UNIVERSIDAD AUTÓNOMA DE MADRID

UNDERGRADUATE THESIS

DOUBLE DEGREE IN MATHEMATICS AND COMPUTER SCIENCE

Linearisation of dimension bounded sets of Horn clauses

Author:
Elena GUTIÉRREZ VIEDMA

Supervisor:
Jesús MUNÁRRIZ ALDAZ

May 20, 2016

Abstract

A *logic program* is a set of first-order logic sentences namely *Horn clauses* that define relations between objects. The *meaning* of a program is the result of computing the consequences of the program. An *atom* is an expression of the form $p(t_1, ...t_m)$ where p is a predicate symbol and each t_i is either a variable or a constant. A Horn clause is a disjunction of literals (an atom or its negation) and can be represented in terms of a logic implication ($\rightarrow$). We write Horn clauses in its reverse implication form, i.e., we use a right-to-left implication ($\leftarrow$). Thus, the right side of the reverse implication is the *body* of the Horn clause and the left side is the *head*. The body of a Horn clause is a (possibly empty) conjunction of atoms called *subgoals*. *Linear clauses* are clauses containing zero or one subgoal. *Linear programs* are programs with at least one linear clause, otherwise they are *non-linear*. *Formal verification* is the use of logic formal methods to prove the correctness of programs. Such a logic formal method is, for instance, translating a program into a logic program and check its satisfiability, i.e., the existence of an interpretation that satisfies all the clauses in the program. Such an interpretation is called a *solution* of the program. *Model checkers* and *solvers* are the tools dedicated to check satisfiability of programs. Most of them are applied to non-linear programs but there are notable exceptions. This thesis presents a procedure that transforms non-linear programs from a syntactic class of programs into linear programs preserving their meaning. This syntactic class of programs is the set of *dimension bounded programs*. We are restricted to this set of programs because they are guaranteed to be linearisable, i.e, there exists an algorithm that transforms these programs into linear that terminates and preserves their meaning. We prove that our procedure terminates and also preserves the meaning between the initial dimension bounded program and the linearised program. We also describe the notion of *dimension* of a program and its interpretation. We define the set of rules, that applied to a set of Horn clauses P, gives as a result a dimension bounded program $P^{[k]}$, given the value of the dimension $k \in \mathbb{N}$. Finally, we will see in which cases we can conclude if a non-linear program P is satisfiable or not depending on the satisfiability of the transformed program, i.e., the program that results from applying the linearisation procedure to $P^{[k]}$ for a given $k \in \mathbb{N}$.

Contents

1 Introduction **1**
1.1 Logic and Computation 1
1.2 Logic Programming 2
1.3 Motivation 3

2 Preliminaries **5**
2.1 Horn clauses 5
2.2 Constrained Logic Programs or CLPs 6
2.2.1 Meaning and equivalence between CLPs 7
2.3 Linear and piecewise linear CLPs 8
2.3.1 Linear CLPs 8
2.3.2 Piecewise linear CLPs 9

3 Dimension bounded sets of Horn clauses **13**
3.1 Preliminaries 13
3.2 Derivation tree and tree dimension 14
3.3 At-most-k-dimension programs $P^{[k]}$ 16

4 Linearisation Procedure **21**
4.1 Preliminaries 21
4.2 Linearisation procedure applied to dimension bounded programs . 24
4.2.1 Clause Linearisation Procedure 29
4.2.2 Program Linearisation Procedure 30
4.2.3 Example 30
4.2.4 Correctness of ELP 34

5 Conclusions **37**

List of Figures

2.1 Dependence graph of CLP in Example 2.3.4 10
2.2 The SCC of the dependence graph in Figure 2.1 10

3.1 A derivation tree of the Fibonacci program in Example 2.2.4. 15
3.2 Dimension tree corresponding to the derivation tree in Figure 3.1 . 16

4.1 Unfolding tree w.r.t. c6. vía S 26
4.2 Tree T 27
4.4 E-linearisable upper portion U of a I-tree w.r.t. P and clause $c3$ . . . 31
4.5 E-linearisable upper portion U of a I-tree w.r.t. P and clause $c5$ . . . 31
4.6 F-linearisable upper portion U' of U w.r.t. ED 31
4.7 F-linearisable upper portion U_{d1} w.r.t. ED 32
4.8 E-linearisable upper portion U of a I-tree w.r.t. P and clause $c6$ . . . 32
4.9 F-linearisable upper portion U' of U w.r.t. ED 33
4.10 F-linearisable upper portion U_{d2} w.r.t. ED 33
4.11 E-linearisable upper portion U of a I-tree w.r.t. P and clause $c7$ . . . 33
4.12 F-linearisable upper portion U' of U w.r.t. ED 34
4.13 F-linearisable upper portion U_{d3} w.r.t. ED 34

Chapter 1

Introduction

1.1 Logic and Computation

Between the late 19th century and the early 20th century, one of the focuses of mathematical work was the study of the foundations of mathematics. This naturally led to the development of a new subfield of mathematics, *Mathematical Logic*.

