NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Implementation of Constructive Negation in Extensional Higher-Order Logic Programming

Ergys A. Dona

Supervisors: **Panos Rondogiannis,** NKUA Professor
Angelos Charalambidis, N.C.S.R. “Demokritos” Researcher

ATHENS

JUNE 2017

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Υλοποίηση Κατασκευαστικής Άρνησης σε Εκτατικό Λογικό Προγραμματισμό Υψηλής Τάξης

Ergys A. Dona

Επιβλέποντες: **Πάνος Ροντογιάννης,** Καθηγητής ΕΚΠΑ
Άγγελος Χαραλαμπίδης, Ερευνητής ΕΚΕΦΕ «Δημόκριτος»

ΑΘΗΝΑ

ΙΟΥΝΙΟΣ 2017

BSc THESIS

Implementation of Constructive Negation in Extensional Higher-Order Logic Programming

Ergys A. Dona
S.N.: 1115200900148

SUPERVISORS: **Panos Rondogiannis,** NKUA Professor
Angelos Charalambidis, N.C.S.R. “Demokritos” Researcher

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Υλοποίηση Κατασκευαστικής Άρνησης σε Εκτατικό Λογικό Προγραμματισμό Υψηλής Τάξης

Ergys A. Dona
Α.Μ.: 1115200900148

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Πάνος Ροντογιάννης,** Καθηγητής ΕΚΠΑ
Άγγελος Χαραλαμπίδης, Ερευνητής ΕΚΕΦΕ «Δημόκριτος»

ABSTRACT

Constructive negation is a logic programming negation method that allows the handling of non-ground negative literals. Logic programming systems which are enhanced with constructive negation may produce not only equalities (assignments or unifications of variables), but also inequalities, that, roughly speaking, restrict variables to be different from some value.

Extensional higher-order logic programming is a logic programming paradigm that extends classical (first-order) logic programming by introducing higher-order terms, while preserving all the well-known properties of the former. These higher-order terms (variables) represent sets, which contain elements for which the former succeed.

An extensional higher-order logic programming language called HOPES ($\mathcal{H}$) has been recently proposed. $\mathcal{H}$ has been enhanced with constructive negation to create $\mathcal{H}_{\text{cn}}$ (HOPES with constructive negation).

While the foundations of $\mathcal{H}_{\text{cn}}$ are based on strong and well-defined semantics, implementing a logic programming system (interpreter) for it proved to be a real challenge. The reason is that the enhanced definition of the $\mathcal{H}_{\text{cn}}$ proof procedure does not define any particular method for selecting the next subgoal to be satisfied. We show that, while the language definition guarantees that there exist paths in the proof tree that lead to correct answers, if a naive selection policy (e.g. always selecting the left-most literal, like PROLOG does) is adopted, then one has to take extra measures in order to avoid following paths that will never lead to correct solutions.

The contribution of this thesis consists of the redefinition of some of the rules in the $\mathcal{H}_{\text{cn}}$ proof procedure. We show that the redefined proof procedure allows the implementation of an interpreter for $\mathcal{H}_{\text{cn}}$, which follows the left-most derivation rule of PROLOG and behaves correctly (only gives correct answers) for any given query.

SUBJECT AREA: Logic Programming

KEYWORDS: logic, programming, negation, high-order, extensional

ΠΕΡΙΛΗΨΗ

Η κατασκευαστική άρνηση είναι μια μέθοδος άρνησης στον λογικό προγραμματισμό, η οποία επιτρέπει το χειρισμό μη-συγκεκριμενοποιημένων αρνητικών προτάσεων. Τα συστήματα λογικού προγραμματισμού τα οποία είναι επαυξημένα με κατασκευαστική άρνηση μπορούν δυνητικά να παράγουν όχι μόνο ισότητες (αναθέσεις ή ενοποιήσεις μεταβλητών), αλλά και ανισότητες, οι οποίες, γενικά μιλώντας, περιορίζουν τις μεταβλητές έτσι ώστε να είναι διαφορετικές από κάποια τιμή.

Ο εκτατικός λογικός προγραμματισμός υψηλής τάξης είναι ένα παράδειγμα λογικού προγραμματισμού που επεκτείνει τον κλασικό (πρώτης τάξης) λογικό προγραμματισμό εισάγοντας όρους υψηλής τάξης, ενώ παράλληλα διατηρεί τις καλά ορισμένες ιδιότητες του προηγούμενου. Αυτοί οι υψηλής τάξης όροι αναπαριστούν σύνολα, τα οποία περιλαμβάνουν στοιχεία για τα οποία οι προηγούμενοι είναι αληθείς.

Μια γλώσσα λογικού προγραμματισμού υψηλής τάξης με όνομα HOPES ($\mathcal{H}$) έχει προταθεί πρόσφατα. Η γλώσσα αυτή έχει επαυξηθεί με κατασκευαστική άρνηση για να διαμορφωθεί η γλώσσα $\mathcal{H}_{\text{cn}}$ (HOPES with constructive negation).

Ενώ η σημασιολογία της $\mathcal{H}_{\text{cn}}$ είναι καλά ορισμένη, η υλοποίηση ενός συστήματος λογικού προγραμματισμού (διερμηνευτή) για την τελευταία αποδείχθηκε δυσκολότερη υπόθεση απ'ότι αναμενόταν. Ο λόγος είναι πως η διαδικασία απόδειξης της $\mathcal{H}_{\text{cn}}$ δεν υποδεικνύει κανέναν συγκεκριμένο τρόπο επιλογής του επόμενου στόχου για ικανοποίηση. Δείχνουμε πως, ενώ ορισμός της γλώσσας εγγυάται την ύπαρξη μονοπατιών στο δέντρο απόδειξης, τα οποία οδηγούν σε σωστές λύσεις, εάν μια απλοϊκή πολιτική επιλογής (π.χ. η επιλογή πάντα του αριστερότερου υποστόχου, όπως κάνει η PROLOG) έχει υιοθετηθεί, τότε πρέπει να παρθούν επιπλέον μέτρα έτσι ώστε η υλοποίηση να αποφύγει μονοπάτια τα οποία δεν θα οδηγήσουν ποτέ σε σωστές λύσεις.

Η συνεισφορά αυτής της πτυχιακής έγκειται στον επαναορισμό κάποιων εκ των κανόνων της διαδικασίας απόδειξης της $\mathcal{H}_{\text{cn}}$. Δείχνουμε ότι η νέα διαδικασία απόδειξης επιτρέπει την υλοποίηση ενός διερμηνευτή για την $\mathcal{H}_{\text{cn}}$, ο οποίος ακολουθεί τον κανόνα της PROLOG για επιλογή του πάντα αριστερότερου υποστόχου και παράλληλα συμπεριφέρεται σωστά (δίνει σωστές λύσεις μόνο) για κάθε επερώτηση.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Λογικός Προγραμματισμός

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: λογική, προγραμματισμός, άρνηση, υψηλή τάξη, εκτατικός

Absence of evidence is not evidence of absence.

ACKNOWLEDGEMENTS

I would like to deeply thank NCSR “Demokritos” associate Angelos Charalambidis for his guidance and overall support during the writing of this thesis. His inputs were of vital importance and the working methodology we followed helped me develop many important research skills and maintain a solid workflow.

I would also like to thank NKUA professor Panos Rondogiannis, for always being in line in order to help me and support my efforts. His way of teaching and his courses that I attended have sparkled my interest in the theoretical elements of computer science. This thesis is the result of that sparkle and I am very grateful to him.

CONTENTS

LIST OF FIGURES

PREFACE

This thesis constitutes my final step towards completing the undergraduate degree programme in the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens. It was carried out while working full-time in CERN, which added to the challenge.

During the process of selecting and taking up a thesis subject, I encountered many interesting topics and had discussions with many professors. I would like to thank them all for their patience, cooperation and tips.

The reason for selecting the current subject was that it was fascinating at first sight. I was no stranger to classical logic programming or high-order programming paradigms (e.g. Haskell), however it was the first time I encountered a combination of them.

$\mathcal{H}_{\mathrm{cn}}$ is a really expressive language. Its higher-order semantics allow one to write even some of the most complex predicates in a very elegant fashion. Of course the main property that enables such elegance is constructive negation, which is the principal feature of $\mathcal{H}_{\mathrm{cn}}$.

$\mathcal{H}_{\mathrm{cn}}$ borrows the semantics of constructive negation as defined by David Chan for the first-order case. It adapts them to the higher-order semantics of itself and does so perfectly in a theoretical level. However, implementing $\mathcal{H}_{\mathrm{cn}}$ turned out to be a more challenging task than anticipated. The motivation behind this thesis is the identification and redefinition of those characteristics of the higher-order language $\mathcal{H}_{\mathrm{cn}}$ that make it difficult for it to be implemented correctly.

A correct implementation of $\mathcal{H}_{\mathrm{cn}}$ shall enable a whole world of applications to demonstrate themselves in solving interesting problems and this is what we want to achieve via this thesis.

1. INTRODUCTION

1.1 Objective

$\mathcal{H}_{\text{cn}}$ is a extensional, higher-order, logic programming language with costructive negation [4]. All the terms in the previous sentence will be explained later on. An implementation of this language has been attempted and can be found at the following GitHub repository: `https://github.com/acharal/hopes`.

While the aforementioned implementation works fine for positive first and higher-order terms, as well as for negative first-order terms, when it comes to negative higher-order terms, it falls short to deliver the intended results.

The objective of this thesis is, initially, the identification of the rules in the proof procedure of $\mathcal{H}_{\text{cn}}$ that are problematic and contribute to the impractical behaviour of the implementation. Thereinafter, the various ways to fix the unwanted behaviour shall be considered and a solution to the problem shall be given.

1.2 Motivation

Some problems (i.e. predicates) can be expressed better and more elegantly via negation. Giving the programmer the ability to write such programs, makes the language a very powerful tool.

Example 1.1. Consider the following higher-order logic program:

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

The above program contains two rules to define the *subset* relation:

- P is a subset of Q if it does not hold that P is not a subset of Q.
- P is not a subset of Q if there exists an X, such that X is in P, but not in Q.

Therefore, if there exists a predicate q that is true for atoms 0, 1 and 2, then the query:

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

would return in P all subsets of the set $\{0, 1, 2\}$.

What makes the above program powerful, is the ability to write the first rule, which makes use of constructive negation, in order to obtain answers by negating the very simple second rule. The second rule itself is not very useful in the sense that it states something very trivial.

If we try and think of how we would write the same program in a positive logic, then we would normally have to check whether every element of P also exists in Q. Namely, for all $\mathsf{X} \in \mathsf{P}$, is $\mathsf{Q}(\mathsf{X})$ true? This is considerably harder to express, if possible at all (in Horn logic), because it involves iterating through P (in some way). However, the problem is way more easy to express in a negative logic: P is a subset of Q if there doesn't exist any X, such that X is in P but not in Q. Therefore, negation adds to the expressivity of the language, i.e. there are programs using negation that cannot be rephrased as positive ones.

The most common logic approach when it comes to negation in logic programming is negation as failure. Unfortunately, due to its nature, negation as failure does not provide variable bindings for a negated query. In SLDNF (the basic negation as failure proof procedure) only “allowed” programs (a subclass of all programs) can be proved. If the program is not allowed, then the correctness theorem for SLDNF does not hold. The real issue for higher-order logic programming is that there are many programs which are not allowed at all. Constructive negation alleviates both the above restrictions by introducing inequalities and the “not in” property for higher-order (set) variables. In the example above, notice that if the `not(non_subset(P,q))` subgoal could not provide variable bindings, it would be impossible to answer the `subset(P,q)` query.

The above example lays solid ground for solving generate and test type of problems. Suppose that we have a collection of objects that we want to test against some property. The `subset` predicate could serve as the generator of sets of objects that we want to test and will prove of special importance in the rest of the thesis. What we want, is to successfully implement the proof procedure that will give the expected answers to queries in programs like the one in example 1.1.

Preference-based queries

The above paradigm can be applied to another interesting class of applications. An interpreter that supports higher-order predicates in conjunction with constructive negation is a system that can potentially answer preference-based queries.

In [5], Charalambidis et al. propose the use of higher-order logic programming as a logical framework that handles preference-based queries. The aforementioned framework supports both *preferences over tuples*, as well as *preferences over sets*. It is shown that while the former can be easily expressed in first-order logic as well, preferences over tuples are more complex and could be better approached by using a higher-order logic programming language that supports existential higher-order variables. The extensional approach that is presented in [5], is a natural candidate for implementing preferences over sets since the binding for higher-order variables are essentially sets.

