdots \mathsf{E}_k)\theta = (\mathsf{E}\, (\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta))$. Moreover, since $(\mathsf{E}\, (\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta))$ has an SLD-refutation of length n using σ, we get by the induction hypothesis that $(\mathsf{E}\, \mathsf{E}_1 \cdots \mathsf{E}_k)$ has an SLD-refutation using substitution δ, where for some substitution γ it holds that $\delta\gamma \supseteq \theta\sigma$ and $dom(\delta\gamma - \theta\sigma)$ is a set of template variables that are introduced during the refutation of $(\mathsf{E}\, \mathsf{E}_1 \cdots \mathsf{E}_k)$. But then, $(\mathsf{p}\, \mathsf{E}_1 \cdots \mathsf{E}_k)$ has an SLD-refutation which satisfies the requirements of the lemma.

Case 2: $\mathsf{G} = \leftarrow (\mathsf{Q}\, \mathsf{E}_1 \cdots \mathsf{E}_k)$. Consider first the case where $\theta(\mathsf{Q}) = \mathsf{B}$, for some basic expression B. Then, $\mathsf{G}\theta = \leftarrow (\mathsf{B}\, (\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta))$. Notice now that B can be either a

higher-order predicate variable or a finite-union of lambda abstractions. We examine the case where B is a single lambda abstraction (the other two cases are similar). Since B is a lambda abstraction, assume that $\mathsf{B} = \lambda \mathsf{V}.\mathsf{C}$. By Definition 7.18 we get that $\mathsf{B}\,(\mathsf{E}_1\theta)\cdots(\mathsf{E}_k\theta) \xrightarrow{\epsilon} \mathsf{C}\{\mathsf{V}/(\mathsf{E}_1\theta)\}\,(\mathsf{E}_2\theta)\cdots(\mathsf{E}_k\theta)$. By assumption, $\mathsf{C}\{\mathsf{V}/(\mathsf{E}_1\theta)\}\,(\mathsf{E}_2\theta)\cdots(\mathsf{E}_k\theta)$ has an SLD-refutation of length n using σ. Consider now the goal G. By Definition 7.18, we get that $(\mathsf{Q}\,\mathsf{E}_1\cdots\mathsf{E}_k) \overset{\{\mathsf{Q}/\mathsf{B}_t\}}{\rightarrow} \mathsf{B}_t\,(\mathsf{E}_1\{\mathsf{Q}/\mathsf{B}_t\})\cdots(\mathsf{E}_k\{\mathsf{Q}/\mathsf{B}_t\})$, where $\mathsf{B}_t = \lambda\mathsf{V}.\mathsf{C}_t$, and $\mathsf{B} = \mathsf{B}_t\gamma_1$, for some substitution γ_1 with $dom(\gamma_1) = FV(\mathsf{B}_t)$. We assume without loss of generality that the set $dom(\gamma_1)$ is disjoint from $FV(\mathsf{G})$ and from $dom(\theta) \cup FV(range(\theta))$. By Definition 7.18 we get that $\mathsf{B}_t\,(\mathsf{E}_1\{\mathsf{Q}/\mathsf{B}_t\})\cdots(\mathsf{E}_k\{\mathsf{Q}/\mathsf{B}_t\}) \xrightarrow{\epsilon} \mathsf{C}_t\{\mathsf{V}/\mathsf{E}_1\{\mathsf{Q}/\mathsf{B}_t\}\}\,(\mathsf{E}_2\{\mathsf{Q}/\mathsf{B}_t\})\cdots(\mathsf{E}_k\{\mathsf{Q}/\mathsf{B}_t\})$. Notice now that:

$$((\mathsf{C}_t\{\mathsf{V}/\mathsf{E}_1\{\mathsf{Q}/\mathsf{B}_t\}\})\,(\mathsf{E}_2\{\mathsf{Q}/\mathsf{B}_t\})\cdots(\mathsf{E}_k\{\mathsf{Q}/\mathsf{B}_t\}))\theta\gamma_1 = (\mathsf{C}\{\mathsf{V}/\mathsf{E}_1\theta\})\,(\mathsf{E}_2\theta)\cdots(\mathsf{E}_k\theta)$$

Then, since $(\mathsf{C}\{\mathsf{V}/\mathsf{E}_1\theta\})\,(\mathsf{E}_2\theta)\cdots(\mathsf{E}_k\theta)$ has an SLD-refutation of length n using σ, we get by the induction hypothesis that $(\mathsf{C}_t\{\mathsf{V}/\mathsf{E}_1\{\mathsf{Q}/\mathsf{B}_t\}\})\,(\mathsf{E}_2\{\mathsf{Q}/\mathsf{B}_t\})\cdots(\mathsf{E}_k\{\mathsf{Q}/\mathsf{B}_t\})$ has an SLD-refutation using substitution δ', where for some substitution γ it holds $\delta'\gamma \supseteq \theta\gamma_1\sigma$ and $dom(\delta'\gamma - \theta\gamma_1\sigma)$ is a set of template variables that are introduced during this SLD-refutation. From the above discussion we conclude that $(\mathsf{Q}\,\mathsf{E}_1\cdots\mathsf{E}_k)$ has an SLD-refutation using substitution $\delta = \{\mathsf{Q}/\mathsf{B}_t\}\delta'$. Moreover, it holds that $\delta\gamma = \{\mathsf{Q}/\mathsf{B}_t\}\delta'\gamma \supseteq \{\mathsf{Q}/\mathsf{B}_t\}\theta\gamma_1\sigma \supseteq \theta\sigma$ and $dom(\delta\gamma - \theta\sigma)$ is a set of template variables that are introduced during the refutation of $(\mathsf{Q}\,\mathsf{E}_1\cdots\mathsf{E}_k)$.