Mathematicians began to search for an axiomatic framework to formalise a large number of areas such as set theory, geometry and arithmetic, whose bases became unstable because of the discovery of the set-theoretic paradoxes. The aim was to build a formal system where every mathematical truth could be formulated and proven in terms of a finite number of rules and axioms (or axiom schemes).

Furthermore, David Hilbert suggested to reduce all the existing theories to a complete and finite set of axioms and then prove the consistence of those axioms. Unfortunately, Kurt Gödel proved in his *Incompleteness theorem* (1931) that there were limits to what could be proved and disproved with a formal system. These results together with the question proposed by David Hilbert in 1928, known as *Hilbert's Entscheidungsproblem* – the existence or not of an *abstract machine* or *algorithm* able to decide whether a statement of a first-order logic is true or false, depending on whether the statement is valid in every structure that satisfies the axioms – motivated the work of the English mathematician Alan Turing and the American logician Alonzo Church.

Turing proved that there was no solution to the *Entscheidungsproblem* and formalised the concept of these *abstract machines* as hypothetical and simple mathematical models able to carry out any mathematical computation. This, together with the work of Church gave place to an important conjecture known as the *Church-Turing* thesis, which esentially affirms that everything that is computable can be carried out by those *abstract machines*, also called *Turing machines*. Their work constituted the foundations of *computation theory*, and opened the door to the development of different models of computation using mathematical logic.

As a result of the work of a number of theorists along the 20th century together with the development of the first general purpose computers, computation theory has experimented a specialisation on a large number of different areas.

In particular, *automated reasoning* attempts to achieve the techniques to *teach* computers how to *reason* completely, or near completely and *automatically*. This is the base of fields such as automated theorem proving, logic programming and formal verification.

Formal verification refers to the use of formal methods to prove the correctness of a certain algorithm or program according to a given specification of its behaviour. Such a formal method is, for instance, translating a program written in a certain programming language into a *logic program* (see section 1.2) and check if there exists a *solution* (see Definition 2.2.8) for the logic program, i.e., if it is satisifiable.

Logic programming plays also an important role as a tool that provides enough expressive power to formalise the behaviour of programs, to automatise reasonings and to prove certain properties of programs.

1.2 Logic Programming

Logic programming appears as point of encounter between computation and logic. A *logic program* consists of a set of first-order logic sentences which can be either facts or rules defining relations between objects.

A *computation* of a logic program is a deduction of consequences of the program. This set of consequences is the *meaning* of the program.

Logic programming does not deal with specific computer languages in the sense that it is derived from an abstract model, which has no direct relation to one machine model or another. In its purest form, logic programming suggests that instead of giving explicit instructions to solve a problem, it is preferable to state explicitly the knowledge about the problem and the sufficient assumptions to solve it. The program can be *executed* by providing it with a problem, formalised as a logical statement to be proved called *goal statement*. *Executing* a program means proving the goal statement, given the assumptions in the logic program.

An example of a *goal statement* is: "order the list [3,1,2] to obtain the object X". The mechanism used to prove the goal statement is *constructive* in the sense that if succesful, it provides the identity of the unknown individuals mentioned in the goal statement – the output of the computation. In this case, assuming we have defined the appropiate axioms that describe the *sort* relation, the output of the computation would be X =[1,2,3].

The expressive power of logic programming relies on particular first-order sentences called *Horn clauses*. Therefore, logic programs are a set of Horn clauses, either *rules* or *facts*, defining relations between objects. This set of Horn clauses are the *axioms* of the logic program.

Let us consider the following rule, which is a Horn clause, and its interpretation within a program:

H *if* B_1 *and* B_2 *and* B_3 *and ... and* B_n.

can be interpreted as follows: To solve (execute) H, solve (execute) B_1, and B_2 and ... and B_n.

H is said to be the *head* of the rule, while B_1... and B_n are called *subgoals* and they form the *body* of the rule.

Facts are also part of logic programs, and are also Horn clauses:

H.

equivalent to

H if true.

and can be interpreted as: H is true in our logic program.

1.3 Motivation

The aim of this thesis is to develop a *linearisation procedure* which transforms a set of *non-linear* Horn clauses into a set of *linear* clauses preserving their meaning. A Horn clause is *non-linear* if it contains more than one subgoal in the body. A Horn clause is *linear* if it contains zero or one subgoal in the body.

On the other hand, a *model checker* or *solver* is a verification tool that automatically checks the correctness of a program. We are interested in checking the satisfiability of a set of non-linear Horn clauses using a *linear* solver for Horn clauses. The linearisation procedure transforms a set of non-linear Horn clauses that can be linearised into a linear set of Horn clauses that can be solved using a linear Horn clause solver. At the end of the day, we are able to solve non-linear sets of Horn clauses using a solver for linear Horn clauses.