In extensional languages, two sets that contain the same elements are considered equal. Working with sets in an extensional way is one of the main motivating points that led to the creation of $\mathcal{H}$. Working with and stating set preferences is one of the main motivating points that led to the creation of $\mathcal{H}_{\text{cn}}$.

To give a flavour of the preferences over sets paradigm, imagine that there is a collection of books of various genres. Imagine that you have decided that you want to buy three books for your summer holidays and you really want them to be detective novels. However, you can only spend a limited time reading books and you get bored easily (it is summer after all), so you want them to have as few pages as possible.

Notice the order of preferences above. The first is the genre. You absolutely enjoy reading detective novels while on the beach. Even if a book of another genre contains fewer pages, you still prefer detective novels. This order of preferences may, depending on the collection of books, force you to buy less than three books (in case there are not enough, i.e. less than three, books of the selected genre in the collection).

The above problem can be reduced to the selection of the subsets of a collection of books that satisfy the imposed preferences. Those preferences over sets can be expressed more elegantly and more efficiently in an extensional higher-order language with constructive negation [5].

1.3 Overview

To get a taste of where and how the current approach fails, consider a higher-order logic program containing the rules of example 1.1. The problematic point, which will be fully elaborated later, is the way that the current implementation constructs the higher-order variable values (sets).

Given the query:

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

Then, the naive approach which is currently employed and involves the selection of the left-most subgoal (exactly like PROLOG does), will answer:

```
yes
P = { 0 } ;

yes
P = { 0, 0 } ;

yes
P = { 0, 0, 0 } ;

...
```

The problem comes down to the way the next candidate value to include in P is considered. The naive approach takes no information into account from the context (namely, the values already in P) and leads to the above result. The problem is two-fold: not only the returned answers contain duplicates, but also the procedure gets “stuck”, i.e. some correct answers will never be returned (e.g. in the above, the implementation will continue returning “sets” that only contain zeroes, for ever).

The proposed approach involves the introduction of restrictions (constraints), so that the answer set which represents P will only contain the expected members. This will in turn enable the implementation to avoid getting into a loop of returning meaningless answers, like above.

1.4 Thesis Structure

The rest of this thesis is organised as follows. Chapter 2 provides some background on the essential terms of negation as failure, constructive negation and higher-order logic programming. In chapter 3 the higher-order logic programming language $\mathcal{H}_{\text{cn}}$ is defined. In chapter 4 the rules that make up the proof procedure of $\mathcal{H}_{\text{cn}}$ are outlined and the problem that makes the implementation challenging is identified and presented. Chapter 5 shows the proposed approach to overcome the problem and outlines possible limitations. Finally, chapter 6 concludes the thesis and outlines the various possibilities for future work.

2. BACKGROUND

2.1 Negation as Failure

Negation as failure [6] is the most widely used approach for handling negation in logic programming systems. The simplicity in its implementation as well as its efficiency make it a really attractive option to employ.

These two advantages of negation as failure originate from the simplicity of its operational semantics: *If P cannot be proved based on the facts of the knowledge base (of the program), then assume that P is false*. The previous statement is also know as the *closed world assumption rule*. This is an extremely simple approach that greatly simplifies the handling of negative literals. Given a negative literal, e.g. not(p), all one has to do is run the positive version of it, i.e. p. If it succeeds, then fail. If it fails, then succeed.

This behaviour can be easily demonstrated if we consider the usual negation as failure implementation in Prolog, using cut and fail:

```
not(Goal) :- call(Goal), !, fail.
not(Goal).
```

A closer look at the above implementation immediately reveals the basic restriction of negation as failure. Note that nowhere in the above description variable bindings are discussed. This is because negation as failure can only be used as a test against program facts and cannot produce any variable bindings. This means that it can only handle ground literals (that is, literals that only contain ground atoms or, put otherwise, literals containing no variables).

Example 2.1. Consider the following logic program:

```
p(X) :- not(q(X)).
q(X) :- not(r(X)).

r(1).
r(2).
```

and the query:

```
?- p(X).
```

Negation as failure will indeed confirm that $p(X)$ is true by answering yes, but it will be unable to provide any bindings for X. This is due to the `fail` in the implementation, which causes any bindings that may have been produced by the `call` meta-predicate call to be lost.

Another problem with negation as failure is that its implementation is unsound.

Example 2.2. Consider the following logic program:

```
p(X) :- not(q(X)), r(X).

q(a).
r(b).
```

Considering the semantics of negation as failure, the predicate $p(X)$ seems to succeed for $X = b$ and to fail for $X = a$. Indeed, that is the case:

```
?- p(a).
no

?- p(b).
yes
```

Therefore, if a query that asks for all X, such that $p(X)$ is true is posed, the answer $X = b$ is expected. However, this is not the case because, as already stated, negation as failure cannot handle non-ground negative literals in a sound manner:

```
?- p(X).
no
```

2.2 Constructive Negation

The concept of Constructive Negation was first introduced by Chan [1] as an alternative to negation as failure. Constructive negation promises the handling of negative non-ground literals, by returning not only variable equalities (bindings) but also inequalities.

The idea behind constructive negation is simple: if a negated query, e.g. $\neg Q$ is given, we run the positive version of it, i.e. Q, and obtain the answers as a disjunction. Then, we return the negation of answers to Q as an answer to $\neg Q$.

Example 2.3. Suppose that we have the following very simple knowledge base:

```
p(a).
p(b).
p(c).
```

and the query:

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

We first run the positive version of the query, i.e. $p(X)$ and obtain the answer the disjunction: $X = a \vee X = b \vee X = c$. Then, we return the negated disjunction as an answer to the original query: $X \neq a \wedge X \neq b \wedge X \neq c$.

Example 2.4. Consider the following logic program:

```
p(X) :- not(q(X)).

q(a).
q(b).
```

then, for the query $p(X)$ the answer will be:

```
?- p(X).
yes
X /= a, X /= b
```

In the above example, notice how inequalities can be generated at any point of the computation where a negative literal is encountered. This resembles the constraint logic programming scheme, however simpler, because the restrictions here are only inequalities ($\neq$) and not e.g. $\leq$ or $\geq$.

Please note that the above examples are fairly simple, in order to demonstrate the idea of constructive negation. In reality, a more complex formula is used in order to negate the answers to the positive version of the negated query. Furthermore, in both examples, notice how the positive version of the negated query returns a finite number of answers. How does constructive negation handle the negation of a disjunction of infinite answers? This issue is addressed in [2], where the constructive negation rule is extended to support the negation of infinite answers.

We will not analyse the techniques that are used in [1] and [2] in order to apply the constructive negation rule in first-order logic programming. The reader is encouraged to refer to the respective texts. However, we will outline an important equivalence, that the constructive negation scheme makes use of when it encounters equalities, in order to negate the answers. This equivalence is of special importance to us, because it is the one that gets extended in order to apply the constructive negation rule to higher-order variables.

Let the goal list be $A = A_1, \dots, A_n$ and the selected expression be A_i, where A_i is $X = s$. Also, let A' be the rest of the goal list except A_i, i.e. $A' = A_1, \dots, A_{i-1}, A_{i+1}, \dots, A_n$. Then, the negation formula is the following:

$$\forall X\Big(\neg\exists \mathbf{Y}, \mathbf{Z}(X = s \wedge A') \;\equiv\; \neg\exists \mathbf{Y}\,(X = s) \vee \exists \mathbf{Y}(X = s \wedge \neg\exists \mathbf{Z}\; A')\Big) \tag{2.1}$$

where **Y** are the variables in s and the free variables in A' are contained in **Y** and **Z**.

The above formula is implied by the equality theory and is the mathematical formalisation of taking each component from the goal list A, negating it and then recombining the components. We will later extend the above formula to apply to higher-order terms.

To better understand why the above formula works, consider the following:

$$\begin{aligned}
\neg\,(A \wedge B) &\Leftrightarrow \neg A \vee \neg B && \\
&\Leftrightarrow \text{true} \wedge (\neg A \vee \neg B) && (\text{true} \wedge P \Leftrightarrow P) \\
&\Leftrightarrow (\neg A \vee A) \wedge (\neg A \vee \neg B) && (P \vee \neg P \Leftrightarrow \text{true}) \\
&\Leftrightarrow \neg A \vee (A \wedge \neg B) && (\text{rev. distributive law})
\end{aligned}$$

The reason why Chan decides to use the above formula in [1] when negating equalities is that the $X = s$ equality in the second term of the right-hand side of the equivalence may generate some interesting variable bindings that may lead to some extra welcome answers.

Example 2.5. Consider the following logic program that defines the `even` predicate:

```
even(0).
even(s(X)) :- not(even(X)).
```

The above program defines a negation-based rule for the `even` property of natural numbers, which are encoded in the following notation:

$$s^n(0) = n, \;\; n \geq 0$$

When asked the query:

```
?- even(X).
```

then using constructive negation, the implementation will answer:

```
yes
X = 0 ;

yes
X = s(V2)
V2 /= 0
V2 /= s(*V5) ;

yes
X = s(s(0)) ;

yes
X = s(s(s(V10)))
V10 /= 0
V10 /= s(*V14) ;

yes
X = s(s(s(s(0)))) ;

...
```

The second and the fourth answers above give a format for `X` along with some restrictions for some variables and they result from the constructive negation rule in equation 2.1. For example, the second answer says that `X` is of the form `s(V2)`, but `V2` is neither `0` nor `s(V5)`, for all `V5` (the star universally quantifies the following variable).

2.3 Extensional Higher-Order Logic Programming

Extensional higher-order logic programming was originally introduced by William W. Wadge [8] with Charalambidis et al. having also worked on that idea recently [3]. The paragim is essentially an extension of classical, first-order logic programming and its principal idea is that program predicates denote sets of objects and one can reason about such sets, i.e. use them in rules and facts. By looking at the name of this paradigm, we can see that it is made up of two terms: *extensional* and *higher-order*.

The term *higher-order* refers to the capability of applying and passing around predicates and predicate variables as parameters. This is the most popular feature of higher-order logic programming languages like HiLog and allows for the sets-based reasoning that was mentioned above.

The term *extensional* refers to the way program predicates can be passed and used, or otherwise, to what makes up a predicate or what a predicate really is. In an extensional language, two predicates are considered to be equal (i.e. the same) if they are true for the same set of arguments. The opposite of this term is *intentional* and in languages belonging to this category (e.g. HiLog), predicates are more than just a set of arguments for which they are true (e.g. they are also discriminated by their name).

The difference between intensionality and extensionality is a fundamental one and, while at first sight intensionality may seem more natural for real world relations, extensionality better materialises our mathematical definition of what a relation is. To better understand the above, consider the following example from [3].

Example 2.6. Consider the predicate `all_members(L, P)`, which is true if all elements of a list `L` have property `P`:

```
all_members([], P).
all_members([H|T], P) :- P(H), all_members(T, P).
```

If there exist two predicates `p` and `q` in our program that are true for exactly the same terms, e.g. the terms `a`, `b` and `c`, then we would expect both the following queries to succeed:

```
?- all_members([a,b,c], p).
yes

?- all_members([a,b,c], q).
yes
```

However, this is not guaranteed under the context of an intensional language.

The notions of extensional higher-order logic programming and constructive negation proved to be very compatible and, based on that observation, the $\mathcal{H}_{\text{cn}}$ language was created [4].

3. THE HIGHER-ORDER LANGUAGE $\mathcal{H}_{\text{cn}}$

3.1 Preliminaries

The reader is presumed to be familiar with the basic concepts and definitions of classical Logic Programming [6]. For example, the concepts **term**, **formula**, **atom**, **(program) clause** and **goal** are defined as in classical Logic Programming, with slight extensions on their definitions where necessary.

3.2 Basic Definitions

We describe the language by giving some definitions from the text where $\mathcal{H}_{\text{cn}}$ is defined [4], for the readers' convenience. These definitions are crucial to the understanding of the rest of the text.

Definition 1. *Types*

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

The types of $\mathcal{H}_{\text{cn}}$ are defined as follows:

$$\begin{array}{lcll} \sigma & := & \iota \mid (\iota \to \sigma) & \textit{(functional)} \\ \rho & := & \iota \mid \pi & \textit{(argument)} \\ \pi & := & o \mid (\rho \to \pi) & \textit{(predicate)} \end{array}$$

The argument type ρ consists of the following subtypes:

$$\begin{array}{lcll} \mu & := & \iota \mid \kappa & \textit{(existential)} \\ \kappa & := & \iota \to o \mid (\iota \to \kappa) & \textit{(set)} \end{array}$$

The operator $\to$ is right-associative.

