Proof Procedures for Extensional Higher-Order Logic Programming*

Angelos Charalambidis

Department of Informatics & Telecommunications
University of Athens
a.charalambidis@di.uoa.gr

Abstract. We consider an extensional higher-order logic programming language which possesses the minimum Herbrand model property. We propose an SLD-resolution proof procedure and we demonstrate that it is sound and complete with respect to this semantics. In this way, we extend the familiar proof theory of first-order logic programming to apply to the more general higher-order case. We then enhance our source language with constructive negation and extend the aforementioned proof procedure to support this new feature. We demonstrate the soundness of the resulting proof procedure and describe an actual implementation of a language that embodies the above ideas.

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. The initial attitude of logic programmers towards higher-order logic programming was somewhat skeptical: it was often argued that there exist ways of encoding or simulating higher-order programming inside Prolog itself (see, for example, [8]). 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. These extensions allow predicates to be applied but also passed as parameters. The existing proposals can be placed in two main categories, namely the *intensional* and the *extensional* ones. In the former category, the two most prominent languages are λProlog [6] and HiLog [4]. The latter category is much less developed: currently there exist two main proposals for extensional higher-order logic

* Advisor: Panos Rondogiannis, Associate Professor, University of Athens. The doctoral dissertation 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.

programming, namely [7] and [1]; however, apart from the work reported in this dissertation, no other 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.

Example 1. Consider a program that consists only of the following rule:

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

In an extensional logic language, predicate `p` can be 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 this program is also a syntactically acceptable program of the existing intensional logic programming languages. The difference is that 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 to such a query (since there does not exist any actual predicate defined in the program that is true of both `0` and `1`).

The first work in this area was [7]. 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 ensures that compliant programs have an extensional declarative reading. 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. Finally, Wadge conjectured that a sound and complete proof system exists for his fragment, but did not further pursue such an investigation.

2 The Proposed Approach: an Intuitive Overview

The first problem we consider is to bypass one important restriction of [7], namely the inability to handle program clauses or queries that contain uninstantiated predicate variables. The following example illustrates these ideas:

Example 2. 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 [7], 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`.

In our example, despite the fact that there exists an infinite number of relations that satisfy `p`, all of them 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? For this purpose we adopt the semantics described in [3]. 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. The key idea can be demonstrated by continuing Example 2. 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. For example, 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))
?-true
```

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 can be created that returns only the answer $R = \{0, s(0)\}$, which represents all the finite relations produced by the proof system. The intuition behind this answer is that the given query succeeds for all unary relations that contain at least `0` and `s(0)`.

One basic property of all the higher-order predicates that can be defined in the language considered so far, is that they are monotonic. Intuitively, the monotonicity property states that if a predicate is true of a relation `R` then it is also true of every superset of `R`. However, there are many natural higher-order predicates that are *nonmonotonic* and which a programmer would like to be able to write. For example, assume we want to define a predicate `disconnected(G)` which succeeds if its input argument, namely a graph `G`, is disconnected. A graph is simply a set of pairs, and therefore `disconnected` is a second-order predicate. Notice that `disconnected` is obviously nonmonotonic: given graphs `G1` and `G2` with $G1 \subseteq G2$, it is possible that `disconnected(G1)` succeeds but `disconnected(G2)` fails.

The obvious idea in order to add nonmonotonicity is to enhance the language with negation-as-failure. However, this is not as straightforward as it sounds,

because even the simpler higher-order programs with negation face the well-known problem of *floundering* [5]. In classical logic programming a computation is said to flounder if at some point a goal is reached that contains only nonground negative literals.

Fortunately, there exists an approach to negation-as-failure that bypasses the problem of floundering. This is usually called *constructive negation* [2] and its main idea can be explained by a simple example. Consider the predicate `p` that holds for `1` and `2` and consider the query `?-not(p(X))`. The original idea behind constructive negation [2] is that in order to answer a negative query that contains uninstantiated variables, the following procedure must be applied: we run the positive version of the query and we collect the solutions as a disjunction; we then return the negation of the disjunction as the answer to the original query. In our example, the corresponding positive query (namely `?-p(X)`) has the answers `X=1` and `X=2`. The negation of their disjunction is the conjunction $(\texttt{X} \neq \texttt{1}) \wedge (\texttt{X} \neq \texttt{2})$. Observe now that the procedure behind constructive negation returns as answers not only substitutions (as it happens in negationless logic programming) but also inequalities. Generalizing the above idea to the higher-order setting requires the ability to express some form of inequalities regarding the elements of sets. Intuitively, we would like to express that some element *does not belong to a set.*

Example 3. Consider the following simple program:

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

Intuitively, `p` is true of all relations that contain `0` but they do not contain `1`. A reasonable answer for `?-p(R)` would be $\texttt{R} = \{\texttt{0}\} \cup \{\texttt{X} \mid \texttt{X} \neq \texttt{1}\}$.

It turns out that the extension of higher-order logic programs with constructive negation offers a much greater versatility to extensional higher-order logic programming. We can extend higher-order logic programming with constructive negation. Moreover, there exists a relative simple sound proof procedure for the new language.

3 Definite Higher-Order Programs

Definition 1. *A type can either be* functional*,* argument*, or* predicate*, denoted by σ, ρ and π respectively and defined as:*

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

We will use τ to denote an arbitrary type.

As usual, the binary operator $\to$ is right-associative. A functional type that is different from 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 $\pi = o$).

Definition 2. *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.

Definition 3. *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*. A program of* $\mathcal{H}$ *is a finite set of program clauses of* $\mathcal{H}$*.*

Example 4. The following is a higher-order program that computes the closure of its input binary relation `R`. The type of `closure` is $\pi = (\iota \to \iota \to o) \to \iota \to \iota \to o$.