In section 2, we shall give a collection of definitions to introduce the subject. In particular, we shall describe the class of *piecewise linear* programs. These programs are proved to be *linearisable*, i.e., they can be transformed into *linear* programs by means of an algorithm that preserves their meaning and terminates. In section 3, we shall present the concept of *derivation tree* associated to a program and the *dimension* of a derivation tree. Also, we shall describe a collection of transformations that changes a set of Horn clauses into a new set whose derivation trees have bounded dimension. This will lead us to the notion of a *dimension bounded set of Horn clauses* as an *under-approximation* of a given program. We will see that dimension bounded programs are contained in the class of piecewise linear programs and therefore, dimension bounded programs are linearisable. In section 4, we shall give a definition of a *linearisation procedure* applied to k-dimension-bounded sets of Horn clauses, for a given natural value k.

The starting point of this work is the three internships I have carried out during the summers of 2013, 2014 and 2015 in the research institute IMDEA Software. However, this thesis goes beyond that work, expanding it with original research.

Chapter 2

Preliminaries

2.1 Horn clauses

Definition 2.1.1. *An **atom** is an expresion of the form $p(t_1, ..., t_m)$ where p is a predicate symbol of arity m and t_i is a term, i.e., a variable or a constant, with $1 \leq i \leq m$.*

In Example 2.2.4, fib(0,0), fib(1,1) and fib(X,Y) are examples of atoms. The three of them have the same predicate, fib, of arity 2. Atoms fib(0,0) and fib(1,1) have constants as arguments while fib(X,Y) has a vector of 2 variables: X and Y.

Definition 2.1.2. *A **ground atom** is an atom $p(t_1, ..., t_m)$ where t_i is a constant, with $1 \leq i \leq m$.*

In Example 2.2.4, fib(0,0) and fib(1,1) are ground atoms.

Definition 2.1.3. *A **literal** is an atom or its negation.*

Positive literals are atoms and we use them interchangeably in this thesis. In Example 2.2.4, fib(0,0), fib(1,1) and fib(X,Y) are examples of positive literals.

Definition 2.1.4. *A **clause** is a disjunction of literals.*

In practice, we restrict the set of *all* clauses to the set of *Horn clauses*, named after the logician Alfred Horn. These are clauses with at most one positive literal.

Definition 2.1.5. *A **Horn clause** is a clause with at most one positive, i.e., unnegated, literal, when written in a disjunctive form.*

Let $h, b_1, ..., b_n$ be predicate symbols of arity m. We write V_j to denote a (possibly empty) set of variables and we write X_j to denote the tuple of variables consisting of all variables in V_j. Let $X, X_1, ..., X_n$ be the tuples of variables in $V, V_1, ..., V_n$ respectively, with $V \subseteq V_1 \cup ... \cup V_n$.

- If $n > 0$ then $\neg b_1(X_1) \vee \ ... \ \vee \neg b_n(X_n) \vee h(X)$ is a definite clause or *rule*. It is common to write definite clauses in its *implication* form

  $$b_1(X_1) \wedge \ ... \ \wedge b_n(X_n) \rightarrow h(X). \tag{2.1}$$

  All variables in a clause are implicitly *universally quantified* with scope the entire clause. Thus, clause 2.1 stands for:

  $$\forall X \forall X_1 ... \forall X_n \ (b_1(X_1) \wedge \ ... \ \wedge b_n(X_n) \rightarrow h(X)). \tag{2.2}$$

  We usually write Horn clauses in its reverse form:

  $$h(X) \leftarrow b_1(X_1) \wedge \ ... \ \wedge b_n(X_n). \tag{2.3}$$

We refer to the right hand side of $\leftarrow$ as the *body* and the left hand side as the *head* of the Horn clause. In this case, $(b_1(X_1) \wedge\ ...\ \wedge b_n(X_n))$ is the *body* while $h(X)$ is the *head* of the clause.

In Example 2.2.4, clause c3. is a rule.

- If $n = 0$ then $h(X)$ is called a *fact*. It is logically equivalent to

  $$\forall X\ h(X)$$

  and to

  $$\forall X\ (h(X) \leftarrow true).$$

  In Example 2.2.4, clauses c1. and c2. are facts.

- If no positive literals occur in the clause, $\neg b_1(X_1) \vee\ ...\ \vee \neg b_n(X_n)$ is a *goal clause*. It is logically equivalent to

  $$\forall X\ (false \leftarrow b_1(X_1) \wedge\ ...\ \wedge b_n(X_n)),$$

  and to

  $$(\neg \exists X\ (b_1(X_1) \wedge\ ...\ \wedge b_n(X_n))).$$

  In Example 2.2.4, clause c4. is a goal clause.

We will use "," instead of "$\wedge$" to separate the atoms in the body of a clause.

2.2 Constrained Logic Programs or CLPs

Definition 2.2.1. *A* ***constraint*** *over the set of variables V_0 is a conjunction of linear equalities and inequalities over the variables in V_0 and the integers.*

Given a set of variables V_0 and the tuple of variables X_0 consisting of all variables in V_0, we will denote a constraint over the variables in V_0 by $C(X_0)$.

In general we will use the integers as our constraint theory. Thus, a constraint represents the set of integers that verifies its linear equalities and inequalities. The *negation* of a constraint, $\neg C(X_0)$, represents the complementary set of $C(X_0)$.