Therefore, a functional type that is different from ι can also be written as $\iota^{\text{n}} \to \iota$, $\text{n} \geq 1$. Similarly, a set type can also be written in the form $\iota^{\text{n}} \to o$, $\text{n} \geq 1$. Finally, every predicate type π can be written in the form $\rho_1 \to \cdots \to \rho_{\text{n}} \to o$, $\text{n} \geq 0$ (for $\text{n} = 0$ we assume that $\pi = o$).

Definition 2. *Alphabet*

The alphabet of $\mathcal{H}_{\text{cn}}$ consists of:

1. *Predicate variables* of every predicate type π.
2. *Predicate constants* of every predicate type π.
3. *Individual variables* of type ι.
4. *Individual constants* of type ι.
5. *Function symbols* of every functional type $\sigma \neq \iota$.

6. The following logical constant symbols:
   (a) the *propositional constants* false and true of type o;
   (b) the *equality constant* $\approx$ of type $\iota \to \iota \to o$;
   (c) the *generalised disjunction and conjunction constants* $\bigvee_\pi$ and $\bigwedge_\pi$ of type $\pi \to \pi \to \pi$, for every predicate type π;
   (d) the *equivalence constants* $\leftrightarrow_\pi$ of type $\pi \to \pi \to o$, for every predicate type π;
   (e) the *existential quantifiers* $\exists_\mu$ of type $(\mu \to o)$, for every existential type μ;
   (f) the *negation constant* $\sim$ of type $o \to o$.
7. The *abstractor* λ and the *parentheses* "(" and ")".

We define the set consisting of the predicate variables and the individual variables of $\mathcal{H}_{\mathsf{cn}}$, as the *argument variables* of $\mathcal{H}_{\mathsf{cn}}$.

Definition 3. *Body expressions*

The set of body expressions of $\mathcal{H}_{\mathsf{cn}}$ is recursively defined as follows:

1. (a) Every predicate variable (respectively, predicate constant) of type π is a body expression of type π;
   (b) Every individual variable (respectively, individual constant) of type ι is a body expression of type ι;
   (c) The propositional constants false and true are body expressions of type o.
2. If f is an n-ary function symbol and $\mathsf{E}_1, \dots, \mathsf{E}_\mathsf{n}$ are body expressions of type ι, then $(\mathsf{f}\ \mathsf{E}_1\ \cdots\ \mathsf{E}_\mathsf{n})$ 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.

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

In the following, we will write $\widehat{\mathsf{A}}$ to denote a (possibly empty) sequence $\langle \mathsf{A}_1, \dots, \mathsf{A}_\mathsf{n} \rangle$.

A body expression of the form $(\mathsf{E}_1 \approx \mathsf{E}_2)$ will be called an *equality*, while a body expression of the form $\sim \exists \widehat{\mathsf{V}}\, (\mathsf{E}_1 \approx \mathsf{E}_2)$ will be called an *inequality*; in the latter case $\widehat{\mathsf{V}}$ may be empty, in which case the inequality is of the form $\sim (\mathsf{E}_1 \approx \mathsf{E}_2)$.

Let $\widehat{\mathsf{E}}$ and $\widehat{\mathsf{E}}'$, where all E_i, E'_i are of type ι. We will write $\left(\widehat{\mathsf{E}} \approx \widehat{\mathsf{E}}'\right)$ to denote the expression $(\mathsf{E}_1 \approx \mathsf{E}'_1) \wedge \cdots \wedge (\mathsf{E}_n \approx \mathsf{E}'_n)$; if $n = 0$, then the conjunction is the constant true.

Definition 4. *Clausal expressions*

The set of clausal expressions of $\mathcal{H}_{\mathsf{cn}}$ 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 $\mathcal{H}_{\mathsf{cn}}$, 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 (existential), then $\mathsf{false} \leftarrow_o \mathsf{E}$ (or $\leftarrow_o \mathsf{E}$, or just $\leftarrow \mathsf{E}$ is a clausal expression of $\mathcal{H}_{\mathsf{cn}}$, 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 $\mathcal{H}_{\mathsf{cn}}$, also called a *completion expression*.

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

Definition 5. *Program*

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

Definition 6. *Completed definition of predicate*

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 $\mathsf{E} = \lambda\widehat{\mathsf{X}}.\mathsf{false}$.

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

Definition 7. *Completion of program*

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

3.3 Examples

The following examples consist valid higher-order logic programs with respect to $\mathcal{H}_{\mathsf{cn}}$.

Example 3.1. Consider the following program:

```
p(Q) :- Q(a), Q(b).
```

When taking into account that `p` represents a set, we can read the above rule as: "`p` is a predicate that is true for all relations that contain, at least, `a` and `b`".

Therefore, given the query:

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

the implementation is expected to answer something of the form:

```
yes
R = { a, b } ∪ L
```

Following $\mathcal{H}_{\text{cn}}$ syntax, the above rule would be written as:

```
p ← λQ.((Q a) ∧ (Q b))
```

In the above notation, notice how the parameter list of the definition of `p` is placed on the right hand side (because of `p` being defined as a lambda expression).

Example 3.2. Consider the following higher-order logic program:

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

The above rule states that `B` is a band if it contains at least one singer and one guitarist. Notice how `S` and `G` are first-order variables, while `B` is a higher-order (to be exact, second-order) one.

One interesting thing to notice here, is how `B` is constructed. Subgoal `singer{G}` will enumerate all possible singers, while subgoal `guitarist{G}` will enumerate all possible guitarists. Finally, subgoal `B{X}` “associates” (includes) the respective `X` element to the answer set.

Therefore, given a knowledge base consisting of singers and guitartists, the query:

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

will return in `B` all the combinations of the singers and guitartists that may consist a band (with respect to the definition given by the program).

Following $\mathcal{H}_{\text{cn}}$ syntax, the above rule would be written as:

```
band ← λB.∃S,G((singer S) ∧ (B S) ∧ (guitarist G) ∧ (B G))
```

and the query, would be written as:

```
← band B
```

As explained in [4], $\mathcal{H}_{\text{cn}}$ operates under the assumption that free variables in a query are the ones for which the proof procedure will attempt to find bindings in order for the goal to be satisfied. That is why variable `B` above is not existentially quantified.

4. THE PROOF PROCEDURE

The proof procedure of $\mathcal{H}_{\mathrm{cn}}$ incorporates and adapts the constructive negation rules, as described by David Chan in [2], to the higher-order case.

Before outlining the procedure itself, we give the following definition, which categorises the various inequalities based on the expressions on their left and right hand side ([1], [4]).

Definition 8. An inequality $\sim \exists \widehat{\mathsf{V}} (\mathsf{E}_1 \approx \mathsf{E}_2)$ is considered:

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

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

4.1 The procedure

In this section, we define the proof procedure, as proposed by Charalambidis et al. in [4]. The procedure consists of three definitions. We explain each definition in its own section.

Definition 9. *Single-step derivation*

Let P be a program and let G_k and $\mathsf{G}_{\mathsf{k}+1}$ be goal clauses.

Let G_k be a conjunction $\leftarrow \mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_\mathsf{n}$, where each A_i is a body expression of type o.

Let A_i be one of $\mathsf{A}_1, \ldots, \mathsf{A}_\mathsf{n}$ and let us call it the *selected expression*.

Let $\mathsf{A}' = \mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_{\mathsf{i}-1} \wedge \mathsf{A}_{\mathsf{i}+1} \wedge \cdots \wedge \mathsf{A}_\mathsf{n}$.

We say that $\mathsf{G}_{\mathsf{k}+1}$ *is derived in one step* from G_k using θ, if one of the following conditions applies:

1. if A_i is true and $\mathsf{n} > 1$, then $\mathsf{G}_{\mathsf{k}+1} = \leftarrow \mathsf{A}'$ is derived from G_k using $\theta = \epsilon$;
2. if A_i is $(\mathsf{E}_1 \vee \mathsf{E}_2)$, then $\mathsf{G}_{\mathsf{k}+1} = \leftarrow \mathsf{A}_1 \wedge \cdots \wedge \mathsf{E}_\mathsf{j} \wedge \cdots \wedge \mathsf{A}_\mathsf{n}$ is derived from G_k using $\theta = \epsilon$, where $\mathsf{j} \in \{1, 2\}$;
3. if A_i is $(\exists \mathsf{V}\ \mathsf{E})$, then $\mathsf{G}_{\mathsf{k}+1} = \leftarrow \mathsf{A}_1 \wedge \cdots \wedge \mathsf{E} \wedge \cdots \wedge \mathsf{A}_\mathsf{n}$ is derived from G_k using $\theta = \epsilon$;
4. if $\mathsf{A}_\mathsf{i} \rightsquigarrow \mathsf{A}'_\mathsf{i}$, namely A_i is reduced to A'_i, then $\mathsf{G}_{\mathsf{k}+1} = \leftarrow \mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}'_\mathsf{i} \wedge \cdots \wedge \mathsf{A}_\mathsf{n}$ is derived from G_k using $\theta = \epsilon$;
5. if A_i is $(\mathsf{E}_1 \approx \mathsf{E}_2)$, then $\mathsf{G}_{\mathsf{k}+1} = \leftarrow \mathsf{A}'\theta$ is derived from G_k using $\theta = \mathsf{mgu}(\mathsf{E}_1, \mathsf{E}_2)$;
6. if A_i is $\left(\mathsf{R}\ \widehat{\mathsf{E}}\right)$ and R is a predicate variable of type κ then $\mathsf{G}_{\mathsf{k}+1} = \leftarrow \mathsf{A}'\theta$ is derived from G_k using $\theta = \left\{\mathsf{R}\middle/\left(\lambda\widehat{\mathsf{X}}.\left(\widehat{\mathsf{X}} \approx \widehat{\mathsf{E}}\right) \vee_\kappa \mathsf{R}'\right)\right\}$, where R' is a fresh predicate variable of type κ;

7. if A_i is $\sim \exists\widehat{V}\ E$ and A_i is negatively-reduced to A_i', then $G_{k+1} = \leftarrow A_1 \wedge \cdots \wedge A_i' \wedge \cdots \wedge A_n$ is derived from G_k using $\theta = \epsilon$;

8. if A_i is $\sim \exists\widehat{V}\left(R\ \widehat{E}\right)$ and R is a predicate variable of type κ and $R \notin \widehat{V}$, then $G_{k+1} = \leftarrow A'\theta$ is derived from G_k using $\theta = \left\{R\middle/\left(\lambda\widehat{X}.\ \sim \exists\widehat{V}\left(\widehat{X} \approx \widehat{E}\right) \bigwedge_\kappa R'\right)\right\}$, where R' is a fresh predicate variable of type κ;

9. is A_i is $\sim \exists\widehat{V} \sim \left(R\ \widehat{E}\right)$ and R is a predicate variable of type κ and $R \notin \widehat{V}$, then $G_{k+1} = \leftarrow A'\theta$ is derived from G_k using $\theta = \left\{R\middle/\left(\lambda\widehat{X}.\ \exists\widehat{V}\left(\widehat{X} \approx \widehat{E}\right) \bigvee_\kappa R'\right)\right\}$.

Definition 9 defines the single-step derivation of goals and is actually an extension of single-step derivation of classical logic programming to support higher-order terms and constructive negation. This definition is the most general of the three, in the sense that it delegates the handling of more “special” or complex subgoals to the other two definitions.

Definition 10. *Reduction*

Let P be a program and E, E' be body expressions of type o.

We say that E is reduced (with respect to P) to E' (denoted as $E \rightsquigarrow E'$) if one of following conditions applies:

1. $p\ \widehat{A} \rightsquigarrow E\ \widehat{A}$, where E is the completed expression for p with respect to P;
2. $(\lambda X.E)\ B\ \widehat{A} \rightsquigarrow E\{X/B\}\ \widehat{A}$;
3. $(E_1 \bigvee_\pi E_2)\ \widehat{A} \rightsquigarrow \left(E_1\ \widehat{A}\right) \vee_o \left(E_2\ \widehat{A}\right)$;
4. $(E_1 \bigwedge_\pi E_2)\ \widehat{A} \rightsquigarrow \left(E_1\ \widehat{A}\right) \wedge_o \left(E_2\ \widehat{A}\right)$.

Definition 10 is the simplest definition among the three. It corresponds to simple operations on a subgoal, in order to advance the its evaluation. More specifically, definition 10.1 replaces a predicate constant by its defined expressions within the program. Definition 10.2 corresponds to a β-reduction for a lambda expression, whereas definitions 10.3 and 10.4 “transform” the generalised disjunction and conjunction predicate constants $\bigvee_\pi$ and $\bigwedge_\pi$ to the normal boolean operators $\vee$ and $\wedge$, respectively.

Definition 11. *Negative reduction*

Let P be a program and B, B' be body expressions where $B = \sim \exists\widehat{U}\,(A_1 \wedge \cdots \wedge A_n)$ and each A_i is a body expression except for conjunction.