```
closure ←π λR.λX.λY.(R X Y)
closure ←π λR.λX.λY.∃Z((R X Z)∧(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 (`a`, `b`) belongs to their transitive closure). In a Prolog-like extended syntax, this 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)`.

3.1 SLD Resolution

Definition 4. *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* $(\mathsf{V}_i\ \mathsf{B}_{11} \cdots \mathsf{B}_{1r}) \wedge \cdots \wedge (\mathsf{V}_i\ \mathsf{B}_{m1} \cdots \mathsf{B}_{mr})$*, where* $m > 0$*, 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} *will be called the* basic subexpressions *of* B*.*

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 nonempty finite union of basic expressions in which all the basic subexpressions involved are *distinct* free variables, is a basic template.

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. The notions of *unifier* and *most general unifier* is defined in a standard way for the zero-order substitutions. The composition of two substitutions and the application of a substitution to a positive expressions is defined as usual.

Definition 5. *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* 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. $(\exists \mathsf{V}\, \mathsf{E}) \xrightarrow{\epsilon} \mathsf{E}$

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 $\mathsf{G}_n = \Box$. Then, we will say that $\mathsf{P} \cup \{\mathsf{G}\}$ has an *SLD-refutation of length n using basic substitution* $\theta = \theta_1 \cdots \theta_n$. 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 5. Consider the program of Example 4. A successful SLD-refutation of the goal $\leftarrow$ (`closure Q a b`) is given here (where we have omitted certain simple steps involving lambda abstractions).

`closure Q a b`	$\theta_1 = \epsilon$
`(λR.λX.λY.(R X Y)) Q a b`	$\theta_2 = \epsilon$
`Q a b`	$\theta_3 = \{\mathtt{Q}/(\lambda \mathtt{X}.\lambda \mathtt{Y}.(\mathtt{X} \approx \mathtt{X}_0) \wedge (\mathtt{Y} \approx \mathtt{Y}_0))\}$
`(λX.λY.(X≈X0)∧(Y≈Y0)) a b`	$\theta_4 = \epsilon$
`(a≈X0)∧(b≈Y0)`	$\theta_5 = \{\mathtt{X}_0/\mathtt{a}\}$
`true∧(b≈Y0)`	$\theta_6 = \epsilon$
`(b≈Y0)`	$\theta_7 = \{\mathtt{Y}_0/\mathtt{b}\}$
`true`	