Consider now the case where $\theta(\mathsf{Q})$ is undefined. Then, $\mathsf{G}\theta =\leftarrow (\mathsf{Q}\,(\mathsf{E}_1\theta)\cdots(\mathsf{E}_k\theta))$. By Definition 7.18 we get that $\mathsf{Q}\,(\mathsf{E}_1\theta)\cdots(\mathsf{E}_k\theta) \overset{\{\mathsf{Q}/\mathsf{B}_t\}}{\rightarrow} \mathsf{B}_t\,(\mathsf{E}_1\theta\{\mathsf{Q}/\mathsf{B}_t\})\cdots(\mathsf{E}_k\theta\{\mathsf{Q}/\mathsf{B}_t\})$. We may assume without loss of generality that the set $FV(\mathsf{B}_t)$ is disjoint from $FV(\mathsf{G})$ and from $dom(\theta) \cup FV(range(\theta))$. By assumption, $\mathsf{B}_t\,(\mathsf{E}_1\theta\{\mathsf{Q}/\mathsf{B}_t\})\cdots(\mathsf{E}_k\theta\{\mathsf{Q}/\mathsf{B}_t\})$ has an SLD-refutation of length n using σ', where $\sigma = \{\mathsf{Q}/\mathsf{B}_t\}\sigma'$. Consider now the goal G. By Definition 7.18, we get that $(\mathsf{Q}\,\mathsf{E}_1\cdots\mathsf{E}_k) \overset{\{\mathsf{Q}/\mathsf{B}_t\}}{\rightarrow} \mathsf{B}_t\,(\mathsf{E}_1\{\mathsf{Q}/\mathsf{B}_t\})\cdots(\mathsf{E}_k\{\mathsf{Q}/\mathsf{B}_t\})$. Notice now that:

$$(\mathsf{B}_t\,(\mathsf{E}_1\{\mathsf{Q}/\mathsf{B}_t\})\cdots(\mathsf{E}_k\{\mathsf{Q}/\mathsf{B}_t\}))\theta\{\mathsf{Q}/\mathsf{B}_t\} = \mathsf{B}_t\,(\mathsf{E}_1\theta\{\mathsf{Q}/\mathsf{B}_t\})\cdots(\mathsf{E}_k\theta\{\mathsf{Q}/\mathsf{B}_t\})$$

Then, since $\mathsf{B}_t\,(\mathsf{E}_1\theta\{\mathsf{Q}/\mathsf{B}_t\})\cdots(\mathsf{E}_k\theta\{\mathsf{Q}/\mathsf{B}_t\})$ has an SLD-refutation of length n using σ', we get by the induction hypothesis that $\mathsf{B}_t\,(\mathsf{E}_1\{\mathsf{Q}/\mathsf{B}_t\})\cdots(\mathsf{E}_k\{\mathsf{Q}/\mathsf{B}_t\})$ has an SLD-refutation using substitution δ', where for some substitution γ it holds $\delta'\gamma \supseteq \theta\{\mathsf{Q}/\mathsf{B}_t\}\sigma'$ and $dom(\delta'\gamma - \theta\{\mathsf{Q}/\mathsf{B}_t\}\sigma')$ is a set of template variables that are introduced during this SLD-refutation; notice that these template variables can be chosen to be different than the variables in $FV(\mathsf{B}_t)$. From the above discussion we conclude that $(\mathsf{Q}\,\mathsf{E}_1\cdots\mathsf{E}_k)$ has an SLD-refutation using substitution $\delta = \{\mathsf{Q}/\mathsf{B}_t\}\delta'$. Moreover, it holds that $\delta\gamma = \{\mathsf{Q}/\mathsf{B}_t\}\delta'\gamma \supseteq \{\mathsf{Q}/\mathsf{B}_t\}\theta\{\mathsf{Q}/\mathsf{B}_t\}\sigma' = \theta\{\mathsf{Q}/\mathsf{B}_t\}\sigma' = \theta\sigma$ and $dom(\delta\gamma - \theta\sigma)$ is a set of template variables that are introduced during the refutation of $(\mathsf{Q}\,\mathsf{E}_1\cdots\mathsf{E}_k)$.

Case 3: $\mathsf{G} =\leftarrow ((\lambda\mathsf{V}.\mathsf{E})\,\mathsf{E}_1\cdots\mathsf{E}_k)$. Then, $\mathsf{G}\theta =\leftarrow ((\lambda\mathsf{V}.\mathsf{E}\theta)\,(\mathsf{E}_1\theta)\cdots(\mathsf{E}_k\theta))$. By Definition 7.18 we get that $(\lambda\mathsf{V}.\mathsf{E}\theta)\,(\mathsf{E}_1\theta)\cdots(\mathsf{E}_k\theta) \xrightarrow{\epsilon} (\mathsf{E}\theta\{\mathsf{V}/(\mathsf{E}_1\theta)\})\,(\mathsf{E}_2\theta)\cdots(\mathsf{E}_k\theta)$. Moreover, by assumption, $(\mathsf{E}\theta\{\mathsf{V}/(\mathsf{E}_1\theta)\})\,(\mathsf{E}_2\theta)\cdots(\mathsf{E}_k\theta)$ has an SLD-refutation of length n using σ. Consider now the goal G. By Definition 7.18, we get that $(\lambda\mathsf{V}.\mathsf{E})\,\mathsf{E}_1\cdots\mathsf{E}_k \xrightarrow{\epsilon} (\mathsf{E}\{\mathsf{V}/\mathsf{E}_1\})\,\mathsf{E}_2\cdots\mathsf{E}_k$. Notice now that $((\mathsf{E}\{\mathsf{V}/\mathsf{E}_1\})\,\mathsf{E}_2\cdots\mathsf{E}_k)\theta = (\mathsf{E}\theta\{\mathsf{V}/(\mathsf{E}_1\theta)\})\,(\mathsf{E}_2\theta)\cdots(\mathsf{E}_k\theta)$, and since the latter expression has an SLD-refutation of length n using σ, we get by the induction hypothesis that $(\mathsf{E}\{\mathsf{V}/\mathsf{E}_1\})\,\mathsf{E}_2\cdots\mathsf{E}_k$ has an SLD-refutation using a substitution δ, where for some substitution γ it holds $\delta\gamma \supseteq \theta\sigma$ and $dom(\delta\gamma - \theta\sigma)$ is a set of template variables that are introduced during this refutation. But then, $((\lambda\mathsf{V}.\mathsf{E})\,\mathsf{E}_1\cdots\mathsf{E}_k)$ has an SLD-refutation using substitution δ which satisfies the requirements of the lemma.