Let A_i be the selected expression and $A' = A_1 \wedge \cdots \wedge A_{i-1} \wedge A_{i+1} \wedge \cdots \wedge A_n$.

Then, we say that B is *negatively-reduced* to B' if one of the following conditions applies:

1. if A_i is false, then $B' = \mathsf{true}$;
2. if A_i is true and $n = 1$, then $B' = \mathsf{false}$, else $B' = \sim \exists\widehat{U}\ A'$;
3. if A_i is $(E_1 \vee E_2)$, then $B' = B_1' \wedge B_2'$, where $B_j' = \sim \exists\widehat{U}\,(A_1 \wedge \cdots \wedge E_j \wedge \cdots \wedge A_n)$, for $j \in \{1, 2\}$;

4. if $\mathsf{A_i}$ is $(\exists \mathsf{V}\ \mathsf{E})$, then $\mathsf{B}' = \sim \exists \widehat{\mathsf{U}}\mathsf{V}\,(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{E} \wedge \cdots \wedge \mathsf{A_n})$;
5. if $\mathsf{A_i} \rightsquigarrow \mathsf{A_i'}$, namely $\mathsf{A_i}$ is reduced to $\mathsf{A_i'}$, then $\mathsf{B}' = \sim \exists \widehat{\mathsf{U}}\,(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A_i'} \wedge \cdots \wedge \mathsf{A_n})$;
6. if $\mathsf{A_i}$ is $(\mathsf{E}_1 \approx \mathsf{E}_2)$, then:
   (a) if $\sim \exists \widehat{\mathsf{U}}\,(\mathsf{E}_1 \approx \mathsf{E}_2)$ is valid, then $\mathsf{B}' = \mathsf{true}$;
   (b) if $\sim \exists \widehat{\mathsf{U}}\,(\mathsf{E}_1 \approx \mathsf{E}_2)$ is non-valid and neither E_1 nor E_2 is a variable, then $\mathsf{B}' = \sim \exists \widehat{\mathsf{U}}\left(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_{i-1} \wedge \left(\widehat{\mathsf{X}} \approx \widehat{\mathsf{X}}\theta\right) \wedge \mathsf{A}_{i+1} \wedge \cdots \wedge \mathsf{A_n}\right)$, where $\theta = \mathsf{unify}\,(\mathsf{E}_1, \mathsf{E}_2)$ and $\widehat{\mathsf{X}} = \mathsf{dom}\,(\theta)$;
   (c) if $\sim \exists \widehat{\mathsf{U}}\,(\mathsf{E}_1 \approx \mathsf{E}_2)$ is unsatisfiable and either E_1 or E_2 is a variable in $\widehat{\mathsf{U}}$, then $\mathsf{B}' = \sim \exists \widehat{\mathsf{U}}\,(\mathsf{A}'\theta)$, where $\theta = \{\mathsf{X}/\mathsf{E}\}$ and X is the one expression that is a variable in $\widehat{\mathsf{U}}$ and E is the other;
   (d) if $\sim \exists \widehat{\mathsf{U}}\,(\mathsf{E}_1 \approx \mathsf{E}_2)$ is primitive and $\mathsf{n} > 1$, then $\mathsf{B}' = \sim \exists \widehat{\mathsf{U}}_1\ \mathsf{A_i} \vee \exists \widehat{\mathsf{U}}_1 \left(\mathsf{A_i} \wedge \sim \exists \widehat{\mathsf{U}}_2\ \mathsf{A}'\right)$, where $\widehat{\mathsf{U}}_1$ are the variables in $\widehat{\mathsf{U}}$ that are free in $\mathsf{A_i}$ and $\widehat{\mathsf{U}}_2$ are the variables in $\widehat{\mathsf{U}}$ not in $\widehat{\mathsf{U}}_1$;
7. if $\mathsf{A_i}$ is $\left(\mathsf{R}\ \widehat{\mathsf{E}}\right)$ and R is a predicate variable, then:
   (a) if $\mathsf{R} \in \widehat{\mathsf{U}}$, then $\mathsf{B}' = \sim \exists \widehat{\mathsf{U}}'\,(\mathsf{A}'\theta)$, where substitution $\theta = \{\mathsf{R}/(\lambda \mathsf{X}.\,(\mathsf{X} \approx \mathsf{E}) \bigvee_\kappa \mathsf{R}')\}$, R' is a predicate variable of the same type as R and $\widehat{\mathsf{U}}'$ is the same as $\widehat{\mathsf{U}}$, except that variable R has been replaced with R';
   (b) if $\mathsf{R} \notin \widehat{\mathsf{U}}$ and $\mathsf{n} > 1$, then $\mathsf{B}' = \sim \exists \widehat{\mathsf{U}}_1\ \mathsf{A_i} \vee \exists \widehat{\mathsf{U}}_1 \left(\mathsf{A_i} \wedge \sim \exists \widehat{\mathsf{U}}_2\ \mathsf{A}'\right) \wedge \mathsf{B}$, where $\widehat{\mathsf{U}}_1$ are the variables in $\widehat{\mathsf{U}}$ that are free in $\mathsf{A_i}$ and $\widehat{\mathsf{U}}_2$ are the variables in $\widehat{\mathsf{U}}$ not in $\widehat{\mathsf{U}}_1$;
8. if $\mathsf{A_i}$ is $\sim \exists \widehat{\mathsf{V}}\ \mathsf{E}$ and $\mathsf{A_i}$ is negatively-reduced to $\mathsf{A_i'}$, then $\mathsf{B}' = \sim \exists \widehat{\mathsf{U}}\,(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A_i'} \wedge \cdots \wedge \mathsf{A_n})$;
9. if $\mathsf{A_i}$ is a primitive inequality $\sim \exists \widehat{\mathsf{V}}\,(\mathsf{E}_1 \approx \mathsf{E}_2)$, then:
   (a) if $\mathsf{A_i}$ contains free variables in $\widehat{\mathsf{U}}$ and A' is a conjunction of primitive inequalities, then $\mathsf{B}' = \sim \exists \widehat{\mathsf{U}}\ \mathsf{A}'$;
   (b) if $\mathsf{A_i}$ does not contain any free variables in $\widehat{\mathsf{U}}$, then $\mathsf{B}' = \exists \widehat{\mathsf{V}}\,(\mathsf{E}_1 \approx \mathsf{E}_2) \vee \sim \exists \widehat{\mathsf{U}}\ \mathsf{A}'$;
10. if $\mathsf{A_i}$ is $\sim \exists \widehat{\mathsf{V}} \left(\mathsf{R}\ \widehat{\mathsf{E}}\right)$ and $\mathsf{R} \notin \widehat{\mathsf{V}}$ is a predicate variable, then:
    (a) if $\mathsf{R} \in \widehat{\mathsf{U}}$, then $\mathsf{B}' = \sim \exists \widehat{\mathsf{U}}'\,(\mathsf{A}'\theta)$, where substitution $\theta = \left\{\mathsf{R}\big/\left(\lambda \mathsf{X}.\ \sim \exists \widehat{\mathsf{V}}\,(\mathsf{X} \approx \mathsf{E}) \bigwedge_\kappa \mathsf{R}'\right)\right\}$, R' is a predicate variable of the same type κ as R and $\widehat{\mathsf{U}}'$ is the same as $\widehat{\mathsf{U}}$, except that variable R has been replaced with R';
    (b) if $\mathsf{R} \notin \widehat{\mathsf{U}}$ and $\mathsf{n} > 1$, then $\mathsf{B}' = \sim \exists \widehat{\mathsf{U}}_1\ \mathsf{A_i} \vee \exists \widehat{\mathsf{U}}_1 \left(\mathsf{A_i} \wedge \sim \exists \widehat{\mathsf{U}}_2\ \mathsf{A}'\right) \wedge \mathsf{B}$, where $\widehat{\mathsf{U}}_1$ are the variables in $\widehat{\mathsf{U}}$ that are free in $\mathsf{A_i}$ and $\widehat{\mathsf{U}}_2$ are the variables in $\widehat{\mathsf{U}}$ not in $\widehat{\mathsf{U}}_1$;
    (c) if $\mathsf{R} \notin \widehat{\mathsf{U}}$, $\mathsf{n} = 1$ and $\widehat{\mathsf{V}}$ is non-empty, then $\mathsf{B}' = \exists \widehat{\mathsf{V}} \sim \exists \widehat{\mathsf{U}} \left(\sim \left(\mathsf{R}\ \widehat{\mathsf{E}}\right) \wedge \sim \exists \widehat{\mathsf{V}}' \left(\mathsf{R}\ \widehat{\mathsf{E}}'\right)\right)$;

11. if $\mathsf{A_i}$ is $\sim \exists \widehat{\mathsf{V}} \sim \left(\mathsf{R}\ \widehat{\mathsf{E}}\right)$ and R is a predicate variable and $\mathsf{R} \notin \widehat{\mathsf{V}}$, then:

    (a) if $\mathsf{R} \in \widehat{\mathsf{U}}$, then $\mathsf{B}' = \sim \exists \widehat{\mathsf{U}}' \,(\mathsf{A}'\theta)$, where substitution $\theta = \left\{\mathsf{R}\big/\left(\lambda \mathsf{X}.\exists \widehat{\mathsf{V}}\,(\mathsf{X} \approx \mathsf{E}) \bigvee_{\kappa} \mathsf{R}'\right)\right\}$, R' is a predicate variable of the same type κ as R and $\widehat{\mathsf{U}}'$ is the same as $\widehat{\mathsf{U}}$, except that variable R has been replaced with R';

    (b) if $\mathsf{R} \notin \widehat{\mathsf{U}}$ and $\mathsf{n} > 1$, then $\mathsf{B}' = \sim \exists \widehat{\mathsf{U}}_1\ \mathsf{A_i} \vee \exists \widehat{\mathsf{U}}_1 \left(\mathsf{A_i} \wedge \sim \exists \widehat{\mathsf{U}}_2\ \mathsf{A}'\right) \wedge \mathsf{B}$;

    (c) if $\mathsf{R} \notin \widehat{\mathsf{U}}$, $\mathsf{n} = 1$ and $\widehat{\mathsf{V}}$ is non-empty, then $\mathsf{B}' = \exists \widehat{\mathsf{V}} \sim \exists \widehat{\mathsf{U}} \left(\left(\mathsf{R}\ \widehat{\mathsf{E}}\right) \wedge \sim \exists \widehat{\mathsf{V}}' \sim \left(\mathsf{R}\ \widehat{\mathsf{E}}'\right)\right)$.

Definition 11 is the most complex of the three definitions that make up the proof procedure. The reason is that it handles negative subgoals. In order to achieve that, it has to employ the constructive negation paradigm as defined by Chan [2], however extended in order to support higher order terms. These higher-order terms are in essence set variables, therefore the procedure in our case shall not only return "equality" relations, but also "belongs to" relations.

4.2 Extending constructive negation for $\mathcal{H}$

Recall that predicates in $\mathcal{H}$ represent sets. In $\mathcal{H}$, a set is constructed via the $\bigvee_{\pi}$ operator, (of type $\pi \to \pi \to \pi$) which is a primitive and generalised version of the set union operator ($\cup$).

Example 4.1. Lets assume that a predicate P in a program is true for the atoms a, b and c. Therefore, it represents the following set that contains (at least) a, b and c:

$$\mathsf{P} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\} \cup \mathsf{L}$$

The above bracket-notion is just a short (or pretty-print) version of the following:

$$\mathsf{P} = \{\mathsf{a}\} \bigvee_{\pi} \{\mathsf{b}\} \bigvee_{\pi} \{\mathsf{c}\} \bigvee_{\pi} \mathsf{L}$$

In order to extend the constructive negation rule for the higher-order logic programming language $\mathcal{H}$, we have to keep in mind that we are also dealing with higher-order (set) variables like in the example above.

Recall equation 2.1, which outlines the rule that is applied in constructive negation, when negating equality subgoals. This rule works well when first order literals (equalities or inequalities) are selected and is employed in rule 6(d) of definition 11. We now want to extend this rule to support higher-order predicate variables. The reason why we extend the aforementioned rule is because it is the rule that handles equalities. In the higher-order case, the analogous of an equality, is an element belonging to a set.

When considering the higher-order terms of $\mathcal{H}$, then a more sophisticated version of that formula is required. The reason is that higher-order terms are not "flat". To understand what that means, let us consider the principal difference between first and higher-order logic programming: the latter allows the usage of uninstantiated predicate variables, which denote sets. Since they denote sets, we would like to express two properties:

- that an element belongs to a set.
- that an element does not belong to a set.