In Example 2.2.4, clauses c3. and c4. contain constraints in their bodies. In clause c3. the constraint is X > 1, X2 = X - 2, X1 = X - 1, Y = Y1 + Y2, while in clause c4. the constraint is X > 5, X > Y.

Definition 2.2.2. *A* ***constrained Horn clause*** *is a Horn clause of the form*

$$p(X) \leftarrow C(X_0), p_1(X_1), p_2(X_2), ..., p_n(X_n), \quad n \geq 0$$

where $p_1, ..., p_n,\ p$ are predicate symbols; $V, V_0, V_1, ..., V_n$ are (possibly empty) sets of variables with $V \subseteq V_0 \cup V_1 \cup ... \cup V_n$; $X, X_1, ..., X_n$ are the tuples of variables consisting of all variables in the sets $V, V_1, ..., V_n$ respectively; and $C(X_0)$ is a constraint over the variables in V_0.

In Example 2.2.4, every clause is a constrained Horn clause. For clauses c1. and c2., $n = 0$ while for clauses c3. and c4., $n > 0$. In clause c3., V is the set of variables {X, Y}, V_0 is the set {X, X_2, X_1, Y, Y_1, Y_2}, V_1 is the set {X_2, Y_2} and V_2 is the set {X_1, Y_1}.

Definition 2.2.3. *A **constrained logic program or CLP** is a finite set of constrained Horn clauses.*

We will refer to CLPs simply as *programs* and constrained Horn clauses as *clauses*.

Example 2.2.4. Example of CLP

c1. fib(0,0).
c2. fib(1,1).
c3. fib(X,Y) ← X > 1, X2 = X - 2, fib(X2, Y2), X1 = X - 1, fib(X1, Y1), Y = Y1 + Y2.
c4. false ← X > 5, fib(X,Y), X > Y.

Example 2.2.4 is a CLP that defines a Fibonacci function. In this case, fib(n,m) means that m is the n^{th} element in the Fibonacci sequence.

Clauses c1. and c2. express that the element in position 0 and position 1 in the Fibonacci sequence are 0 and 1, respectively. Clause c3. expresses that every other element in the Fibonacci sequence in a position greater than 1 is computed adding the values in the previous two positions. Clause c4. expresses that every element in the Fibonacci sequence in a position greater than 5, *is not* less than the position itself.

2.2.1 Meaning and equivalence between CLPs

Definition 2.2.5. *The **Herbrand Universe of a CLP** is the set of all ground atoms that can be formed using the constant symbols that appear in P.*

In Example 2.2.4, the set of constant symbols is $\mathbb{Z}$. Thus, the Herbrand Universe of this program is {fib(a,b) | a,b ∈ $\mathbb{Z}$}.

Definition 2.2.6. *An **interpretation of a CLP** is a set of ground atoms whose predicate symbols occur in the program and whose arguments are in the Herbrand Universe of the program.*

An interpretation for the CLP in Example 2.2.4 is {fib(-1,-1), fib(0,0)}.

Definition 2.2.7. *A **Herbrand model of a CLP** is an interpretation that satisfies all clauses in the program.*

A Herbrand model for the CLP in Example 2.2.4 is {fib(0,0), fib(1,1)}. The reader can check that both ground atoms satisfy all the clauses in the program.

Definition 2.2.8. *A **solution of a CLP** P, is a Herbrand model of P.*

In Example 2.2.4, *a* solution for the CLP given is {fib(0,0), fib(1,1), fib(2,1), fib(3,2), fib(4,3), fib(5,5)}. The reader can check that this is an interpretation that satisfies every clause in the CLP.

Definition 2.2.9. *The **meaning of a CLP** P, denoted by $M(P)$, is the set of all solutions of P.*

We say that two programs are *equivalent* if they have the same meaning. The meaning of the program in Example 2.2.4 is {fib(a,b) | a,b∈ $\mathbb{Z}$ and b is the a^{th} element in the Fibonacci sequence}.

Definition 2.2.10. *The **meaning of a CLP** P**, restricted to the predicates in** S, is defined as $M_S(P) = M(P) \cap \{A \mid A \text{ is a ground atom whose predicate is in } S\}$.*

Example 2.2.11. Example of the meaning of a CLP restricted to a set of predicates

Let P be the following CLP:

c1. triangular(X) $\leftarrow$ $Y_1 \geq 0, Y_2 \geq 0$, $Y_2 = Y_1 + 1$, X = (Y_1*Y_2)/2.
c2. square(X) $\leftarrow$ X$\geq$ 0.
c3. square(X) $\leftarrow$ $Y_1 \geq 0$, $Y_2 \geq 0$, $Y_2 = Y_1$+1, triangular(Y_1), triangular(Y_2), $X = Y_1 + Y_2$.

where triangular(X) means that X is a triangular number (a number that can be represented in the form of a triangular grid of points where the first row contains a single point and each subsequent row contains one more element than the previous one) and square(X) means that X is a square number (a number which is equal to the product of a natural number by itself). This program computes all the square numbers relying on the property that the sum of two consecutive triangular numbers is a square number.