Case 4: $\mathsf{G} =\!\leftarrow ((\mathsf{E}' \bigvee_\pi \mathsf{E}'')\,\mathsf{E}_1 \cdots \mathsf{E}_k)$. Then, $\mathsf{G}\theta =\!\leftarrow ((\mathsf{E}'\theta \bigvee_\pi \mathsf{E}''\theta)\,(\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta))$. By Definition 7.18 we get that $(\mathsf{E}'\theta \bigvee_\pi \mathsf{E}''\theta)\,(\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta) \xrightarrow{\epsilon} (\mathsf{E}'\theta)\,(\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta)$ (and symmetrically for E''). By assumption, either $(\mathsf{E}'\theta)\,(\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta)$ or $(\mathsf{E}''\theta)\,(\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta)$ has an SLD-refutation of length n using σ. Assume, without loss of generality, that $(\mathsf{E}'\theta)\,(\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta)$ has an SLD-refutation of length n using σ. Consider now the goal G. By Definition 7.18, we get that $(\mathsf{E}' \bigvee_\pi \mathsf{E}'')\,\mathsf{E}_1 \cdots \mathsf{E}_k \xrightarrow{\epsilon} \mathsf{E}'\,\mathsf{E}_1 \cdots \mathsf{E}_k$. Notice now that $(\mathsf{E}'\,\mathsf{E}_1 \cdots \mathsf{E}_k)\theta = (\mathsf{E}'\theta)\,(\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta)$, and since the latter expression has an SLD-refutation of length n using σ, we get by the induction hypothesis that $\mathsf{E}'\,\mathsf{E}_1 \cdots \mathsf{E}_k$ has an SLD-refutation using a substitution δ, where for some substitution γ it holds $\delta\gamma \supseteq \theta\sigma$ and $dom(\delta\gamma - \theta\sigma)$ is a set of template variables that are introduced during this refutation. But then, $(\mathsf{E}' \bigvee_\pi \mathsf{E}'')\,\mathsf{E}_1 \cdots \mathsf{E}_k$ has an SLD-refutation using substitution δ which satisfies the requirements of the lemma.

Case 5: $\mathsf{G} =\!\leftarrow ((\mathsf{E}' \bigwedge_\pi \mathsf{E}'')\,\mathsf{E}_1 \cdots \mathsf{E}_k)$. Then, $\mathsf{G}\theta =\!\leftarrow ((\mathsf{E}'\theta \bigwedge_\pi \mathsf{E}''\theta)\,(\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta))$. By Definition 7.18 we get $(\mathsf{E}'\theta \bigvee_\pi \mathsf{E}''\theta)\,(\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta) \xrightarrow{\epsilon} ((\mathsf{E}'\theta)\,(\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta)) \wedge ((\mathsf{E}''\theta)\,(\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta))$. By assumption, $((\mathsf{E}'\theta)\,(\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta)) \wedge ((\mathsf{E}''\theta)\,(\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta))$ has an SLD-refutation of length n using σ. Consider now the goal G. By Definition 7.18, we get that $(\mathsf{E}' \bigwedge_\pi \mathsf{E}'')\,\mathsf{E}_1 \cdots \mathsf{E}_k \xrightarrow{\epsilon} (\mathsf{E}'\,\mathsf{E}_1 \cdots \mathsf{E}_k) \wedge (\mathsf{E}''\,\mathsf{E}_1 \cdots \mathsf{E}_k)$. Notice now that it holds that $((\mathsf{E}'\,\mathsf{E}_1 \cdots \mathsf{E}_k) \wedge (\mathsf{E}''\,\mathsf{E}_1 \cdots \mathsf{E}_k))\theta = ((\mathsf{E}'\theta)\,(\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta)) \wedge ((\mathsf{E}''\theta)\,(\mathsf{E}_1\theta) \cdots (\mathsf{E}_k\theta))$; since the latter expression has an SLD-refutation of length n using σ, we get by the induction hypothesis that $(\mathsf{E}'\,\mathsf{E}_1 \cdots \mathsf{E}_k) \wedge (\mathsf{E}''\,\mathsf{E}_1 \cdots \mathsf{E}_k)$ has an SLD-refutation using a substitution δ, where for some substitution γ it holds $\delta\gamma \supseteq \theta\sigma$ and $dom(\delta\gamma - \theta\sigma)$ is a set of template variables that are introduced during this refutation. But then, $(\mathsf{E}' \bigwedge_\pi \mathsf{E}'')\,\mathsf{E}_1 \cdots \mathsf{E}_k$ has an SLD-refutation using substitution δ which satisfies the requirements of the lemma.

Case 6: $\mathsf{G} =\!\leftarrow (\mathsf{true} \wedge \mathsf{E})$. Then, $\mathsf{G}\theta =\!\leftarrow (\mathsf{true} \wedge (\mathsf{E}\theta))$. By Definition 7.18 we get $(\mathsf{true} \wedge (\mathsf{E}\theta)) \xrightarrow{\epsilon} \mathsf{E}\theta$. By assumption, $\mathsf{E}\theta$ has an SLD-refutation of length n using σ. Consider now the goal G. By Definition 7.18, we get that $(\mathsf{true} \wedge \mathsf{E}) \xrightarrow{\epsilon} \mathsf{E}$. Since $\mathsf{E}\theta$ has an SLD-refutation of length n using σ, we get by the induction hypothesis that E has an SLD-refutation using a substitution δ, where for some substitution γ it holds $\delta\gamma \supseteq \theta\sigma$ and $dom(\delta\gamma - \theta\sigma)$ is a set of template variables that are introduced during this refutation. But then, $(\mathsf{true} \wedge \mathsf{E})$ has an SLD-refutation using substitution δ which satisfies the requirements of the lemma.

Case 7: $\mathsf{G} =\!\leftarrow (\mathsf{E} \wedge \mathsf{true})$. Almost identical to the previous case.

Case 8: $\mathsf{G} =\!\leftarrow (\exists\mathsf{V}\,\mathsf{E})$. Then, $\mathsf{G}\theta =\!\leftarrow (\exists\mathsf{V}\,(\mathsf{E}\theta))$. By Definition 7.18 we get $(\exists\mathsf{V}\,(\mathsf{E}\theta)) \xrightarrow{\epsilon} \mathsf{E}\theta$. By assumption, $\mathsf{E}\theta$ has an SLD-refutation of length n using σ. Consider now the goal G. By Definition 7.18, we get that $(\exists\mathsf{V}\,\mathsf{E}) \xrightarrow{\epsilon} \mathsf{E}$. Since $\mathsf{E}\theta$ has an SLD-refutation of length n using σ, we get by the induction hypothesis that E has an SLD-refutation using a substitution δ, where for some substitution γ it holds $\delta\gamma \supseteq \theta\sigma$ and $dom(\delta\gamma - \theta\sigma)$ is a set of template variables that are introduced during this refutation. But then, $(\exists\mathsf{V}\,\mathsf{E})$ has an SLD-refutation using substitution δ which satisfies the requirements of the lemma.