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 $\{(\mathtt{a}, \mathtt{b})\}$. Notice that by substituting `Q` with different basic templates, one can get answers that are "similar" to the previous one, such as for example $\{(\mathtt{a}, \mathtt{b}), (\mathtt{Z1}, \mathtt{Z2})\}$, and so on. Answers of this type are in some sense "represented" by the answer $\{(\mathtt{a}, \mathtt{b})\}$.

We say that the basic substitution θ is *a correct answer* for $\mathsf{P} \cup \{\mathsf{G}\}$ if $\mathsf{A}\theta$ is a logical consequence of P.

Theorem 1 (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* $\mathsf{P} \cup \{\mathsf{G}\}$.

Theorem 2 (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* $\mathsf{G}\theta = \mathsf{G}\delta\gamma$.

4 Normal Higher-Order Programs

The basic difference in the types of $\mathcal{H}_{cn}$ from those of $\mathcal{H}$ is the existence of a type μ, which restricts the set of predicate variables that can be existentially

quantified or appear free in goal clauses. The subtypes μ (existential type) and κ (set type) of ρ and π, respectively, are defined as follows.

$$\mu := \iota \mid \kappa$$
$$\kappa := \iota \to o \mid (\iota \to \kappa)$$

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

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

Definition 7. *The set of* clausal expressions *of $\mathcal{H}_{cn}$ is defined as follows:*

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

All clausal expressions of $\mathcal{H}_{cn}$ have type o. A program of $\mathcal{H}_{cn}$ is a finite set of program clauses of $\mathcal{H}_{cn}$.

Consider a graph represented by a binary relation over the set of its vertices. Then, given a graph of type $\pi = (\iota \to \iota \to o)$, we can express the set of its vertices as a predicate of type $\iota \to o$ of the graph and a path of the graph as a predicate of type $\iota \to \iota \to o$.

$\mathtt{v} \leftarrow (\lambda \mathtt{G}.\lambda \mathtt{X}.\exists \mathtt{Y}(\mathtt{G\ X\ Y})) \bigvee (\lambda \mathtt{G}.\lambda \mathtt{X}.\exists \mathtt{Y}(\mathtt{G\ Y\ X}))$
$\mathtt{p} \leftarrow \lambda \mathtt{G}.\lambda \mathtt{X}.\lambda \mathtt{Y}.(\mathtt{closure\ G\ X\ Y})$

The predicate v succeeds if its second argument is a vertex of its first argument. The predicate p succeeds if there is a path between two vertices of the graph. Note that the path is actually a transitive closure of the graph. We can easily express basic connectivity properties of a graph.

$\mathtt{disconnected} \leftarrow \lambda \mathtt{G}.\exists \mathtt{X} \exists \mathtt{Y}((\mathtt{v\ G\ X}) \wedge (\mathtt{v\ G\ Y}) \wedge \sim(\mathtt{X} \approx \mathtt{Y}) \wedge \sim(\mathtt{p\ G\ X\ Y}))$
$\mathtt{nonclique} \leftarrow \lambda \mathtt{G}.\lambda \mathtt{S}.\exists \mathtt{X} \exists \mathtt{Y}((\mathtt{S\ X}) \wedge (\mathtt{S\ Y}) \wedge \sim(\mathtt{X} \approx \mathtt{Y}) \wedge \sim(\mathtt{G\ X\ Y}))$
$\mathtt{connected} \leftarrow \lambda \mathtt{G}.\sim(\mathtt{disconnected\ G})$
$\mathtt{clique} \leftarrow \lambda \mathtt{G}.\lambda \mathtt{S}.(\mathtt{subset\ S\ (v\ G)}) \wedge \sim(\mathtt{nonclique\ G\ S})$

Let P be a program and let p be a predicate constant of type π. Then, the *completed definition* for p with respect to P is obtained as follows:

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

The *program completion* $comp(\mathsf{P})$ of P is the set consisting of all the completed definitions for all predicate constants that appear in P.