Let S be the set of predicates {square}, the meaning of this CLP restricted to S is $M_S(P)$ ={square(a) | $a = b^2$ and $a, b \in \mathbb{N}$}.

Definition 2.2.12. ***(Meaning preservation)*** *Let P_1 be a program and $pred(P_1)$ the set of predicates occurring in P_1. We say that a set of transformations applied to P_1 that gives as a result program P_2* preserves the meaning *between them if $M(P_1) = M_{pred(P_1)}(P_2)$.*

As we shall see in Chapter 4 the introduction of new clauses that define new predicates is a step of the linearisation procedure here presented. Thus, when we say that the linearisation procedure preserves the meaning between the initial non-linear program P and the linearised program LP, we mean that the meaning of P is equal to the meaning of LP *restricted* to the predicates ocurring in P. We will say that the procedure preserves the meaning between the initial and the linearised program or that it *preserves the equivalence* between them.

2.3 Linear and piecewise linear CLPs

2.3.1 Linear CLPs

Definition 2.3.1. *A **subgoal** is an atom appearing in the body of a clause.*

In Example 2.2.4, the subgoals of clause c3. are fib(X2,Y2) and fib(X1,Y1).

Definition 2.3.2. ***Non-linear clauses*** *are those containing more than one subgoal. Otherwise, they are called* linear.

In Example 2.2.4, only clause c3. is non-linear, as it has 2 subgoals in its body.

Definition 2.3.3. *A **linear CLP** is a program where every clause is linear.*

If one or more non-linear clauses occur in the program, it is *non-linear*.
CLP in Example 2.2.4 is non-linear as c3. is non-linear.

Our aim is to *linearise* non-linear programs, i.e, transform a set of clauses which include at least one or more non-linear clauses into a set of linear clauses by means of a *linearisation procedure*. A natural question is: Can we linearise any non-linear program so that, every solution for the linear program is also a solution for the original, and viceversa? In other words, is there, for any non-linear program, an *equivalent* linear program? In [1] it is proven that the answer to this question is negative. However, in [1] it is also proved that there exists a syntactic class of non-linear programs, namely the *piecewise linear programs*, where every program can always be transformed into an equivalent linear program by means of a procedure that terminates and preserves the equivalence between them. In fact, the procedure presented here deals with a syntactic class of programs, namely *dimension bounded programs*, which is contained in the class of piecewise linear programs as it will be shown in the next chapter. By describing this procedure we will prove constructively that non-linear dimension bounded programs are also linearisable.

2.3.2 Piecewise linear CLPs

Example 2.3.4. Example of CLP

c1. arc(1,2).
c2. arc(2,3).
c3. path(X,Y) $\leftarrow$ arc(X,Y).
c4. path(X,Y) $\leftarrow$ path(X,Z),path(Z,Y).

Example 2.3.4 shows a CLP that describes the relation *path* between two points by means of the relation *arc*. Clauses c1. and c2. state that there is an arc between points 1 and 2, and between points 2 and 3. Clause c3. states that there exists a path between 2 given points X and Y if there exists an arc between them. Clause 4. states that there exists a path bewteen 2 points if there exists a path between the first one and an intermediate point, and a path bewteen the intermediate point and the second one. We will use Example 2.3.4 to illustrate the following definitions.

Definition 2.3.5. ***Dependence graph of a CLP*** *P. Let P be a CLP. The dependence graph of P is the directed graph, where loops are allowed but multiple edges are not, constructed as follows:*

For each predicate symbol p appearing in P, there is exactly one node representing p. For each clause c in P whose head is an atom with predicate p, and for each predicate q of a subgoal in c, there is exactly one directed edge from p to q. If there is already a directed edge from p to q, no new directed edge from p to q is added.

The node in the dependence graph of P, representing the predicate symbol p, will be named as *node* p.

Figure 2.1 shows the dependence graph of the CLP in Example 2.3.4. As there are two different predicates in the CLP, the graph has two nodes. There exists an edge from node *path* to node *arc* because the head of clause c1. is an atom whose predicate is *path* and *arc* is the predicate ocurring in its subgoal. There exists an edge from node *path* to itself because the head of clause c2. is an atom whose predicate is *path*, and the predicate of the first subgoal ocurring in the body of c2. is also *path*. Note that the predicate of the second subgoal ocurring in the body of c2. is *path* as well. However, no new edges from node *path* to itself are added.

Definition 2.3.6. ***Strongly connected components (SCCs) of the dependence graph.*** *Given two subgraphs G_1 and G_2 of the dependence graph, we say that $G_1 < G_2$ under the inclusion ordering if and only if the set of nodes of G_1 is contained in the set of nodes of*

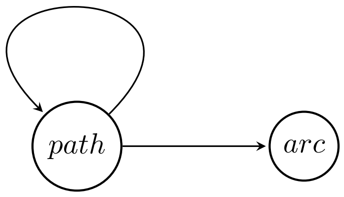


FIGURE 2.1: Dependence graph of CLP in Example 2.3.4

G_2 and the set of edges of G_1 is contained in the set of edges of G_2.
The strongly connected components of the dependence graph are the maximal subgraphs of the dependence graph (w.r.t. the inclusion ordering) such that every node is reachable (there exists a non-empty sequence of edges) from all the nodes in the subgraph.