Case 9: $\mathsf{G} =\!\leftarrow (\mathsf{E}_1 \wedge \mathsf{E}_2)$. We may assume without loss of generality that given the goal $\mathsf{G}\theta =\!\leftarrow (\mathsf{E}_1\theta \wedge \mathsf{E}_2\theta)$, the first step in the refutation will take place due to the subexpression $\mathsf{E}_1\theta$. Moreover, again without loss of generality, due to the associativity of $\wedge$, we assume that E_1 is not an expression that contains a top-level $\wedge$ (ie., it is not of the form $\mathsf{A}_1 \wedge \mathsf{A}_2$). The proof could be easily adapted to circumvent the two assumptions just mentioned (but this would result in more cumbersome notation). We perform a case analysis on E_1:

Subcase 9.1: $\mathsf{E}_1 = (\mathsf{A}_1 \approx \mathsf{A}_2)$. Then, we have $((\mathsf{A}_1 \approx \mathsf{A}_2)\theta \wedge \mathsf{E}_2\theta) \stackrel{\sigma_1}{\rightarrow} (\mathsf{true} \wedge \mathsf{E}_2\theta\sigma_1)$, where σ_1 is an mgu of $\mathsf{A}_1\theta$ and $\mathsf{A}_2\theta$. By assumption, $(\mathsf{true} \wedge \mathsf{E}_2\theta\sigma_1)$ has an SLD-refutation of length n using σ', where $\sigma = \sigma_1\sigma'$. Consider now $(\mathsf{E}_1 \wedge \mathsf{E}_2)$. By Definition 7.18, it holds that $(\mathsf{A}_1 \approx \mathsf{A}_2) \stackrel{\delta_1}{\rightarrow} \mathsf{true}$, where δ_1 is an mgu of $\mathsf{A}_1, \mathsf{A}_2$. By Definition 7.18 we get that $((\mathsf{A}_1 \approx \mathsf{A}_2) \wedge \mathsf{E}_2) \stackrel{\delta_1}{\rightarrow} (\mathsf{true} \wedge \mathsf{E}_2\delta_1)$. Since $\theta\sigma_1$ is a unifier of $\mathsf{A}_1, \mathsf{A}_2$, there exists θ' such that $\theta\sigma_1 = \delta_1\theta'$, and since $(\mathsf{true} \wedge \mathsf{E}_2\theta\sigma_1)$ has an SLD-refutation of length n using σ', we get that $(\mathsf{true} \wedge \mathsf{E}_2\delta_1\theta') = (\mathsf{true} \wedge \mathsf{E}_2\delta_1)\theta'$ has an SLD-refutation of length n using σ'. By the induction hypothesis we get that $(\mathsf{true} \wedge \mathsf{E}_2\delta_1)$ has an SLD-refutation using δ', where $\delta'\gamma \supseteq \theta'\sigma'$ and $dom(\delta'\gamma - \theta'\sigma')$ is a set of template variables that are introduced during the refutation of this goal. But then, $(\mathsf{E}_1 \wedge \mathsf{E}_2)$ has an SLD-refutation using substitution $\delta = \delta_1\delta'$. Moreover, it holds that $\delta\gamma = \delta_1\delta'\gamma \supseteq \delta_1\theta'\sigma' = \theta\sigma_1\sigma' = \theta\sigma$.

Subcase 9.2: $\mathsf{E}_1 = (\mathsf{Q}\,\mathsf{A}_1 \cdots \mathsf{A}_r)$. Consider first the case where $\theta(\mathsf{Q}) = \mathsf{B}$, for some basic expression B. Notice now that B can be either a higher-order predicate variable or a finite-union of lambda abstractions. We examine the case where B is a single lambda abstraction (the other two cases are similar). Since B is a lambda abstraction, we have that $\mathsf{E}_1\theta \stackrel{\epsilon}{\rightarrow} \mathsf{E}'_1$, where E'_1 is the resulting expression after performing the outer beta reduction in $\mathsf{E}_1\theta$. By Definition 7.18 we have that $(\mathsf{E}_1 \wedge \mathsf{E}_2)\theta \stackrel{\epsilon}{\rightarrow} \mathsf{E}'_1 \wedge \mathsf{E}_2\theta$. By assumption, $\mathsf{E}'_1 \wedge \mathsf{E}_2\theta$ has an SLD-refutation of length n using σ. Consider now $(\mathsf{E}_1 \wedge \mathsf{E}_2)$. By Definition 7.18, it holds that $\mathsf{E}_1 \stackrel{\{\mathsf{Q}/\mathsf{B}_t\}}{\rightarrow} \mathsf{E}''_1$, where $\mathsf{E}''_1 = \mathsf{E}_1\{\mathsf{Q}/\mathsf{B}_t\}$ and $\mathsf{B} = \mathsf{B}_t\gamma_1$, for some substitution γ_1, with $dom(\gamma_1) = FV(\mathsf{B}_t)$. We may assume without loss of generality that the set $dom(\gamma_1)$ is disjoint from $FV(\mathsf{G})$ and from $dom(\theta) \cup FV(range(\theta))$. By Definition 7.18, we also get that $\mathsf{E}''_1 \stackrel{\epsilon}{\rightarrow} \mathsf{E}'''_1$, where E'''_1 is the expression that results after performing the outer beta reduction in E''_1. Then, by Definition 7.18 we get that $\mathsf{E}_1 \wedge \mathsf{E}_2 \stackrel{\{\mathsf{Q}/\mathsf{B}_t\}}{\rightarrow} \mathsf{E}''_1 \wedge \mathsf{E}_2\{\mathsf{Q}/\mathsf{B}_t\}$ and $\mathsf{E}''_1 \wedge \mathsf{E}_2\{\mathsf{Q}/\mathsf{B}_t\} \stackrel{\epsilon}{\rightarrow} \mathsf{E}'''_1 \wedge \mathsf{E}_2\{\mathsf{Q}/\mathsf{B}_t\}$. Notice now that $(\mathsf{E}'''_1 \wedge \mathsf{E}_2\{\mathsf{Q}/\mathsf{B}_t\})\theta\gamma_1 = \mathsf{E}'_1 \wedge \mathsf{E}_2\theta$, and since $\mathsf{E}'_1 \wedge \mathsf{E}_2\theta$ has an SLD-refutation of length n using σ, we get by the induction hypothesis that $(\mathsf{E}'''_1 \wedge \mathsf{E}_2\{\mathsf{Q}/\mathsf{B}_t\})$ has an SLD-refutation using δ', where for some substitution γ it holds $\delta'\gamma \supseteq \theta\gamma_1\sigma$ and $dom(\delta'\gamma - \theta\gamma_1\sigma)$ is a set of template variables that are introduced during this SLD-refutation. But then, $\mathsf{E}_1 \wedge \mathsf{E}_2$ has an SLD-refutation using substitution $\delta = \{\mathsf{Q}/\mathsf{B}_t\}\delta'$. Moreover, it holds that $\delta\gamma = \{\mathsf{Q}/\mathsf{B}_t\}\delta'\gamma \supseteq \{\mathsf{Q}/\mathsf{B}_t\}\theta\gamma_1\sigma \supseteq \theta\sigma$ and $dom(\delta\gamma - \theta\sigma)$ is a set of template variables that are introduced during the refutation of G.