4.1 Proof Procedure

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

- *valid* if E_1 and E_2 cannot be unified;
- *unsatisfiable* if there is a substitution θ that unifies E_1 and E_2 and contains only bindings of variables in $\hat{\mathsf{V}}$;
- *satisfiable* if it is not unsatisfiable.

An inequality will be called *primitive* if it is satisfiable, non valid and either E_1 or E_2 is a variable.

Definition 8. *The set of* normal basic expressions *of $\mathcal{H}_{cn}$ of type μ is defined recursively as follows.*

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

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

Definition 9. *Let* P *be a program and* E, E' *be body expressions of type* o. *We say that* E *is reduced (wrt. to* P*) to* E' *(denoted as* $\mathsf{E} \rightsquigarrow \mathsf{E}'$*) if one of the following conditions applies:*

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

Definition 10. *Let* P *a program and let* G_k *and* G_{k+1} *be goal clauses and let* G_k *be a conjunction* $\leftarrow \mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n$, *where each* A_i *is a body expression of type* o. *Moreover, let* A_i *one of the* $\mathsf{A}_1, \ldots, \mathsf{A}_n$ *(called* selected expression*) and* $\mathsf{A}' = \mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_{i-1} \wedge \mathsf{A}_{i+1} \wedge \cdots \wedge \mathsf{A}_n$. *We say that* G_{k+1} *is* derived in one step *from* G_k *using* θ *(denoted as* $\mathsf{G}_k \xrightarrow{\theta} \mathsf{G}_{k+1}$*) if one of the following conditions applies:*

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

Note that the single step derivation will essentially behave similarly with the Definition 5 for definite higher-order programs. The last three cases of the single-step derivation handle the cases of a negative expression.

Definition 11. *Let* P *be a program and let* $\mathsf{B} = \sim\exists \hat{\mathsf{U}}(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n)$ *be a body expression where* A_i *is a body expression except from conjunction. Let* A_i *be one of* $\mathsf{A}_1, \ldots, \mathsf{A}_n$ *and* $\mathsf{A}' = \mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_{i-1} \wedge \mathsf{A}_{i+1} \wedge \cdots \wedge \mathsf{A}_n$. *Moreover, let* B' *be a body expression. We say that* B *is* negatively reduced *to* B' *if one of the following conditions applies:*