Figure 2.2 shows an SCC of the dependence graph of the CLP in Example 2.3.4. Node *path* is itself an SCC because it is reachable from itself.

This is the only SCC of the dependence graph of CLP in Example 2.3.4. Node *arc* is not an SCC as it is not reachable from itself. The union of nodes *path* and *arc* is not an SCC either, because node *path* cannot be reached from node *arc* and, again, node *arc* can not be reached from itself.

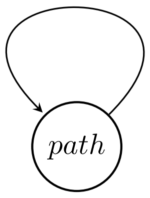


FIGURE 2.2: The SCC of the dependence graph in Figure 2.1

Definition 2.3.7. *A* ***recursive clause*** *is a clause that has a subgoal whose predicate is represented by a node in the same SCC as the node representing the predicate in its head.*

Clause c2. in Example 2.3.4 is a recursive clause since predicate *path*, appearing in both subgoals of its body, belongs to the same SCC as the predicate in its head (which is also *path*).

Definition 2.3.8. *A* ***recursive subgoal*** *in a recursive clause is a subgoal in the same SCC as the head of the clause. Otherwise, it is called a* non-recursive *subgoal.*

In the recursive clause c2. in Example 2.3.4, both subgoals in its body are recursive.

Definition 2.3.9. *A* ***piecewise linear program*** *is a program P where every clause has at most one recursive subgoal.*

It is easy to see that every *linear* program is *piecewise linear*, while piecewise linear programs are not guaranteed to be linear in general.

The CLP shown in Example 2.3.4 is not piecewise linear as clause c2. has two recursive subgoals.

Example 2.3.10. Example of CLP

```
c1. arc(1,2).
c2. arc(2,3).
c3. path(X,Y) ← arc(X,Y).
c4. path(X,Y) ← arc(X,Z),path(Z,Y).
```

The CLP shown in Example 2.3.10 is piecewise linear as clause c3. has no recursive subgoals and clause c4. has exactly one recursive subgoal.

Chapter 3

Dimension bounded sets of Horn clauses

3.1 Preliminaries

In this section we will introduce the notions of *unification*, *most general unifier* and *most specific generalisation*.

Definition 3.1.1. *Let Q be a (non-empty) conjunction of atoms and constraints, V_1 the set of all the variables ocurring in Q, V_2 a set of variables and C a set of constants. A **substitution** for Q is a total function $\sigma : V_1 \to V_2 \cup C$ extended in the natural way to conjuctions of atoms and constraints. The result of the substitution is an* instance *of the original conjunction of atoms and constraints.*

Example 3.1.2. Example of substitution

Let $\sigma = \{X \mapsto Z, Y \mapsto 0, W \mapsto 1\}$ be a substitution. Given the conjunction (fib(X,Y), W≥ 0, fib(W,1)) then:
(fib(X,Y), W≥ 0, fib(W,1))σ = ((fib(X,Y))σ, (W≥ 0)σ, (fib(W,1))σ) =
(fib((X)σ,(Y)σ), (W)$\sigma \geq 0$, fib((W)σ,1) = (fib(Z,0), 1≥ 0, fib(1,1)).

Definition 3.1.3. *Given the (non-empty) conjunctions of atoms and constraints D and D', D' **unifies** with D if there exists a substitution σ such that $D'\sigma = D$.*

We say that D' is *more general* than, or *subsumes* D; and that D is *unifiable* with, or *more special* than D'.
We call σ the unifier of D (w.r.t. D').

Example 3.1.4. Example of unifiers of a given atom

Given the atom D' = path(X,X):

1. D = path(1,1) is unifiable with D' as the substitution $\sigma = \{X \mapsto 1\}$ verifies $D'\sigma = D$.

2. D = path(X,Y) is not unifiable with D'.
   $\sigma = \{X \mapsto X, X \mapsto Y\}$ is not a substitution since it is not function. In fact, it does not exist such a function σ that verifies $D'\sigma = D$.

Definition 3.1.5. *The **most general unifier** (up to variable renaming) of a given conjunction of atoms and constraints D is the unifier σ, with $D_1\ \sigma = D$, such that if there exists another unifier σ', with $D_2\ \sigma' = D$ then D_1 subsumes D_2, where D_1 and D_2 are conjunctions of atoms and constraints verifying the equalities.*

Example 3.1.6. Example of the most general unifier of a given atom

The most general unifier of atom $D =$ path(1,1) is $\sigma = \{\text{X} \mapsto 1, \text{Y} \mapsto 1\}$ with (path(X,Y))$\sigma =$ path(1,1).