Consider now the case where $\theta(\mathsf{Q})$ is undefined, ie., there does not exist a binding for Q in θ. Then, we have that $\mathsf{E}_1\theta \stackrel{\{\mathsf{Q}/\mathsf{B}_t\}}{\rightarrow} \mathsf{E}'_1$, where $\mathsf{E}'_1 = \mathsf{B}_t\,(\mathsf{A}_1\theta\{\mathsf{Q}/\mathsf{B}_t\}) \cdots (\mathsf{A}_r\theta\{\mathsf{Q}/\mathsf{B}_t\})$. We may assume without loss of generality that the set $FV(\mathsf{B}_t)$ is disjoint from $FV(\mathsf{G})$ and from $dom(\theta) \cup FV(range(\theta))$. By Definition 7.18 we have that $(\mathsf{E}_1 \wedge \mathsf{E}_2)\theta \stackrel{\{\mathsf{Q}/\mathsf{B}_t\}}{\rightarrow} \mathsf{E}'_1 \wedge (\mathsf{E}_2\theta\{\mathsf{Q}/\mathsf{B}_t\})$. By assumption, $\mathsf{E}'_1 \wedge (\mathsf{E}_2\theta\{\mathsf{Q}/\mathsf{B}_t\})$ has an SLD-refutation of length n using σ', where $\sigma = \{\mathsf{Q}/\mathsf{B}_t\}\sigma'$. Consider now $(\mathsf{E}_1 \wedge \mathsf{E}_2)$. By Definition 7.18, it holds that $\mathsf{E}_1 \stackrel{\{\mathsf{Q}/\mathsf{B}_t\}}{\rightarrow} \mathsf{E}''_1$, where $\mathsf{E}''_1 = \mathsf{E}_1\{\mathsf{Q}/\mathsf{B}_t\}$. By Definition 7.18 we get that $\mathsf{E}_1 \wedge \mathsf{E}_2 \stackrel{\{\mathsf{Q}/\mathsf{B}_t\}}{\rightarrow} \mathsf{E}''_1 \wedge \mathsf{E}_2\{\mathsf{Q}/\mathsf{B}_t\}$. Notice now that $(\mathsf{E}''_1 \wedge \mathsf{E}_2\{\mathsf{Q}/\mathsf{B}_t\})\theta\{\mathsf{Q}/\mathsf{B}_t\} = \mathsf{E}'_1 \wedge \mathsf{E}_2\theta\{\mathsf{Q}/\mathsf{B}_t\}$, and since $\mathsf{E}'_1 \wedge \mathsf{E}_2\theta\{\mathsf{Q}/\mathsf{B}_t\}$ has an SLD-refutation of length n using σ', we get by the induction hypothesis that $(\mathsf{E}''_1 \wedge \mathsf{E}_2\{\mathsf{Q}/\mathsf{B}_t\})$ has an SLD-refutation using δ', where for some substitution γ it holds that $\delta'\gamma \supseteq \theta\{\mathsf{Q}/\mathsf{B}_t\}\sigma'$, and $dom(\delta'\gamma - \theta\{\mathsf{Q}/\mathsf{B}_t\}\sigma')$ is a set of template variables that are introduced during this SLD-refutation; notice that these template variables can be chosen to be different than the variables in $FV(\mathsf{B}_t)$. Then, $\mathsf{E}_1 \wedge \mathsf{E}_2$ has an SLD-refutation using substitution $\delta = \{\mathsf{Q}/\mathsf{B}_t\}\delta'$. Moreover, it holds that $\delta\gamma = \{\mathsf{Q}/\mathsf{B}_t\}\delta'\gamma \supseteq \{\mathsf{Q}/\mathsf{B}_t\}\theta\{\mathsf{Q}/\mathsf{B}_t\}\sigma' = \theta\{\mathsf{Q}/\mathsf{B}_t\}\sigma' = \theta\sigma$ and $dom(\delta\gamma - \theta\sigma)$ is a set of template variables that are introduced during the refutation of G.

Subcase 9.3: $\mathsf{E}_1 = \leftarrow ((\mathsf{A}' \bigvee_\pi \mathsf{A}'')\, \mathsf{A}_1 \cdots \mathsf{A}_r)$. Then, $\mathsf{E}_1\theta = (\mathsf{A}'\theta \bigvee_\pi \mathsf{A}''\theta)\,(\mathsf{A}_1\theta) \cdots (\mathsf{A}_r\theta)$. By Definition 7.18 we get that $(\mathsf{A}'\theta \bigvee_\pi \mathsf{A}''\theta)\,(\mathsf{A}_1\theta) \cdots (\mathsf{A}_r\theta) \xrightarrow{\epsilon} (\mathsf{A}'\theta)\,(\mathsf{A}_1\theta) \cdots (\mathsf{A}_r\theta)$ (and symmetrically for A''). By Definition 7.18 we have $(\mathsf{E}_1\theta \wedge \mathsf{E}_2\theta) \xrightarrow{\epsilon} ((\mathsf{A}'\theta)\,(\mathsf{A}_1\theta) \cdots (\mathsf{A}_r\theta)) \wedge \mathsf{E}_2\theta$ and $(\mathsf{E}_1\theta \wedge \mathsf{E}_2\theta) \xrightarrow{\epsilon} ((\mathsf{A}''\theta)\,(\mathsf{A}_1\theta) \cdots (\mathsf{A}_r\theta)) \wedge \mathsf{E}_2\theta$. By assumption, either $((\mathsf{A}'\theta)\,(\mathsf{A}_1\theta) \cdots (\mathsf{A}_r\theta)) \wedge \mathsf{E}_2\theta$ or $((\mathsf{A}''\theta)\,(\mathsf{A}_1\theta) \cdots (\mathsf{A}_r\theta)) \wedge \mathsf{E}_2\theta$ has an SLD-refutation of length n using σ. Assume, without loss of generality, that $((\mathsf{A}'\theta)\,(\mathsf{A}_1\theta) \cdots (\mathsf{A}_r\theta)) \wedge \mathsf{E}_2\theta$ has an SLD-refutation of length n using σ. Notice now that by Definition 7.18, we have that $(\mathsf{A}' \bigvee_\pi \mathsf{A}'')\,\mathsf{A}_1 \cdots \mathsf{A}_r \xrightarrow{\epsilon} \mathsf{A}'\,\mathsf{A}_1 \cdots \mathsf{A}_r$. Moreover, notice that $((\mathsf{A}'\,\mathsf{A}_1 \cdots \mathsf{A}_r) \wedge \mathsf{E}_2)\theta = ((\mathsf{A}'\theta)\,(\mathsf{A}_1\theta) \cdots (\mathsf{A}_r\theta)) \wedge \mathsf{E}_2\theta$, and since the latter expression has an SLD-refutation of length n using σ, we get by the induction hypothesis that $((\mathsf{A}'\,\mathsf{A}_1 \cdots \mathsf{A}_r) \wedge \mathsf{E}_2)$ has an SLD-refutation using a substitution δ, where for some substitution γ it holds $\delta\gamma \supseteq \theta\sigma$ and $dom(\delta\gamma - \theta\sigma)$ is a set of template variables that are introduced during this refutation. But then, $((\mathsf{A}' \bigvee_\pi \mathsf{A}'')\,\mathsf{A}_1 \cdots \mathsf{A}_r) \wedge \mathsf{E}_2$ has an SLD-refutation using substitution δ which satisfies the requirements of the lemma.