1. *if* A_i *is* false, *then* $\mathsf{B}' = \mathsf{true}$;
2. *if* A_i *is* true *and* $n = 1$, *then* $\mathsf{B}' = \mathsf{false}$; *otherwise* $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}\ \mathsf{A}'$;
3. *if* A_i *is* $(\mathsf{E}_1 \vee \mathsf{E}_2)$, *then* $\mathsf{B}' = \mathsf{B}'_1 \wedge \mathsf{B}'_2$ *where* $\mathsf{B}'_j = \sim\exists\hat{\mathsf{U}}(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{E}_j \wedge \cdots \wedge \mathsf{A}_n)$ *with* $j \in \{1, 2\}$;
4. *if* A_i *is* $(\exists\mathsf{V}\ \mathsf{E})$, *then* $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}\mathsf{V}(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{E} \wedge \cdots \wedge \mathsf{A}_n)$;
5. *if* $\mathsf{A}_i \rightsquigarrow \mathsf{A}'_i$, *then* $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}'_i \wedge \cdots \wedge \mathsf{A}_n)$;
6. *if* A_i *is* $(\mathsf{E}_1 \approx \mathsf{E}_2)$, *then*
   (a) *if* $\sim\exists\hat{\mathsf{U}}(\mathsf{E}_1 \approx \mathsf{E}_2)$ *is valid, then* $\mathsf{B}' = \mathsf{true}$;
   (b) *if* $\sim\exists\hat{\mathsf{U}}(\mathsf{E}_1 \approx \mathsf{E}_2)$ *is not valid and if neither* E_1 *nor* E_2 *is a variable, then* $\hat{\mathsf{X}}$ *is* $dom(\theta)$, $\theta = unify(\mathsf{E}_1, \mathsf{E}_2)$ *and* $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}(\mathsf{A}_1 \wedge \cdots \wedge (\hat{\mathsf{X}} \approx \hat{\mathsf{X}}\theta) \wedge \cdots \wedge \mathsf{A}_n)$.
   (c) *if* $\sim\exists\hat{\mathsf{U}}(\mathsf{E}_1 \approx \mathsf{E}_2)$ *is unsatisfiable and either* E_1 *or* E_2 *is a variable in* $\hat{\mathsf{U}}$, *then* $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}(\mathsf{A}'\theta)$, *where* $\theta = \{\mathsf{X}/\mathsf{E}\}$ *and* X *is the one expression that is a variable in* $\hat{\mathsf{U}}$ *and* E *is the other;*
   (d) *if* $\sim\exists\hat{\mathsf{U}}(\mathsf{E}_1 \approx \mathsf{E}_2)$ *is primitive and* $n > 1$, *then* $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}_1\ \mathsf{A}_i \vee \exists\hat{\mathsf{U}}_1(\mathsf{A}_i \wedge \sim \exists\hat{\mathsf{U}}_2\ \mathsf{A}')$, *where* $\hat{\mathsf{U}}_1$ *are in* $\hat{\mathsf{U}}$ *that are free in* A_i *and* $\hat{\mathsf{U}}_2$ *in* $\hat{\mathsf{U}}$ *not in* $\hat{\mathsf{U}}_1$;
7. *if* A_i *is* $(\mathsf{R}\ \hat{\mathsf{E}})$ *and variable* $\mathsf{R} : \kappa$, *then*
   (a) *if* $\mathsf{R} \in \hat{\mathsf{U}}$, *then* $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}'(\mathsf{A}'\theta)$, *where* $\theta = \{\mathsf{R}/(\lambda\mathsf{X}.(\mathsf{X} \approx \mathsf{E}) \bigvee_\kappa \mathsf{R}')\}$, $\mathsf{R}' : \kappa$ *is a fresh variable and* $\hat{\mathsf{U}}' = \hat{\mathsf{U}}\{\mathsf{R}/\mathsf{R}'\}$;
   (b) *if* $\mathsf{R} \notin \hat{\mathsf{U}}$ *and* $n > 1$, *then* $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}_1\ \mathsf{A}_i \vee \exists\hat{\mathsf{U}}_1(\mathsf{A}_i \wedge \sim\exists\hat{\mathsf{U}}_2\ \mathsf{A}') \wedge \mathsf{B}$, *where* $\hat{\mathsf{U}}_1$ *are the variables in* $\hat{\mathsf{U}}$ *that are free in* A_i *and* $\hat{\mathsf{U}}_2$ *in* $\hat{\mathsf{U}}$ *not in* $\hat{\mathsf{U}}_1$;