The first point is implemented via positive computation logic, while the second point requires negative computation logic. Furthermore, the first point may also be the result of the application of double negation.

What the implementation is supposed to do when it encounters a higher-order (predicate) variable, is to perform a systematic enumeration of the possible values that variable can take. The variable itself represents a set, therefore the implementation must be able to generate multiple values that "belong" to the set that the variable represents.

Suppose that a negated expression (goal list) B contains the subgoal $(\mathsf{R}\ \mathsf{E})$, where R is a predicate variable and E is an expression. The way R will be constructed is by first adding variables to it and then producing bindings for those variables. It is clear that the formula in equation 2.1 is not enough to add multiple values to R, because the two terms of the conjunction will argue about whether E, and only E, belongs or not to the set.

What we would like to do is to continue arguing about more variables, i.e. try to satisfy the rest of the goal list if R contained more variables. In essence, what we want to do is, after having argued about R containing E, to argue about R not containing anything else or R containing one more element. If in every step we repeat this process, we can argue about R containing as many elements as it can.

The above idea translates to appending B (the negated expression) itself in conjunction with the second term of the disjunction in rule 2.1. This is the approach that $\mathcal{H}_{\mathsf{cn}}$ takes and is reflected in rules 11.7 (b), 11.10 (b) and 11.11 (b). Notice however, that this approach automatically makes the rule (i.e. the procedure) recursive. And, as we know, for a recursive procedure to stop at some point, there has to be some condition that, when satisfied, stops it. The above procedure seems to have none. We analyse this issue further in the next section.

Notice however that, theoretically, the above procedure is perfectly fine, because the rules do not make any assumption about the expression selection policy. There exist paths in the computation tree that "escape" from the infinite recursion and a non-deterministic policy may follow these paths. The problem is that when it comes to implementing things (writing algorithms), we cannot employ non-deterministic strategies.

4.3 Identifying the Problem

While being theoretically correct, the above approach does not work if implemented in a naive way (for example, selecting the left-most reducable literal in every step of the derivation). The reason is the repeated subgoal B. As discussed earlier, when a higher-order variable R is encountered, B is re-added to the goal list in order to explore whether R can contain extra elements. However, notice that B is repeated exactly "as is", without any care being taken in order to restrict the new values that are to be added to R.

If we now look at this procedure from an implementation point of view, it is only logical that exactly the same computation steps will be repeated, because B is exactly the same as before. It seems inevitable that the new goal will produce exactly the same bindings as the previous one. And this is actually the case, which is easily demonstrable if we look at the proof procedure computation steps of an example.

Example 4.2. Because program predicates represent sets, the most typical example we can investigate and which also shows off the problem pretty well, is the `subset` predicate, which was first presented in example 1.1.

This time we make sure that our program also contains some facts, so that we can run `subset` against the facts' predicate. So, consider the following higher-order logic program:

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

q(0).
q(1).
q(2).
```

and the query:

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

which asks for all subsets of q. Since q represents the set $\{0, 1, 2\}$, normally, 8 answers are expected and their order does not matter. Let us see what happens if we run the query by hand, highlighting the rules of the proof procedure that are applied at each step. In each step, we will show the pending goal list, as well as the unifications that are produced at that point of the computation.

The steps shown next are the steps that the current implementation follows, when given the above program and query. Please note that a left to right derivation order, just like in PROLOG, is followed. Also note that, following PROLOG notation, a comma (`,`) represents a logical AND, while a semicolon (`;`) represents a logical OR.

Step 1: The first step is just the expansion of the `subset` query:

```
λ P Q. ∼(non_subset(P, Q))(P, q) ?
```

Step 2: By applying reduction rule 10.2, we get:

```
∼(non_subset(P, q)) ?
```

Step 3: Next, the `non_subset` predicate is expanded:

```
∼(λ P Q. (∃X P(X), ∼Q(X))(P, q)) ?
```

Step 4: By applying reduction rule 10.2, we get:

```
∼(∃V4 P(V4), ∼q(V4)) ?
```

Notice that in this step that, while variable `X` is renamed to `V4`, the query variable `P` is left intact.

Step 5: This is the first interesting step of the example. We are inside the scope of a negated expression, therefore one of the negative reduction (definition 11) rules will be applied. Notice that, with respect to the aforementioned definition, $\widehat{\mathsf{U}} = \langle \texttt{V4} \rangle$. Since the selected literal is `P(V4)`, rule 11.7 matches, with $\mathsf{R} = \texttt{P}$ and $\widehat{\mathsf{E}} = \langle \texttt{V4} \rangle$. Since $\mathsf{R} \notin \widehat{\mathsf{U}}$ and $n = 2$, we follow case (b):

```
∼(∃V4 P(V4));
(∃V4 P(V4), ∼(∼q(V4))), ∼(∃V4 P(V4), ∼q(V4)) ?
```

The above is an application of the extended constructive negation rule for higher-order predicate variables, as proposed in [4].

Step 6: We now have two paths to follow (logical OR). Rule 9.2 is applied and, initially, the first path is followed:

```
∼(∃V4 P(V4)) ?
```

Step 7: Rule 9.8 is applied and substitution $\theta = \{\mathtt{P}/\,(\lambda\mathtt{X}.\sim\exists\mathtt{V4}(\mathtt{X}=\mathtt{V4})\bigwedge_\kappa \mathtt{P'})\}$ is produced. What this substitution says is that there doesn't exist any variable in `P`, i.e. `P` is the empty set. The goal list is now empty, and the first answer is returned:

```
yes
P = { }
```

So far so good, the empty set is a subset every possible set.

Step 8: At this point, the implementation backtracks and the second path of step 5 is followed. Notice that in the goal list there are two mathematical contexts, as suggested by the parentheses and the variable quantifications.

```
(∃V4 P(V4), ∼(∼q(V4))), ∼(∃V4 P(V4), ∼q(V4)) ?
```

Step 9: The selected expression is `(∃V4 P(V4), ∼(∼q(V4)))`, which matches with rule 9.3, where $\mathsf{V} = \mathtt{V4}$ and $\mathsf{E} = \mathtt{P(V4),\ \sim(\sim q(V4))}$.

```
P(V5), ∼(∼q(V5)), ∼(∃V4 P(V4), ∼q(V4)) ?
```

Notice that the `∃V4` quantification is removed and a fresh variable is used.

Step 10: The selected expression is now `P(V5)`, which matches rule 9.6, where $\mathsf{R} = \mathtt{P}$ and $\widehat{\mathsf{E}} = \langle\mathtt{V5}\rangle$. Therefore, substitution $\theta = \{\mathtt{P}/\,(\lambda\mathtt{X}.(\mathtt{X}=\mathtt{V5})\bigvee_\kappa \mathtt{V8})\}$ is produced and the goal list becomes:

```
∼(∼q(V5)), ∼(∃V4 ((λX.X=V5) ∨κ V8)(V4), ∼q(V4)) ?
```

Step 11: The selected expression is now `∼(∼q(V5))`. Since `q` is a predicate constant, it will be replaced by the completed definition for `q` with respect to the input program (definition 6). That is, $\mathsf{q} \leftrightarrow_\pi \mathsf{E}$, where $\mathsf{E} = (\lambda\mathsf{X}.\mathsf{X}=0)\bigvee_\kappa(\lambda\mathsf{X}.\mathsf{X}=1)\bigvee_\kappa(\lambda\mathsf{X}.\mathsf{X}=2)$.

```
∼(∼((λX.X=0) ∨κ (λX.X=1) ∨κ (λX.X=2))(V5)),
∼(∃V4 ((λX.X=V5) ∨κ V8)(V4), ∼q(V4)) ?
```

Step 12: After reducing the completed expression with `V5` according to definition 10 and applying the distributive law for the inner negation (∼), we end up with the following goal list:

```
∼(∼((λX.X=0)(V5)), ∼((λX.X=1)(V5)), ∼((λX.X=2)(V5))),
∼(∃V4 ((λX.X=V5) ∨κ V8)(V4), ∼q(V4)) ?
```

And after reducing the first expression of the completed definition, we get:

```
∼(∼(V5=0), ∼((λX.X=1)(V5)), ∼((λX.X=2)(V5))),
∼(∃V4 ((λX.X=V5) ∨κ V8)(V4), ∼q(V4)) ?
```

Step 13: We are inside the scope of a negated expression where $\widehat{\mathrm{U}}$ is empty and the selected expression is ∼(V5=0), which is a primitive inequality (definition 8). Rule 11.9 (b) matches and the resulting goal list is:

```
V5=0; ∼(∼((λX.X=1)(V5)), ∼((λX.X=2)(V5))),
∼(∃V4 ((λX.X=V5) ∨κ V8)(V4), ∼q(V4)) ?
```

The logical OR takes precedence and creates two possible paths. Initially, the first one is followed:

```
V5=0, ∼(∃V4 ((λX.X=V5) ∨κ V8)(V4), ∼q(V4)) ?
```

The selected expression is V5=0, which is a simple unification. Rule 9.5 is applied, which gives:

```
∼(∃V4 ((λX.X=0) ∨κ V8)(V4), ∼q(V4)) ?
```

Here the situation becomes interesting, because we are now ready to compute the $\ldots \wedge \mathrm{B}$ part of the constructive negation rule (definition 11.7 (b)).

Step 14: We are once more inside the scope of a negated expression where $\widehat{\mathrm{U}} = \langle \mathtt{V4} \rangle$. The selected expression is ((λX.X=0) ∨κ V8)(V4) and after applying the reduction rule 3 from definition 10, we get:

```
∼(∃V4 (λX.X=0)(V4); V8(V4), ∼q(V4)) ?
```

Now, the logical OR again takes precedence, thus the selected expression is (λX.X=0)(V4); V8(V4). This means that rule 11.3 is applied and the resulting goal list is:

```
∼(∃V4 (λX.X=0)(V4), ∼q(V4)), ∼(∃V4 V8(V4), ∼q(V4)) ?
```

Which after reducing the lambda expression (rule 10.2), becomes:

```
∼(∃V4 V4=0, ∼q(V4)), ∼(∃V4 V8(V4), ∼q(V4)) ?
```

And after applying the unification, gives:

```
∼(∃V4 ∼q(0)), ∼(∃V4 V8(V4), ∼q(V4)) ?
```

Step 15: We have to substitute the completed definition of q in place of the predicate constant q. After substituting, reducing and distributing the negation constant, as we previously did in steps 11 and 12, we get:

```
∼(∃V4 ∼((λX.X=0)(0)), ∼((λX.X=1)(0)), ∼((λX.X=2)(0))),
∼(∃V4 V8(V4), ∼q(V4)) ?
```

Furthermore, we reduce the first lambda expression of the completed expression:

```
∼(∃V4 ∼(0=0), ∼((λX.X=1)(0)), ∼((λX.X=2)(0))),
∼(∃V4 V8(V4), ∼q(V4)) ?
```

Step 16: The selected expression is now ∼(0=0), which is a negated expression. Therefore, the have to “descend” one level down (inside the negated expression) and the selected expression becomes 0=0. Rule 11.6 (b) is applied and the resulting expression is true. Therefore, the goal list becomes:

```
∼(∃V4 ∼(true), ∼((λX.X=1)(0)), ∼((λX.X=2)(0))),
∼(∃V4 V8(V4), ∼q(V4)) ?
```

Equivalently, after applying rule 11.1:

```
∼(∃V4 false, ∼((λX.X=1)(0)), ∼((λX.X=2)(0))),
∼(∃V4 V8(V4), ∼q(V4)) ?
```

The first negated expression is now a conjunction of expressions, the first of which is `false`, therefore the whole expression is `true` (rule 11.1 again), thus the remaining goal list is:

```
∼(∃V4 V8(V4), ∼q(V4)) ?
```

There is one important observation to make at this point. The above goal list is exactly the same as the goal list in step 4, except for one difference: instead of predicate variable `P` we now have `V8`.

Recall that, from step 10, predicate variable `P` is bound to $(\lambda\mathtt{X}.(\mathtt{X=V5})\ \bigvee_\kappa\ \mathtt{V8})$ and, from step 13, variable `V5` is bound to `0`. Therefore, $\mathtt{P\ =\ 0}\ \bigvee_\kappa\ \mathtt{V8}$.

This should give more clear view of how the extended constructive negation rule for higher-order predicates works. Higher-order variable `V8` occured from step 9 and acts as the "tail" of `P`. However, should the rule not contain the $\cdots \wedge \mathsf{B}$ part, it would never have a chance to be computed.

Step 17: At this point, the constructive negation rule (definition 11.7 (b)) is applied again:

```
∼(∃V4 V8(V4));
(∃V4 V8(V4), ∼(∼q(V4))), ∼(∃V4 V8(V4), ∼q(V4)) ?
```

As in step 6, the first path is followed:

```
∼(∃V4 V8(V4)) ?

true ?
```

and the second answer is returned:

```
yes
P = { 0 }
```

The computation continues with the second path:

```
(∃V4 V8(V4), ∼(∼q(V4))), ∼(∃V4 V8(V4), ∼q(V4)) ?
```

Since the rules that are applied are identical as before, in the following we omit detailed explanation of every rule application.

Step 18: Rule 9.3 is applied and a fresh variable is introduced in place of `V8` in the previously quantified context:

```
V8(V12), ∼(∼q(V12)), ∼(∃V4 V8(V4), ∼q(V4)) ?
```

Step 19: The selected literal is now `V8(V12)`, therefore rule 9.6 is applied and substitution $\theta = \{\mathtt{V8}/\ (\lambda\mathtt{X}.(\mathtt{X} = \mathtt{V12})\bigvee_\kappa \mathtt{V15})\}$ is produced.