Subcase 9.4: E_1 has any other form except for the ones examined in the previous three subcases. Then, it can be verified that in all these subcases it holds that $\mathsf{E}_1\theta \xrightarrow{\epsilon} \mathsf{E}'_1$, for some E'_1. By Definition 7.18 we have that $(\mathsf{E}_1 \wedge \mathsf{E}_2)\theta \xrightarrow{\epsilon} \mathsf{E}'_1 \wedge (\mathsf{E}_2\theta)$. By assumption, $\mathsf{E}'_1 \wedge (\mathsf{E}_2\theta)$ has an SLD-refutation of length n using σ. Consider now $(\mathsf{E}_1 \wedge \mathsf{E}_2)$. By Definition 7.18 and by examination of all the possible cases for E_1 it can be seen that $\mathsf{E}_1 \xrightarrow{\epsilon} \mathsf{E}''_1$, where $\mathsf{E}'_1 = \mathsf{E}''_1\theta$. By Definition 7.18 we get that $\mathsf{E}_1 \wedge \mathsf{E}_2 \xrightarrow{\epsilon} \mathsf{E}''_1 \wedge \mathsf{E}_2$. Notice now that $(\mathsf{E}''_1 \wedge \mathsf{E}_2)\theta = \mathsf{E}'_1 \wedge \mathsf{E}_2\theta$, and since $\mathsf{E}'_1 \wedge \mathsf{E}_2\theta$ has an SLD-refutation of length n using σ, we get by the induction hypothesis that $(\mathsf{E}''_1 \wedge \mathsf{E}_2)$ has an SLD-refutation using δ, where for some substitution γ it holds that $\delta\gamma \supseteq \theta\sigma$, and $dom(\delta\gamma - \theta\sigma)$ is a set of template variables that are introduced during this SLD-refutation. Then, $\mathsf{E}_1 \wedge \mathsf{E}_2$ has an SLD-refutation using substitution δ which satisfies the requirements of the lemma. □

F. PROOF OF LEMMA 7.35

Lemma 7.35 Let P be a program and $\mathsf{G} = \leftarrow \mathsf{A}$ be a goal such that $[\![\mathsf{A}]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = \mathit{true}$ for all Herbrand states s. Then, there exists an SLD-refutation for $\mathsf{P} \cup \{\mathsf{G}\}$ with computed answer equal to the identity substitution.

PROOF. We establish a stronger statement which has the statement of the lemma as a special case. Let us call a substitution θ *closed* if every expression in $range(\theta)$ is closed. We demonstrate that for every closed basic substitution θ, if $[\![\mathsf{A}\theta]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = \mathit{true}$ for all Herbrand states s, then there exists an SLD-refutation for $\mathsf{P} \cup \{\mathsf{A}\theta\}$ with computed answer equal to the identity substitution. The statement of the lemma is then a direct consequence for $\theta = \epsilon$.

We start by noting that A is always of the form $(\mathsf{A}_0\ \mathsf{A}_1 \cdots \mathsf{A}_n)$, $n \geq 0$ (if $n = 0$ then A_0 is of type o). We perform induction on the type $\rho_1 \to \cdots \rho_n \to o$ of A_0.

Outer Induction Basis: The outer induction basis is for $n = 0$, ie., for $type(\mathsf{A}_0) = o$, and in order to establish it we need to perform an inner structural induction on A_0.

Inner Induction Basis: For the inner induction basis we need to examine the cases where A_0 is false, true, $(\mathsf{E}_1 \approx \mathsf{E}_2)$ and Q, where $type(\mathsf{Q}) = o$. The first case is not applicable since $[\![\mathsf{false}\theta]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = \mathit{false}$. The second case is immediate. We examine the latter two cases:

Case 1: $\mathsf{A}_0 = (\mathsf{E}_1 \approx \mathsf{E}_2)$. Since for all s it holds that $[\![(\mathsf{E}_1 \approx \mathsf{E}_2)\theta]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = \mathit{true}$, we get that for all s, $[\![\mathsf{E}_1\theta]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = [\![\mathsf{E}_2\theta]\!]_s(\bot_{\mathcal{I}_\mathsf{P}})$. By the fact that $\bot_{\mathcal{I}_\mathsf{P}}$ is a Herbrand interpretation, we conclude that $\mathsf{E}_1\theta$ and $\mathsf{E}_2\theta$ must be identical expressions of type ι, and therefore they are unifiable using the identity substitution.

Case 2: $\mathsf{A}_0 = \mathsf{Q}$, with $type(\mathsf{Q}) = o$. If $\theta(\mathsf{Q}) = false$ then it can not be the case that $[\![\mathsf{A}_0]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$, and therefore this case is not applicable. If $\theta(\mathsf{Q}) = true$, the result is trivial. If on the other hand $\mathsf{Q} \notin dom(\theta)$, then this case is not applicable since it is not possible to have $[\![\mathsf{A}_0\theta]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$, for all s (eg. choose s such that $s(\mathsf{Q}) = false$).