8. *if* A_i *is* $\sim\exists\hat{\mathsf{V}}\ \mathsf{E}$ *and* A_i *is negatively reduced to* A'_i, *then* $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}'_i \wedge \cdots \wedge \mathsf{A}_n)$;
9. *if* A_i *is a primitive inequality* $\sim\exists\hat{\mathsf{V}}(\mathsf{E}_1 \approx \mathsf{E}_2)$, *then*
   (a) *if* $fv(\mathsf{A}_i) \cap \hat{\mathsf{U}} \neq \emptyset$ *and* A' *is conjunction of primitive inequalities, then* $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}\ \mathsf{A}'$;
   (b) *if* $fv(\mathsf{A}_i) \cap \hat{\mathsf{U}}$ *is empty, then* $\mathsf{B}' = \exists\hat{\mathsf{V}}(\mathsf{E}_1 \approx \mathsf{E}_2) \vee \sim\exists\hat{\mathsf{U}}\ \mathsf{A}'$;
10. *if* A_i *is* $\sim\exists\hat{\mathsf{V}}(\mathsf{R}\ \hat{\mathsf{E}})$, *variable* $\mathsf{R} : \kappa$ *and* $\mathsf{R} \notin \hat{\mathsf{V}}$, *then*
   (a) *if* $\mathsf{R} \in \hat{\mathsf{U}}$, *then* $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}'(\mathsf{A}'\theta)$ *where* $\theta = \{\mathsf{R}/(\lambda\mathsf{X}.\ \sim\exists\hat{\mathsf{V}}(\mathsf{X} \approx \mathsf{E}) \bigwedge_\kappa \mathsf{R}')\}$, $\mathsf{R}' : \kappa$ *is a fresh variable and* $\hat{\mathsf{U}}' = \hat{\mathsf{U}}\{\mathsf{R}/\mathsf{R}'\}$;
   (b) *if* $\mathsf{R} \notin \hat{\mathsf{U}}$ *and* $n > 1$, *then* $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}_1\ \mathsf{A}_i \vee \exists\hat{\mathsf{U}}_1(\mathsf{A}_i \wedge \sim\exists\hat{\mathsf{U}}_2\ \mathsf{A}') \wedge \mathsf{B}$, *where* $\hat{\mathsf{U}}_1$ *are the variables in* $\hat{\mathsf{U}}$ *that are free in* A_i *and* $\hat{\mathsf{U}}_2$ *in* $\hat{\mathsf{U}}$ *not in* $\hat{\mathsf{U}}_1$;
   (c) *if* $\mathsf{R} \notin \hat{\mathsf{U}}$, $n = 1$ *and* $\hat{\mathsf{V}} \neq \emptyset$, *then* $\mathsf{B}' = \exists\hat{\mathsf{V}}\ \sim\exists\hat{\mathsf{U}}(\sim(\mathsf{R}\ \hat{\mathsf{E}}) \wedge \sim\exists\hat{\mathsf{V}}'(\mathsf{R}\ \mathsf{E}'))$;
11. *if* A_i *is* $\sim\exists\hat{\mathsf{V}}\ \sim(\mathsf{R}\ \hat{\mathsf{E}})$ *and variable* $\mathsf{R} : \kappa$ *and* $\mathsf{R} \notin \hat{\mathsf{V}}$, *then*
   (a) *if* $\mathsf{R} \in \hat{\mathsf{U}}$, *then* $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}'(\mathsf{A}'\theta)$, *where* $\theta = \{\mathsf{R}/(\lambda\mathsf{X}.\exists\hat{\mathsf{V}}(\mathsf{X} \approx \mathsf{E}) \bigvee_\kappa \mathsf{R}')\}$, $\mathsf{R}' : \kappa$ *is a fresh variable and* $\hat{\mathsf{U}}' = \hat{\mathsf{U}}\{\mathsf{R}/\mathsf{R}'\}$;
   (b) *if* $\mathsf{R} \notin \hat{\mathsf{U}}$ *and* $n > 1$, *then* $\mathsf{B}' = \sim\exists\hat{\mathsf{U}}_1\ \mathsf{A}_i \vee \exists\hat{\mathsf{U}}_1(\mathsf{A}_i \wedge \sim\exists\hat{\mathsf{U}}_2\ \mathsf{A}') \wedge \mathsf{B}$;
   (c) *if* $\mathsf{R} \notin \hat{\mathsf{U}}$, $n = 1$ *and* $\hat{\mathsf{V}} \neq \emptyset$, *then* $\mathsf{B}' = \exists\hat{\mathsf{V}} \sim\exists\hat{\mathsf{U}}((\mathsf{R}\ \hat{\mathsf{E}}) \wedge \sim\exists\hat{\mathsf{V}}' \sim(\mathsf{R}\ \mathsf{E}'))$.