```
∼(∼q(V12)), ∼(∃V4 ((λX.X = V12) ∨κ V15)(V4), ∼q(V4)) ?
```

Step 20: As before, the completed definition of predicate q is replaced in place of predicate constant q.

```
∼(∼((λX.X=0) ∨κ (λX.X=1) ∨κ (λX.X=2))(V12)),
∼(∃V4 ((λX.X=V12) ∨κ V15)(V4), ∼q(V4)) ?
```

which, as before, becomes:

```
∼(∼((λX.X=0)(V12)), ∼((λX.X=1)(V12)), ∼((λX.X=2)(V12))),
∼(∃V4 ((λX.X=V12) ∨κ V15)(V4), ∼q(V4)) ?
```

And after reducing the first expression of the completed definition, we get:

```
∼(∼(V12=0), ∼((λX.X=1)(V12)), ∼((λX.X=2)(V12))),
∼(∃V4 ((λX.X=V12) ∨κ V15)(V4), ∼q(V4)) ?
```

Application of rule 11.9 (b) gives:

```
V12=0; ∼(∼((λX.X=1)(V12)), ∼((λX.X=2)(V12))),
∼(∃V4 ((λX.X=V12) ∨κ V15)(V4), ∼q(V4)) ?
```

Which gives:

```
V12=0, ∼(∃V4 ((λX.X=V12) ∨κ V15)(V4), ∼q(V4)) ?
```

Recall that the composition of unifiers up to this point have bounded P to be P = 0 $\vee_\kappa$ (λX.(X=V12) $\vee_\kappa$ V15). After applying the unification and reducing, we get:

```
∼(∃V4 (λX.X=0)(V4); V15(V4), ∼q(V4)) ?
```

So P becomes: P = 0 $\vee_\kappa$ 0 $\vee_\kappa$ V15.

Step 21: Skipping some rule applications identical to previous steps, the goal list becomes:

```
∼(∃V4 V15(V4), ∼q(V4)) ?
```

The constructive negation rule (definition 11.7 (b)) is applied at this point and the goal list becomes:

```
∼(∃V4 V15(V4));
(∃V4 V15(V4), ∼(∼q(V4))), ∼(∃V4 V15(V4), ∼q(V4)) ?
```

The first path is followed, which leads to true and the third answer is returned:

```
yes
P = { 0, 0 }
```

Steps 17 through 21 will repeat for ever, adding only zeroes to the returned answers. In the absence of restrictions for the next selected expression, the implementation will always add the first choice it finds. Furthermore, the rest of paths that would occur from selecting alternative expressions will never be followed.

If we consider the computation tree (show in figure 4.1 below), the above process would correspond to following the leftmost branch every time, thus ending up at the bottom left of the tree (which is of course of infinite depth assuming that the machine it runs on has infinite memory).

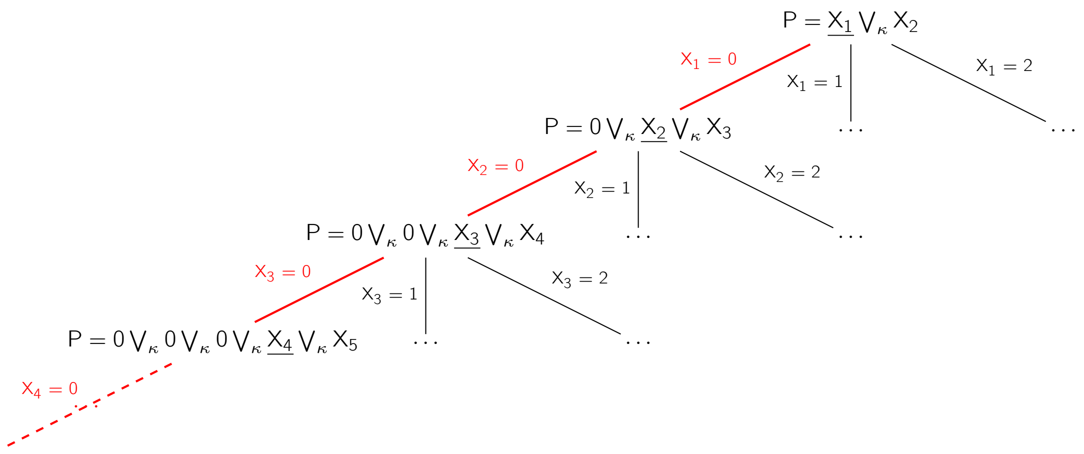
$P = \underline{X_1} \bigvee_\kappa X_2$
$X_1 = 0$
$X_1 = 1$
$X_1 = 2$
$P = 0 \bigvee_\kappa \underline{X_2} \bigvee_\kappa X_3$
$X_2 = 0$
$X_2 = 1$
$X_2 = 2$
$P = 0 \bigvee_\kappa 0 \bigvee_\kappa \underline{X_3} \bigvee_\kappa X_4$
$X_3 = 0$
$X_3 = 1$
$X_3 = 2$
$P = 0 \bigvee_\kappa 0 \bigvee_\kappa 0 \bigvee_\kappa \underline{X_4} \bigvee_\kappa X_5$
$X_4 = 0$

Figure 4.1: The (simplified) computation tree of the current implementation

5. THE PROPOSED APPROACH

5.1 Tweaking the proof procedure

Based on the previous chapter, it is obvious that any implementation of $\mathcal{H}_{\mathrm{cn}}$ has to take measures in order to make sure that the sets that are constructed via the constructive negation rule do not contain duplicate members. Therefore, some kind of restriction has to be put in place before the proof procedure continues with the computation of the "next" element.

This is the approach that we take. We introduce restrictions in the format of (in)equalities, when the constructive negation rule is applied. These (in)equalities have to be introduced in such a way, that will cause the computation to fail when an expression is to be "inserted" for the second time in a set and they will allow it to proceed otherwise.

Based on the above description, an obvious observation can be made. As the number of elements in a set increases, so does the number of the restrictive inequalities. This is normal, as we need an inequality per element currently in the set.

In this section we will define the modified proof procedure. In the next section we will present how and explain why the proposed approach works. Before outlining the modified versions of the rules of negative reduction (definition 11), we need to define the *trimmed substitution* operator.

Definition 12. *Trimmed substitution operator*

Let L be a lambda expression and P be a predicate variable of type κ.

The trimmed substitution operator S^{-} is defined as follows:

1. $\mathsf{S}^{-}(\mathsf{L} \bigvee_{\kappa} \mathsf{P}) = \mathsf{P}$;
2. $\mathsf{S}^{-}(\mathsf{L} \bigwedge_{\kappa} \mathsf{P}) = \mathsf{P}$.

Namely, given an expression that matches one of the above, S^{-} drops the lambda expression L and only returns the predicate variable P, i.e. it trims the expression to only contain P.

In the following, we will write $\left(\widehat{\mathsf{E}} \approx_{\vee} \widehat{\mathsf{E}'}\right)$ to denote the expression $(\mathsf{E}_1 \approx \mathsf{E}'_1) \vee \cdots \vee (\mathsf{E}_n \approx \mathsf{E}'_n)$; if $\mathsf{n} = 0$, then the disjunction is the constant false.

We can now give the modified negative reduction definition. Notice that we only need to tweak the rules that correspond to the constructive negation application for higher-order expressions, namely rules 7 (b), 10 (b) and 11 (b) of definition 11.

Definition 13. *Negative reduction (tweaked)*

Let P be a program and B, B' be body expressions where $\mathsf{B} = \sim \exists \widehat{\mathsf{U}}\,(\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n)$ and each A_i is a body expression except for conjunction.

Let A_i be the selected expression and $\mathsf{A}' = \mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_{i-1} \wedge \mathsf{A}_{i+1} \wedge \cdots \wedge \mathsf{A}_n$.

Then, we say that B is *negatively-reduced* to B' if one of the conditions of definition 11, overriden by the following rules where indicated, applies:

7. if A_i is $\left(R\ \widehat{E}\right)$ and R is a predicate variable, then:

   (b) if $R \notin \widehat{U}$ and $n > 1$, then $B' = \sim \exists \widehat{U}_1\ A_i \vee \exists \widehat{U}_1 \left(A_i \wedge \sim \exists \widehat{U}_2\ A'\ \wedge B_1\right)$,
   where $B_1 = \sim \exists \widehat{U}' \left(\left(S^-\ (R)\ \widehat{E}'\right) \wedge \left(A'' \vee \left(\widehat{E} \approx_V \widehat{E}'\right)\right)\right)$,
   $\widehat{U}_1$ are the variables in $\widehat{U}$ that are free in A_i, $\widehat{U}_2$ are the variables in $\widehat{U}$ not in $\widehat{U}_1$, $\widehat{U}'$ and $\widehat{E}'$ are renamed versions of variables in $\widehat{U}$ and $\widehat{E}$ respectively and A'' is the same as A', except that all occurrencies of R have been replaced with $S^-(R)$.

10. if A_i is $\sim \exists \widehat{V} \left(R\ \widehat{E}\right)$ and $R \notin \widehat{V}$ is a predicate variable, then:

   (b) if $R \notin \widehat{U}$ and $n > 1$, then $B' = \sim \exists \widehat{U}_1\ A_i \vee \exists \widehat{U}_1 \left(A_i \wedge \sim \exists \widehat{U}_2\ A' \wedge B_1\right)$,
   where $B_1 = \sim \exists \widehat{U}' \left(\left(S^-\ (R)\ \widehat{E}'\right) \wedge \left(A'' \vee \left(\widehat{E} \approx_V \widehat{E}'\right)\right)\right)$,
   $\widehat{U}_1$ are the variables in $\widehat{U}$ that are free in A_i, $\widehat{U}_2$ are the variables in $\widehat{U}$ not in $\widehat{U}_1$, $\widehat{U}'$ and $\widehat{E}'$ are renamed versions of variables in $\widehat{U}$ and $\widehat{E}$ respectively and A'' is the same as A', except that all occurrencies of R have been replaced with $S^-(R)$.

11. if A_i is $\sim \exists \widehat{V} \sim \left(R\ \widehat{E}\right)$ and R is a predicate variable and $R \notin \widehat{V}$, then:

   (b) if $R \notin \widehat{U}$ and $n > 1$, then $B' = \sim \exists \widehat{U}_1\ A_i \vee \exists \widehat{U}_1 \left(A_i \wedge \sim \exists \widehat{U}_2\ A' \wedge B_1\right)$,
   where $B_1 = \sim \exists \widehat{U}' \left(\left(S^-\ (R)\ \widehat{E}'\right) \wedge \left(A'' \vee \left(\widehat{E} \approx_V \widehat{E}'\right)\right)\right)$,
   $\widehat{U}_1$ are the variables in $\widehat{U}$ that are free in A_i, $\widehat{U}_2$ are the variables in $\widehat{U}$ not in $\widehat{U}_1$, $\widehat{U}'$ and $\widehat{E}'$ are renamed versions of variables in $\widehat{U}$ and $\widehat{E}$ respectively and A'' is the same as A', except that all occurrencies of R have been replaced with $S^-(R)$.

In order to introduce a restriction among the variables in the selected expression A_i and the ones in the repeated expression B, the latter needs to be within the scope of the quantification of $\widehat{U}_1$. That's why B was moved inside the aforementioned scope, in conjunction with the other subgoals and renamed to B_1 for the needs of the definition.

The first thing to observe here is that we have not introduced inequalities, but equalities! The reason is that these equalities are inside a negated expression and upon its evaluation they will turn into inequalities. Another interesting point is that the restrictive inequalities have been introduced in disjunction with the rest of the goal (A').

The detailed reasons for the above choices will become evident in the next section. Intuitively however, think again of expression B_1, which is a negated expression. If we distribute the negation (after evaluating $S^-\ (R)\ \widehat{E}'$, which causes the constructive negation rule for higher-order expressions to be applied again), the disjunction will become a conjunction and its expressions will become negated.

5.2 Why the approach works

It may be initially unclear why the proposed approach works and the truth is that it is not so obvious. It took a “reverse engineering” methodology in order to come up with it and once we delve into it, everything will become more clear.

First of all, let us consider the computation point that does not exhibit the desired behaviour. Look at step 20 of example 4.2. At that step, the implementation computes the second value of `P`, which is variable `V12`. There are three choices for `V12`: `0`, `1` or `2`. The computation procedure is obliged to check all of them. When it picks the first value (`V12 = 0`), there has to be something in the goal list that will cause the path that is currently being followed to fail.

To help us with the task, we consider a simplified snapshot of the computation procedure for the `subset` query. A trace that goes up to the second returned answer is sufficient in order to demonstrate the desired points:

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

   (1) ∼(non_subset(P, q)) ?

   (2) ∼(∃V4 P(V4), ∼q(V4)) ?

   (3) ∼(∃V4 P(V4));
       (∃V4 P(V4), ∼(∼q(V4))), ∼(∃V4 P(V4), ∼q(V4)) ?

   (4) ∼(∃V4 P(V4)) ?

   (5) true ?

   yes
   P = { }

   (6) (∃V4 P(V4), ∼(∼q(V4))), ∼(∃V4 P(V4), ∼q(V4)) ?

   (7) P(V5), ∼(∼q(V5)), ∼(∃V4 P(V4), ∼q(V4)) ?

   (8) ∼(∼q(V5)), ∼(∃V4 ((λX.X=V5) ∨κ V8)(V4), ∼q(V4)) ?

   ...

   (9) ∼(∼((λX.X=0)(V5)), ∼((λX.X=1)(V5)), ∼((λX.X=2)(V5))),
       ∼(∃V4 ((λX.X=V5) ∨κ V8)(V4), ∼q(V4)) ?

  (10) ∼(∼(V5 = 0), ∼((λX.X=1)(V5)), ∼((λX.X=2)(V5))),
       ∼(∃V4 ((λX.X=V5) ∨κ V8)(V4), ∼q(V4)) ?

  (11) V5 = 0; ∼(∼((λX.X=1)(V5)), ∼((λX.X=2)(V5))),
       ∼(∃V4 ((λX.X=V5) ∨κ V8)(V4), ∼q(V4)) ?

  (12) V5 = 0, ∼(∃V4 ((λX.X=V5) ∨κ V8)(V4), ∼q(V4)) ?

  (13) ∼(∃V4 ((λX.X=0) ∨κ V8)(V4), ∼q(V4)) ?

  (14) ∼(∃V4 ((λX.X=0)(V4); V8(V4), ∼q(V4)) ?

  (15) ∼(∃V4 ((λX.X=0)(V4), ∼q(V4)), ∼(∃V4 V8(V4), ∼q(V4)) ?
  (16) ∼(∃V4 V4 = 0, ∼q(V4)), ∼(∃V4 V8(V4), ∼q(V4)) ?

  (17) ∼(∃V4 ∼q(0)), ∼(∃V4 V8(V4), ∼q(V4)) ?

   ...

  (18) ∼(∃V4 ∼((λX.X=0)(0)), ∼((λX.X=1)(0)), ∼((λX.X=2)(0))),
       ∼(∃V4 V8(V4), ∼q(V4)) ?

  (19) ∼(∃V4 ∼(0 = 0), ∼((λX.X=1)(0)), ∼((λX.X=2)(0))),
       ∼(∃V4 V8(V4), ∼q(V4)) ?

  (20) ∼(∃V4 ∼(true), ∼((λX.X=1)(0)), ∼((λX.X=2)(0))),
       ∼(∃V4 V8(V4), ∼q(V4)) ?

  (21) ∼(∃V4 false, ∼((λX.X=1)(0)), ∼((λX.X=2)(0))),
       ∼(∃V4 V8(V4), ∼q(V4)) ?

  (22) ∼(∃V4 V8(V4), ∼q(V4)) ?

  (23) ∼(∃V4 V8(V4));
       (∃V4 P(V4), ∼(∼q(V4))), ∼(∃V4 V8(V4), ∼q(V4)) ?

  (24) ∼(∃V4 V8(V4)) ?

  (25) true ?

  yes
  P = { 0 }

  (26) (∃V4 V8(V4) ∼(∼q(V4))), ∼(∃V4 V8(V4), ∼q(V4)) ?

  (27) V8(V12), ∼(∼q(V12)), ∼(∃V4 V8(V4), ∼q(V4)) ?

  (28) ∼(∼q(V12)), ∼(∃V4 ((λX.X=V12) ∨κ V15)(V4), ∼q(V4)) ?

   ...

  (29) ∼(∼((λX.X=0)(V12)), ∼((λX.X=1)(V12)), ∼((λX.X=2)(V12))),
       ∼(∃V4 ((λX.X=V12) ∨κ V15)(V4), ∼q(V4)) ?

  (30) ∼(∼(V12 = 0), ∼((λX.X=1)(V12)), ∼((λX.X=2)(V12))),
       ∼(∃V4 ((λX.X=V12) ∨κ V15)(V4), ∼q(V4)) ?

  (31) V12 = 0; ∼(∼((λX.X=1)(V12)), ∼((λX.X=2)(V12))),
       ∼(∃V4 ((λX.X=V12) ∨κ V15)(V4), ∼q(V4)) ?

  (32) V12 = 0, ∼(∃V4 ((λX.X=V12) ∨κ V15)(V4), ∼q(V4)) ?

  (33) ∼(∃V4 ((λX.X=0) ∨κ V15)(V4), ∼q(V4)) ?

  (34) ∼(∃V4 (λX.X=0)(V4); V15(V4), ∼q(V4)) ?

  (35) ∼(∃V4 (λX.X=0)(V4), ∼q(V4)), ∼(∃V4 V15(V4), ∼q(V4)) ?

  ...

  (36) ∼(∃V4 V15(V4), ∼q(V4)) ?

  (37) ∼(∃V4 V15(V4));
```

```
      (∃V4 V15(V4), ∼(∼q(V4))), ∼(∃V4 V15(V4), ∼q(V4)) ?

  (38) ∼(∃V4 V15(V4)) ?

  (39) true ?

  yes
  P = { 0, 0 }
  ...
```