Inner Induction Step: We distinguish the following cases:

Case 1: $\mathsf{A}_0 = (\exists \mathsf{Q}\, \mathsf{E})$. We can assume without loss of generality that $\mathsf{Q} \notin dom(\theta)$. Since for all s it holds that $[\![(\exists \mathsf{Q}\, \mathsf{E})\theta]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$, it follows that $[\![\mathsf{E}\theta]\!]_{s[b/\mathsf{Q}]}(\bot_{\mathcal{I}_\mathsf{P}}) = true$ for some $b \in \mathcal{F}_{U_\mathsf{P}}(type(\mathsf{Q}))$. Let $\theta' = \{\mathsf{Q}/\mathsf{B}\}$ where B is a closed basic expression such that $[\![\mathsf{B}]\!](\bot_{\mathcal{I}_\mathsf{P}}) = b$ (the existence of such an expression B is ensured by Lemma 7.6). Then it is easy to see that $[\![\mathsf{E}\theta\theta']\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$ for all states s. By the induction hypothesis there exists an SLD-refutation for $\mathsf{P} \cup \{\mathsf{E}\theta\theta'\}$ using some substitution σ and with computed answer equal to the identity substitution. Using Lemma 7.27, it follows that there exists an SLD-refutation of $\mathsf{P} \cup \{\mathsf{E}\theta\}$ using substitution δ where for some substitution γ it holds that $\delta\gamma \supseteq \{\mathsf{Q}/\mathsf{B}\}\sigma$; moreover, $dom(\delta\gamma - \{\mathsf{Q}/\mathsf{B}\}\sigma)$ is a set of template variables that are introduced during the refutation of $\mathsf{P} \cup \{\mathsf{E}\theta\}$. Since the restriction of σ to the free variables of $\mathsf{E}\theta\theta'$ is the identity substitution, it follows that the restriction of δ to the free variables of $\mathsf{E}\theta$ will either be empty or it will only contain the binding Q/B. We conclude that there exists a refutation of $\mathsf{P} \cup \{(\exists \mathsf{Q}\, \mathsf{E})\theta\}$ using substitution $\epsilon\delta = \delta$. The computed answer is the identity substitution since Q is not a free variable of $(\exists \mathsf{Q}\, \mathsf{E})\theta$.

Case 2: $\mathsf{A}_0 = (\mathsf{E}_1 \wedge \mathsf{E}_2)$. By assumption, $[\![(\mathsf{E}_1 \wedge \mathsf{E}_2)\theta]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$, for all s. Then, it holds $[\![\mathsf{E}_1\theta]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$ and $[\![\mathsf{E}_2\theta]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$. By the induction hypothesis there exist SLD-refutations for $\mathsf{P} \cup \{\leftarrow \mathsf{E}_1\theta\}$ and $\mathsf{P} \cup \{\leftarrow \mathsf{E}_2\theta\}$ with computed answers equal to the identity substitution. Let θ_1 and θ_2 be the compositions of the substitutions used for the refutations of $\mathsf{P} \cup \{\leftarrow \mathsf{E}_1\theta\}$ and $\mathsf{P} \cup \{\leftarrow \mathsf{E}_2\theta\}$ respectively. Now, since the computed answer of the refutation for $\mathsf{P} \cup \{\mathsf{E}_1\theta\}$ is the identity, this implies that the free variables of $\mathsf{E}_2\theta$ that also appear free in $\mathsf{E}_1\theta$ do not belong to $dom(\theta_1)$. Moreover, the rest of the free variables of $\mathsf{E}_2\theta$ do not belong to $dom(\theta_1)$, because the variables of θ_1 have been obtained by using resolution steps that only involve "fresh" variables. In conclusion, the restriction of θ_1 to the free variables of $(\mathsf{E}_1 \wedge \mathsf{E}_2)\theta$ is the identity substitution (and similarly for θ_2). These observations imply that $\mathsf{E}_2\theta\theta_1 = \mathsf{E}_2\theta$. But then, we can construct a refutation for $\mathsf{P} \cup \{\leftarrow (\mathsf{E}_1\theta \wedge \mathsf{E}_2\theta)\}$ by first deriving true from $\mathsf{E}_1\theta$ and then deriving true from $\mathsf{E}_2\theta\theta_1 = \mathsf{E}_2\theta$. The substitution used for the refutation of $\mathsf{P} \cup \{\leftarrow (\mathsf{E}_1 \wedge \mathsf{E}_2)\theta\}$ is $\theta_1\theta_2$ and the computed answer is equal to the restriction of $\theta_1\theta_2$ to the free variables of $(\mathsf{E}_1 \wedge \mathsf{E}_2)\theta$, which gives the identity substitution.

Case 3: $\mathsf{A}_0 = (\mathsf{E}_1 \vee \mathsf{E}_2)$. By assumption, $[\![(\mathsf{E}_1 \vee \mathsf{E}_2)\theta]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$, for all s. Then, it either holds that $[\![\mathsf{E}_1\theta]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$ or $[\![\mathsf{E}_2\theta]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$. Without loss of generality, assume that $[\![\mathsf{E}_1\theta]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$. By the induction hypothesis there exists an SLD-refutation for $\mathsf{P} \cup \{\leftarrow \mathsf{E}_1\theta\}$ with computed answer equal to the identity substitution. But then $\mathsf{P} \cup \{\leftarrow (\mathsf{E}_1\theta \vee \mathsf{E}_2\theta)\}$ has an SLD-refutation whose first step leads to $\leftarrow (\mathsf{E}_1\theta)$ using ϵ and then proceeds according to the SLD-refutation of $\leftarrow (\mathsf{E}_1\theta)$. The computed answer of this refutation is obviously the identity substitution.

Outer Induction Step: Assume the lemma holds when A_0 has type $\rho_1 \rightarrow \cdots \rho_{n-1} \rightarrow o$, $n \geq 1$. We establish the lemma for the case where A_0 has type $\pi = \rho_1 \rightarrow \cdots \rho_n \rightarrow o$. We distinguish the following cases:

Case 1: $\mathsf{A}_0 = \mathsf{p}$ (ie., $\mathsf{A} = \mathsf{p}\,\mathsf{A}_1 \cdots \mathsf{A}_n$). This case is not applicable since $\bot_{\mathcal{I}_\mathsf{P}}(\mathsf{p}) = \bot_\pi$ and therefore $[\![\mathsf{A}]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = false$, for all s.