The computed answer of a successful derivation with primitive goal G' and basic substitution θ is extended as follows: The tuple (σ, G'') is a computed answer for P where σ is the basic substitution obtained by restricting θ to the free variables of G and G'' is the primitive goal G' restricted to the free variables of G and the variables in $fv(range(\sigma))$.

Example 6. Consider the following simple definition for the predicate $\mathtt{q}$.

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

that holds only for the tuple $(\mathtt{a}, \mathtt{b})$. Then, consider the goal: $\leftarrow (\mathtt{R}\ \mathtt{X}) \wedge \sim(\mathtt{q}\ \mathtt{X}\ \mathtt{Y})$ that requests bindings for the variables $\mathtt{R}, \mathtt{X}$ and $\mathtt{Y}$.

0. $\leftarrow \underline{(\mathtt{R}\ \mathtt{X})} \wedge \sim(\mathtt{q}\ \mathtt{X}\ \mathtt{Y})$
1. $\leftarrow \sim\underline{(\mathtt{q}\ \mathtt{X}\ \mathtt{Y})}$ using $\theta_1 = \{\mathtt{R}/\lambda\mathtt{Z}.(\mathtt{Z} \approx \mathtt{X}) \bigvee \mathtt{R}'\}$
2. $\leftarrow \sim\underline{((\lambda\mathtt{Z}_1.\lambda\mathtt{Z}_2.(\mathtt{Z}_1 \approx \mathtt{a}) \wedge (\mathtt{Z}_2 \approx \mathtt{b}))\ \mathtt{X}\ \mathtt{Y})}$
3. $\leftarrow \underline{\sim((\mathtt{X} \approx \mathtt{a}) \wedge (\mathtt{Y} \approx \mathtt{b}))}$
4. $\leftarrow \sim(\mathtt{X} \approx \mathtt{a}) \vee (\mathtt{X} \approx \mathtt{a}) \wedge \sim(\mathtt{Y} \approx \mathtt{a})$
5. $\leftarrow \sim(\mathtt{X} \approx \mathtt{a})$

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

Theorem 3 (Soundness). *Let* P *be a program and* G *be a goal. Then, every computed answer for* $\mathsf{P} \cup \{\mathsf{G}\}$ *is a correct answer for* $comp(\mathsf{P}) \cup \{\mathsf{G}\}$.

5 Conclusions

In this dissertation we have extended the study initiated in [7] and derived a complete framework for extensional higher-order logic programming. We have introduced the higher-order language $\mathcal{H}$ that extends the language in [7] and allows uninstantiated predicate variables. We have proposed a sound and complete SLD-resolution with respect to the minimum Herbrand model semantics.

We have also introduced an extension of $\mathcal{H}$ that supports the operator of negation-as-failure. We have proposed a proof procedure that extends the higher-order SLD-resolution in order to handle constructive negation. As a result, the proposed proof procedure, avoids the floundering problem of negation-as-failure. We have also established the soundness of the proof procedure with respect to the completion semantics.

References

1. M. Bezem. An improved extensionality criterion for higher-order logic programs. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL)*, pages 203–216, London, UK, 2001. Springer-Verlag.
2. D. Chan. Constructive negation based on the completed database. In *ICLP/SLP*, pages 111–125, 1988.
3. A. Charalambidis, K. Handjopoulos, P. Rondogiannis, and W. W. Wadge. Extensional higher-order logic programming. *ACM Trans. Comput. Log.*, 14(3), 2013.
4. W. Chen, M. Kifer, and D. S. Warren. Hilog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.
5. J. W. Lloyd. *Foundations of Logic Programming, 2nd Edition.* Springer, 1987.
6. G. Nadathur. *A Higher-Order Logic as the Basis for Logic Programming.* PhD thesis, University of Pennsylvania, 1987.
7. W. W. Wadge. Higher-order horn logic programming. In *Proceedings of the International Symposium on Logic Programming (ISLP)*, pages 289–303, 1991.
8. D. H. Warren. Higher-order extensions to prolog: are they needed? *Machine Intelligence*, 10:441–454, 1982.