By carefully looking at the trace above, we notice two things:

- Look that the expression `((λX.X=0) ∨κ V15)` at step 28.

  This expression occurs from the substitution of variable `V8` at step 27. The path that is due to the lambda expression within the above expression (that is, the first context in step 35) contributes nothing to the computation. This is because in step 32, `V12` is being assigned a value from the completed definition of `q` and then the aforementioned path essentially asks whether the resulting expression is true for `q` (look at steps 15-21 for a detailed sequence of steps). Of course, that is always true. We utilise this property in our modified procedure (we essentially trim that path, recall definition 12 – trimmed substitution).

  The reason for trimming the aforementioned path is because, along with our introduced restrictions, it caused our modified procedure to fail. Since it contribues nothing to the computation, we can safely trim it.

- Now look at the second context of step 35, whose computation commences at step 36. Since its first expression is a higher-order expression, the constructive negation rule will be applied, leading to step 37. By then, it is too late to apply any restriction, because step 38 always succeeds. Therefore, we need to add our restriction in such a way, so that in step 32 there is another subgoal that disproves the `V12=0` equality, namely:

  ```
   (32) V12 = 0, ∼(V12 = 0), ∼(∃V4 ((λX.X=V12) ∨κ V15)(V4), ∼q(V4)) ?
  ```

Pay close attention to the variables that we want to restrict: variable `V12` will be the second variable to be added to `P` and we want to find an acceptable value for it. We want that value to be different from the value of the variable already in `P`. This means that any restriction has to be put into place while we still have access to the first variable of `P` (namely, `V5` from step 7).

Our modified definition adds the restriction upon the first application of the constructive negation rule, that is step 3. At that point, we do not yet know what value of the first variable will be, but we surely know that we want its value to be different from the value of the second variable.

Let us now show how the modified proof procedure works. The first step that changes is step 3. Instead of appending B to the goal list, we insert it into the context of the quantified `V4` variable and also include an equality in disjunction with A′:

```
 (3) ∼(∃V4 P(V4));
     (∃V4 P(V4), ∼(∼q(V4)), ∼(∃V5 S⁻(P)(V5), (∼q(V5); V4 = V5))) ?
```

The first expression in the disjunction remains the same, so it leads to the first answer being returned, as before.

```
   (4) ∼(∃V4 P(V4)) ?

   (5) true ?

   yes
   P = { }
```

The computation now continues with the second path:

```
   (6) (∃V4 P(V4), ∼(∼q(V4)), ∼(∃V5 S⁻(P)(V5), (∼q(V5); V4 = V5))) ?
```

Notice that now, the ∃V4 quantification is over the whole goal list, therefore every V4 variable gets renamed to V6.

```
   (7) P(V6), ∼(∼q(V6)), ∼(∃V5 S⁻(P)(V5), (∼q(V5); V6 = V5)) ?

   (8) ∼(∼q(V6)), ∼(∃V5 S⁻((λX.X=V6) ∨κ V9)(V5), (∼q(V5); V6 = V5)) ?

   ...

   (9) ∼(∼((λX.X=0)(V6)), ∼((λX.X=1)(V6)), ∼((λX.X=2)(V6))),
       ∼(∃V5 S⁻((λX.X=V6) ∨κ V9)(V5), (∼q(V5); V6 = V5)) ?

  (10) ∼(∼(V6 = 0), ∼((λX.X=1)(V6)), ∼((λX.X=2)(V6))),
       ∼(∃V5 S⁻((λX.X=V6) ∨κ V9)(V5), (∼q(V5); V6 = V5)) ?

  (11) V6 = 0; ∼(∼((λX.X=1)(V6)), ∼((λX.X=2)(V6))),
       ∼(∃V5 S⁻((λX.X=V6) ∨κ V9)(V5), (∼q(V5); V6 = V5)) ?

  (12) V6 = 0, ∼(∃V5 S⁻((λX.X=V6) ∨κ V9)(V5), (∼q(V5); V6 = V5)) ?

  (13) ∼(∃V5 S⁻((λX.X=0) ∨κ V9)(V5), (∼q(V5); 0 = V5)) ?
```

The first variable that has been inserted to P (that is V6) has gotten the value 0. At step 13 the implementation has to apply the trimmed substitution operator, which trims the lambda expression and returns only predicate variable V9:

```
  (14) ∼(∃V5 V9(V5), (∼q(V5); 0 = V5)) ?
```

At this point, the extended constructive negation rule is applied again and another restriction is introduced. The purpose of this restriction is to prevent the third variable that may later be added to P of being equal to the first or the second one.

```
  (15) ∼(∃V5 V9(V5));
       (∃V5 V9(V5), ∼(∼q(V5); 0 = V5),
          ∼(∃V13 S⁻(V9)(V13), (∼q(V13); 0 = V13; V5 = V13))) ?
```

In the above step 15, V13 is the next (third) variable to be inserted to P, 0 = V13 restricts it be different than the first one and V5 = V13 restricts it to be different than the second one.