Case 2: $\mathsf{A}_0 = \mathsf{Q}$ (ie., $\mathsf{A} = \mathsf{Q}\,\mathsf{A}_1 \cdots \mathsf{A}_n$). If $\mathsf{Q} \notin dom(\theta)$ then this case is not applicable since it is not possible to have $[\![\mathsf{A}]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$, for all s (eg. take $s(\mathsf{Q}) = \bot_\pi$). If on the other hand $\mathsf{Q} \in dom(\theta)$, then $\theta(\mathsf{Q})$ is a basic expression of type π, ie., it is a non-empty finite union of lambda abstractions. We demonstrate the case where $\theta(\mathsf{Q})$ is a single lambda abstraction; the more general case is similar and omitted. Assume therefore that $\theta(\mathsf{Q}) = \lambda\mathsf{V}.\mathsf{E}$. Then, since $[\![(\lambda\mathsf{V}.\mathsf{E})\,(\mathsf{A}_1\theta)\cdots(\mathsf{A}_n\theta)]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$, by Lemma 7.14 we get that $[\![(\mathsf{E}\{\mathsf{V}/\mathsf{A}_1\theta\})\,(\mathsf{A}_2\theta)\cdots(\mathsf{A}_n\theta)]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$. By assumption, θ is a closed substitution and therefore the only free variable that appears in E is V. Therefore, $[\![((\mathsf{E}\{\mathsf{V}/\mathsf{A}_1\})\,\mathsf{A}_2\cdots\mathsf{A}_n)\theta]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$. By the outer induction hypothesis we get that $\mathsf{P} \cup \{\leftarrow ((\mathsf{E}\{\mathsf{V}/\mathsf{A}_1\})\,\mathsf{A}_2\cdots\mathsf{A}_n)\theta\}$ has an SLD-refutation using substitution δ, with computed answer equal to the identity substitution. By the definition of SLD-resolution we get that $\mathsf{P} \cup \{\leftarrow (\lambda\mathsf{V}.\mathsf{E})\,(\mathsf{A}_1\theta)\cdots(\mathsf{A}_n\theta))\}$ has an SLD-refutation using the substitution $\epsilon\delta = \delta$; the computed answer of this refutation is the restriction of δ to the free variables of $((\lambda\mathsf{V}.\mathsf{E})\,\mathsf{A}_1\cdots\mathsf{A}_n)\theta$ which (by our previous discussion) gives the identity substitution.

Case 3: $\mathsf{A}_0 = \lambda\mathsf{V}.\mathsf{E}$ (ie., $\mathsf{A} = (\lambda\mathsf{V}.\mathsf{E})\,\mathsf{A}_1\cdots\mathsf{A}_n$. We can assume without loss of generality that $\mathsf{V} \notin dom(\theta) \cup FV(range(\theta))$. Then, since $[\![(\lambda\mathsf{V}.\mathsf{E}\theta)\,(\mathsf{A}_1\theta)\cdots(\mathsf{A}_n\theta)]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$, by Lemma 7.14 we get that $[\![(\mathsf{E}\theta\{\mathsf{V}/\mathsf{A}_1\theta\})\,(\mathsf{A}_2\theta)\cdots(\mathsf{A}_n\theta)]\!]_s(\bot_{\mathcal{I}_\mathsf{P}}) = true$. By the outer induction hypothesis $\mathsf{P} \cup \{\leftarrow (\mathsf{E}\theta\{\mathsf{V}/\mathsf{A}_1\theta\})\,(\mathsf{A}_2\theta)\cdots(\mathsf{A}_n\theta)\}$ has an SLD-refutation using substitution δ, with computed answer equal to the identity substitution. By the definition of SLD-resolution we get that $\mathsf{P} \cup \{\leftarrow ((\lambda\mathsf{V}.\mathsf{E}\theta)\,(\mathsf{A}_1\theta)\cdots(\mathsf{A}_n\theta))\}$ has an SLD-refutation using the substitution $\epsilon\delta = \delta$; the computed answer of this refutation is the restriction of δ to the free variables of $((\lambda\mathsf{V}.\mathsf{E})\,\mathsf{A}_1\cdots\mathsf{A}_n)\theta$ which (by our previous discussion) gives the identity substitution.

Case 4: $\mathsf{A}_0 = (\mathsf{E}' \bigvee_\pi \mathsf{E}'')$ (ie., $\mathsf{A} = (\mathsf{E}' \bigvee_\pi \mathsf{E}'')\,\mathsf{A}_1\cdots\mathsf{A}_n$), where $\pi \neq o$. The proof for this case follows easily using the outer induction hypothesis.

Case 5: $\mathsf{A}_0 = (\mathsf{E}' \bigwedge_\pi \mathsf{E}'')$ (ie., $\mathsf{A} = (\mathsf{E}' \bigwedge_\pi \mathsf{E}'')\,\mathsf{A}_1\cdots\mathsf{A}_n$), where $\pi \neq o$. The proof for this case follows easily using the outer induction hypothesis. □

G. PROOF OF PROPOSITION 8.2

Proposition 8.2 Let P be a program and let F be a formula of $\mathcal{H}$. Then, F is a logical consequence of P under the proposed semantics iff F is a logical consequence of P under the continuous semantics.

PROOF OUTLINE. We start by defining a relation $\sim$ between elements of our semantic domains (Definition 5.1) and elements of the domains under the continuous interpretation (see Subsection 8.2). Let $f \in [\![\tau]\!]_D$ and $f^* \in [\![\tau]\!]^*_D$. We write $f \sim f^*$ if one of the following holds:

— $\tau \in \{\iota, o\}$ and $f = f^*$
— $\tau = \rho \to \pi$ and for every $d \in \mathcal{F}_D(\rho)$ and $d^* \in [\![\rho]\!]^*_D$ such that $d \sim d^*$, $f(d) \sim f^*(d^*)$.

One can show that $\sim$ is a bijection. We can then extend the $\sim$ relation on interpretations and states. In particular, given interpretations I and I^* under the proposed and the continuous semantics respectively, we write $I \sim I^*$ iff for every $\mathtt{p}$, $I(\mathtt{p}) \sim I^*(\mathtt{p})$; similarly for states s, s^*.

Consider now a program P and let I be an interpretation and s a state of P under the proposed semantics and I^*, s^* an interpretation and a state under the continuous semantics such that $I \sim I^*$ and $s \sim s^*$. One can demonstrate the following lemma: for every formula F it holds that $[\![\mathsf{F}]\!]_s(I) \sim [\![\mathsf{F}]\!]^*_{s^*}(I^*)$. The proof is by a structural induction on F. Actually, the interesting case is when $\mathsf{F} = (\mathsf{E}_1\mathsf{E}_2)$, where the algebraicity of the

proposed semantics is used in order to establish the equivalence of the two semantics of application. Then, one can establish the following lemma:

> **Lemma:** Let S be a set of formulas of $\mathcal{H}$ and let I, I^* be interpretations of $\mathcal{H}$ under the proposed and continuous semantics respectively such that $I \sim I^*$. Then I is a model of S under the proposed semantics if and only if I^* is a model of S under the continuous semantics.

Suppose now that F is a logical consequence of P under the proposed semantics. Let I^* be a model of P under the continuous semantics. Assume that I denotes the unique interpretation under the proposed semantics such that $I \sim I^*$. By the above lemma it follows that I is also a model of P under the proposed semantics. In addition, since F is a logical consequence of P under the proposed semantics, we also have that I is a model of F under the proposed semantics. But then, again by the above lemma, we also get that I^* is a model of F under the continuous semantics. Hence, F is a logical consequence of P under the continuous semantics. The other direction of the lemma is completely symmetrical. □