If we consider the unifier $\sigma' = \{\text{X} \mapsto 1\}$ with (path(X,X))$\sigma' =$ path(1,1) then the reader can check that path(X,Y) subsumes path(X,X).

Since it will be used in future sections, we give the notion of *the most specific generalisation*. Informally, the most specific generalisation (*msg*) of two terms is a term that retains the information common to both terms, introducing new variables in case of conflict. This notion can be extended to atoms and conjuctions of atoms. Further details about an algorithm that computes the most specific generalisation of a set of expression is given in [4]. The following examples illustrate its use.

Example 3.1.7. Examples of use of the most specific generalisation

Given the constants a and b and the variables X, Y, Z, U, V, W, Q, R and T.

1. msg(a,a) = a.
2. msg(a,b) = X.
3. msg(X,a) = X.
4. msg(X,X) = X.
5. msg(X,Y) = Z. Note that a new variable, Z, is introduced because neither term (X or Y) is more general than the other.
6. Given the atoms A_1 = $p(X,Y)$ and A_2 = $p(Z,a)$, msg(A_1,A_2) = p(U,V). We only compute the msg of atoms with the same predicate symbol.
7. Given A_1 = (p(X,Y),p(Y,Z)) and A_2 = (p(U,V),p(V,W)) two conjunctions of atoms, msg(A_1,A_2) = (p(Q,R),p(R,T)).

3.2 Derivation tree and tree dimension

Definition 3.2.1. ***Derivation step****. Given a program P and an atom B, a derivation step at atom B consists in selecting a clause c_n in P such that its head is unifiable with B. Then* unfold *w.r.t. atom B using clause c_n (see Definition 4.2.1). A derivation step can be performed if there exists such a clause c_n in the program whose head is unifiable with B, and if the resulting conjunction of literals and constraints is satisfiable (i.e., the unfolding is feasible). Otherwise, a* fail derivation *is produced.*

The result of performing a derivation step at an atom B is a *derivation word*. Each derivation step produces a new derivation word.

Definition 3.2.2. *A* ***labeled tree*** *$c(t_1, ..., t_k)$ ($k \geq 0$) is a tree such that c is the label of the root and $t_1, ..., t_k$ are labeled trees corresponding to the children of the root.*

Definition 3.2.3. *A* ***derivation tree of a CLP*** *is a labeled tree $c(t_1, ..., t_k)$ where every node represents a clause from the program and verifies the following properties:*

1. *Each node in the tree represents the clause in the program used to perform each derivation step. Thus, given an atom B, the root of the tree represents the clause in the program used to perform the first derivation step at B.*

2. *The root of the tree is denoted by label c.*

3. *Each child node is denoted by c_n, where c_n is the clause used to perform the corresponding derivation step.*

4. *Leaf nodes represent clauses with no atoms in their bodies (i.e., facts).*

Figure 3.1 shows a derivation tree of the CLP shown in Example 2.2.4. This tree shows in which clause the computation starts, in this case clause c_3. It also describes which clauses of the CLP are used in each derivation step from the root, in this case clauses c_2, c_3, c_1 and c_2, assuming a depth-first traversal of the derivation tree where the left nodes are visisted before the right nodes.

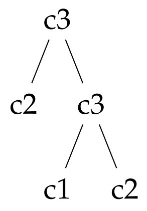


FIGURE 3.1: A derivation tree of the Fibonacci program in Example 2.2.4.

The reader can observe that the number of derivation trees from a given CLP can be infinite.

Definition 3.2.4. ***Dimension of a derivation tree.*** *Given a labeled tree t =c (t_1,...,t_k), the dimension of t, denoted by $dim(t)$, is defined as follows:*

$$dim(t) = \begin{cases} 0 & \textit{if } k = 0, \\ max_{i \in \{1,\ldots,k\}} dim(t_i) & \textit{if there is a unique maximum}, \\ max_{i \in \{1,\ldots,k\}} dim(t_i) + 1 & \textit{otherwise} \end{cases}$$

Note that this definition is not *circular* but *recursive*. A derivation tree is always finite and, therefore, the function dim visits recursively every subtree from the root until the base case ($k = 0$). This is a subtree with no child nodes, i.e., a leaf. A leaf has dimension 0. The rest of the dimension values of the subtrees are calculated according to the recursive definition of dim.

Derivation trees which are a list of nodes, i.e., each node of the tree has at most one child node, have dimension 0 and correspond to linear programs, while complete binary trees have dimension equal to its height and correspond to nonlinear programs. Figure 3.2 shows that the derivation tree of Figure 3.1 has dimension 1.

In section 3.3 we shall see that given a set of constrained Horn clauses P and any natural value k, P can be transformed into a new set of clauses $P^{[k]}$ whose derivation trees are a subset of P's derivation trees where we only consider those with dimension *at-most* k. This transformation of P into $P^{[k]}$ can be performed for any natural value k. Program $P^{[k]}$ is called *at-most-k-dimension program*.