Continuing with the computation, the first branch of step 15 leads to the second answer being returned:

```
  (16) ∼(∃V5 V9(V5)) ?

  (17) true ?

  yes
  P = { 0 }
```

And the second branch does its magic as follows:

```
(18) (∃V5 V9(V5), ∼(∼q(V5); 0 = V5),
        ∼(∃V13 S⁻(V9)(V13), (∼q(V13); 0 = V13; V5 = V13))) ?

(19) V9(V14), ∼(∼q(V14); 0 = V14),
        ∼(∃V13 S⁻(V9)(V13), (∼q(V13); 0 = V13; V14 = V13)) ?

(20) ∼(∼q(V14); 0 = V14),
        ∼(∃V13 S⁻((λX.X=V14) ∨κ V17)(V13), (∼q(V13); 0 = V13; V14 = V13)) ?

(21) ∼(∼q(V14)), not(0 = V14),
        ∼(∃V13 S⁻((λX.X=V14) ∨κ V17)(V13), (∼q(V13); 0 = V13; V14 = V13)) ?
```

Notice how at step 21 above, due to the negation, the equality in disjunction became an inequality in conjunction. The computation continues by replacing the completed definition of q and selecting the first possible value (V14=0).

```
(21) ∼(∼q(V14)), not(0 = V14),
        ∼(∃V13 S⁻((λX.X=V14) ∨κ V17)(V13), (∼q(V13); 0 = V13; V14 = V13)) ?

...

(22) ∼(∼((λX.X=0)(V14)), ∼((λX.X=1)(V14)), ∼((λX.X=2)(V14))), not(0 = V14),
        ∼(∃V13 S⁻((λX.X=V14) ∨κ V17)(V13), (∼q(V13); 0 = V13; V14 = V13)) ?

(23) ∼(∼(V14 = 0), ∼((λX.X=1)(V14)), ∼((λX.X=2)(V14))), not(0 = V14),
        ∼(∃V13 S⁻((λX.X=V14) ∨κ V17)(V13), (∼q(V13); 0 = V13; V14 = V13)) ?

(24) V14 = 0; ∼(∼((λX.X=1)(V14)), ∼((λX.X=2)(V14))), not(0 = V14),
        ∼(∃V13 S⁻((λX.X=V14) ∨κ V17)(V13), (∼q(V13); 0 = V13; V14 = V13)) ?

(25) V14 = 0, not(0 = V14),
        ∼(∃V13 S⁻((λX.X=V14) ∨κ V17)(V13), (∼q(V13); 0 = V13; V14 = V13)) ?
```

The goal list in step 25 above contains everything that we have been fighting for. Obviously the first two expressions form a contradiction and will cause this path to fail.

```
(26) not(0 = 0),
        ∼(∃V13 S⁻((λX.X=0) ∨κ V17)(V13), (∼q(V13); 0 = V13; 0 = V13)) ?

(27) not(true),
        ∼(∃V13 S⁻((λX.X=0) ∨κ V17)(V13), (∼q(V13); 0 = V13; 0 = V13)) ?

(28) false,
        ∼(∃V13 S⁻((λX.X=0) ∨κ V17)(V13), (∼q(V13); 0 = V13; 0 = V13)) ?
```

At this point the implementation will backtrack to step 24 and select the other path:

```
(25) ∼(∼((λX.X=1)(V14)), ∼((λX.X=2)(V14))), not(0 = V14),
        ∼(∃V13 S⁻((λX.X=V14) ∨κ V17)(V13), (∼q(V13); 0 = V13; V14 = V13)) ?

(26) ∼(∼(V14 = 1), ∼((λX.X=2)(V14))), not(0 = V14),
        ∼(∃V13 S⁻((λX.X=V14) ∨κ V17)(V13), (∼q(V13); 0 = V13; V14 = V13)) ?

(27) V14 = 1; ∼(∼((λX.X=2)(V14))), not(0 = V14),
        ∼(∃V13 S⁻((λX.X=V14) ∨κ V17)(V13), (∼q(V13); 0 = V13; V14 = V13)) ?

(28) V14 = 1, not(0 = V14),
        ∼(∃V13 S⁻((λX.X=V14) ∨κ V17)(V13), (∼q(V13); 0 = V13; V14 = V13)) ?
```

```
(29) not(0 = 1),
        ∼(∃V13 S⁻((λX.X=1) ∨κ V17)(V13), (∼q(V13); 0 = V13; 1 = V13)) ?
```

At step 29 above, rule 11.6 is applied. The inequality `not(0 = 1)` is valid, therefore the first expression evaluates to `true`.

```
(30) true,
     ∼(∃V13 S⁻((λX.X=1) ∨κ V17)(V13), (∼q(V13); 0 = V13; 1 = V13)) ?

(31) ∼(∃V13 S⁻((λX.X=1) ∨κ V17)(V13), (∼q(V13); 0 = V13; 1 = V13)) ?

(32) ∼(∃V13 V17(V13), (∼q(V13); 0 = V13; 1 = V13)) ?
```

The extended constructive negation rule is applied again and the first path of the resulting goal list will provide us with the third answer.

```
(33) ∼(∃V13 V17(V13));
     (∃V13 V17(V13), ∼(∼q(V13); 0 = V13; 1 = V13),
        ∼(∃V30 V17(V30), (∼q(V30); 0 = V30; 1 = V30; V13 = V30))) ?

(34) ∼(∃V13 V17(V13)) ?

(35) true ?

yes
P = { 0, 1 }

...
```

In step 33 above, a third restriction is placed, in order to restrict the possible values for the fourth member of `P`. Since the third member will be selected to be `2`, there is no possible value for the fourth member, so there will be no fourth member. The implementation will backtrack and will try to find an alternative value for the second member, then try to recompute a third element, etc.

The above procedure will continue until all possible combinations of valid values for the members of `P` have been checked. It is also obvious that the above procedure terminates, since the completed definition of `q` is finite and every value from the latter that is added to `P` is different from the values added before it.

5.3 Solving generate and test problems

The modified proof procedure allows us to easily select elements from a collection of objects. This enables us to solve generate and test problems fairly easily.

Example 5.1. Consider the following logic program:

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

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

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

The above program defines the `twocolor` predicate, which is true if `R` is a subset of the vertices of a graph `G` that can be painted with the same colour in a two-colouring of `G`. Note that in order to exist a valid two-coloring for a graph, we must be able to paint each pair of vertices consisting an edge with different colors.

The `twocolor` predicate first enumerates all subsets of vertices of `G` and then tests whether they can be painted with the same colour. The `vertex_set` predicate returns a set with all the vertices of `G`.

The `non_twocolor` predicate is interesting. It states that the set of vertices `R` of graph `G` cannot be painted with the same colour, if there exist some vertices `X` and `Y`, such that `X` and `Y` are adjacent in `G` and both `X` and `Y` are in `R`. Alternatively, vertices in `R` cannot be painted with the same colour, if there exist some vertices `X` and `Y`, such that `X` and `Y` are adjacent in `G` and neither of them is in `R` (because then there would not be enough colours remaining to paint `X` and `Y`).

If we have the following simple definition for a (undirected) graph:

```
graph(a, b).
graph(b, c).
```

Then the answers to the following query shall be (not necessarily in the indicated order):

```
?- two_color(graph, R).

yes
R = { b } ;

yes
R = { a, c } ;

no
```

Example 5.2. Consider the following logic program:

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

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

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

The above program defines the `clique` predicate, which is true if `R` is a subset of the vertices of a graph `G` that form a subgraph of `G`, which is a clique (every vertex is connected to every other vertex in the set).

Notice that the main skeleton of the program is the same as in example 5.1. We only added the `non_clique` predicate definition, which simply states that a subset of the vertices of a graph `G` is not a clique if there exist two nodes `X` and `Y` in `G`, such that there is no edge between them.

Suppose that we have the following simple definition for a (undirected) graph:

```
graph(a, b).
graph(b, c).
graph(c, a).
```

which is know as K_3 (the complete graph with 3 vertices).

Then the answers to the following query shall be (not necessarily in the indicated order):

```
?- clique(graph, R).

yes
R = { a } ;

yes
R = { a, b } ;

yes
R = { b } ;

yes
R = { b, c } ;

yes
R = { c } ;

yes
R = { a, b, c } ;

no
```

5.4 Limitations

While the approach outlined in the previous sections fixes the problem of non-termination when there exist higher-order negative expressions in the goal list, it stills produces duplicate answers, in the sense that it may return the same set more than once, with its elements in different order. This would normally cause the queries in the examples 5.1 and 5.2 to return the same answer more than once. However, this is a different and more fundamental problem, which is out of the scope of this thesis.

To understand why this problem exists, consider again the `subset` trace (example 4.2). Every time that the negation rule is applied on the expression that extends P, normally the definition of predicate q is inserted into the computation. The completed definition of q is always the same and the implementation has to choose one of the possible values (in example 4.2 the possible values are 0, 1 or 2). Our modified rule inserts inequalities in order to restrict the new value to be different from any previous value.

This is however not enough, because when the computation backtracks, the inequalities are lost (and are meant to be lost, so that new subsets can be computed). The computation finds itself in the following situation:

- After $\mathsf{P} = \{0, 1, 2\}$ has been computed, there is no other expression with which P can be extended and there is no other choice in place of the third member (2), therefore the implementation backtracks in order to find an alternative choice in place of the second member (1).
- Indeed, since there is only one restriction for the second member, namely to be different that 0, there is one alternative expression: 2. Therefore, the following set is computed: $\mathsf{P} = \{0, 2\}$.
- As the computation continues, it will attempt to find whether there is a third expression (from the completed definition of q) that can be “put” into P, that also satisfies

the restrictions of it being different than 0 and 2. There is such an expression: 1 is different than 0 and 2, so it is chosen and the following answer is computed: $\mathsf{P} = \{0, 2, 1\}$.

Clearly, another tweak is necessary in order to avoid the above behaviour. One idea would be, instead of inequalities, to introduce "greater than" restrictions for the members of P. However, for a complete set of answers to be computed, this would require the completed expression of q to be "sorted". When talking about numbers, sorting is obvious. However, in the general case that is not true. Furthermore, it is not straight-forward neither how one would sort the completed expression, nor how sorting would affect the performance of the implementation.

5.5 Implementation

A prototype implementation of the above modified procedure can be found at the following GitHub repository: `https://github.com/errikos/hopes` (branch `cn_restrict`). The aforementioned implementation is in essence the same with Angelos Charalambidis' implementation, with minor changes to reflect the modified procedure and it is written in Haskell.

6. CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

We modified the proof procedure of $\mathcal{H}_{\text{cn}}$ to make it more implementation-oriented. The new proof procedure avoids the non-termination problem that the previous one encountered when it had to deal with negated higher-order expressions.

We did so by introducing expressions that restrict the new elements of the respective set when applying the constructive negation rule to higher-order predicate expressions. These expressions are essentially inequalities which are handled by the proof procedure itself and emulate an $\in$ operator for the set constructed so far.

6.2 Future work

Avoiding multiple answers

While the modified proof procedure allows the implementation to find the correct paths in the computation tree, there is still room for improvement. One issue that has to be tackled is that the same answer may be returned multiple times, with its terms re-ordered (see “Limitations” – section 5.4). This may require extra restrictions placed in the proof procedure rules, as well as a different order of evaluation of program predicates.

One may argue that, since the arrangement of terms differs, the answers are not the same. However, recall that in $\mathcal{H}_{\text{cn}}$, predicates represent sets, which normally have no ordering. Therefore, in order to be compatible with the language semantics, an implementation ideally shall not return multiple answers.

Employing CLP techniques in $\mathcal{H}_{\text{cn}}$

While looking for possible solutions to the problem that this paper tries to address, one particular scheme kept coming forward. In every step made towards solving the problem, there was always the thought that everything would possibly be simpler if constraint logic programming (CLP) techniques were employed in order to restrict variable values.

Indeed, the concept of CLP seems to very compatible with the semantics of $\mathcal{H}_{\text{cn}}$. This is mainly due to the nature of constructive negation, which in addition to variable assignments, also generates inequalities. Handling these inequalities the way a CLP language does, could have multiple benefits for the implementation and this idea deserves further investigation.

Warren Abstract Machine for $\mathcal{H}_{\text{cn}}$

Recently, work has been done in extending the Warren Abstract Machine in order to implement higher-order logic programming languages [7]. It would be really interesting to use the extended WAM definition in order to implement $\mathcal{H}_{\text{cn}}$ and see how constructive negation behaves in that case.

ABBREVIATIONS - ACRONYMS

HOPES ($\mathcal{H}$)	Higher Order Prolog with Extensional Semantics
$\mathcal{H}_{\text{cn}}$	HOPES with Constructive Negation

BIBLIOGRAPHY

[1] David Chan. Constructive negation based on the completed database. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pages 111–125. MIT Press, 1988.

[2] David Chan. An extension of constructive negation and its application in coroutining. In Ewing L. Lusk and Ross A. Overbeek, editors, *Logic Programming, Proceedings of the North American Conference 1989, Cleveland, Ohio, USA, October 16-20, 1989. 2 Volumes*, pages 477–493. MIT Press, 1989.

[3] Angelos Charalambidis, Konstantinos Handjopoulos, Panagiotis Rondogiannis, and William W. Wadge. Extensional higher-order logic programming. *ACM Trans. Comput. Log.*, 14(3):21:1–21:40, 2013.

[4] Angelos Charalambidis and Panos Rondogiannis. Constructive negation in extensional higher-order logic programming. In Chitta Baral, Giuseppe De Giacomo, and Thomas Eiter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*. AAAI Press, 2014.

[5] Angelos Charalambidis, Panos Rondogiannis, and Antonis Troumpoukis. Higher-order logic programming: an expressive language for representing qualitative preferences. In James Cheney and Germán Vidal, editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 24–37. ACM, 2016.

[6] J. W. Lloyd. *Foundations of Logic Programming; (2nd Extended Ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[7] Alexandros Tasos. WAM extensions for implementing higher order logic languages, 2016.

[8] William W. Wadge. Higher-order horn logic programming. In Vijay A. Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991*, pages 289–303. MIT Press, 1991.