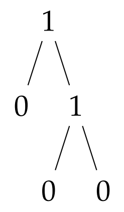


FIGURE 3.2: Dimension tree corresponding to the derivation tree in Figure 3.1

Each such set of clauses $P^{[k]}$ represents an *under-approximation* of the original set of clauses P. We use the term *under-approximation* in the sense that this program transformation gives as a result a new program where some sets of derivation trees have been possibly eliminated. Therefore, if we find a solution of P, then we also have a solution of $P^{[k]}$. However, finding a solution for $P^{[k]}$ does not mean that P has a solution.

We will present the set of rules that given a natural value k, transforms any CLP P into $P^{[k]}$. When the value of k is not important, any program generated using the rules defined in Definition 3.3.1 is called *dimension bounded program*.

We will also prove that any at-most-k-dimension program $P^{[k]}$ of that form is contained in the class of piecewise linear programs. Hence, program $P^{[k]}$ can be transformed into a linear program preserving the equivalence between them. The linearisation procedure developed in this thesis relies on the structure and properties of dimension bounded programs.

3.3 At-most-k-dimension programs $P^{[k]}$

Given a CLP P and $k \in \mathbb{N}$, we want to generate a new program $P^{[k]}$ whose derivation trees have dimension less or equal to k. For each predicate p ocurring in P , and for each d with $0 \leq d \leq k$, we generate new predicates $p^{(d)}$ and $p^{[d]}$ with the following properties. We shall see that $p^{(d)}$ *generates derivation trees* of dimension *exactly* d, i.e., when it occurs in the head of a clause in a derivation tree, it is the root of a subtree of dimension *exactly* d. Thus, we say that $p^{(d)}$ is predicate p of dimension *exactly* d. Similarly, we shall see that $p^{[d]}$ *generates derivation trees* of dimension *at most* d, i.e., when it occurs in the head of a clause in a derivation tree, it is the root of a subtree of dimension *at most* d. Thus, we say that $p^{[d]}$ is predicate p of dimension *at most* d. An atom with predicate $p^{[d]}$ or $p^{(d)}$ is denoted $H^{[d]}$ or $H^{(d)}$ respectively.

Definition 3.3.1. ***At-most-k-dimension program (or k-dimension-bounded program)*** $P^{[k]}$. *Given a CLP P and $k \in \mathbb{N}$, $P^{[k]}$ is a new set of constrained Horn clauses obtained by applying the following rules.*

Let H, B, B_i be atoms and let C be a constraint.

1. *Linear clauses*
   If $H \leftarrow C$ is in P, then $H^{(0)} \leftarrow C$ is in $P^{[k]}$.
   If $H \leftarrow C, B$ is in P, then $H^{(d)} \leftarrow C, B^{(d)}$ is in $P^{[k]}$ for $0 \leq d \leq k$.

2. *Non-linear clauses*
   If $H \leftarrow C, B_1, B_2, ..., B_n$ is in P, with $n > 1$, then :

(a) *For $1 \leq d \leq k$ and $1 \leq j \leq n$, set $Z_j = B_j^{(d)}$ and for $1 \leq i \leq n \wedge i \neq j$, set $Z_i = B_i^{[d-1]}$. Then: $H^{(d)} \leftarrow C, Z_1, ...Z_n$ is in $P^{[k]}$.*

(b) *For $1 \leq d \leq k$, and $J \subseteq \{1, ..., n\}$ with $|J| = 2$, set $Z_i = B_i^{(d-1)}$ if $i \in J$ and $Z_i = B_i^{[d-1]}$ if $i \in \{1, ..., n\} \setminus J$. Then: $H^{(d)} \leftarrow C, Z_1, ..., Z_n$ is in $P^{[k]}$.*

3. *e-clauses:*
$B^{[d]} \leftarrow B^{(e)}$ is in $P^{[k]}$, for $0 \leq d \leq k$, and every $0 \leq e \leq d$.

Example 3.3.2. At-most-1-dimension program of Fibonacci program

c1. fib$^{(0)}$(0,0).
c2. fib$^{(0)}$(1,1).
c3. fib$^{(1)}$(X,Y) $\leftarrow$ X > 1, X2 = X - 2, fib$^{[0]}$(X2, Y2), X1 = X - 1, fib$^{(1)}$(X1, Y1), Y = Y1 + Y2.
c4. fib$^{(1)}$(X,Y) $\leftarrow$ X > 1, X2 = X - 2, fib$^{(1)}$(X2, Y2), X1 = X - 1, fib$^{[0]}$(X1, Y1), Y = Y1 + Y2.
c5. fib$^{(1)}$(X,Y) $\leftarrow$ X > 1, X2 = X - 2, fib$^{(0)}$(X2, Y2), X1 = X - 1, fib$^{(0)}$(X1, Y1), Y = Y1 + Y2.
c6. false$^{(1)}$ $\leftarrow$ X > 5, fib$^{(1)}$(X,Y), X > Y.
c7. false$^{(0)}$ $\leftarrow$ X > 5, fib$^{(0)}$(X,Y), X > Y.
c8. false$^{[1]}$ $\leftarrow$ false$^{(1)}$.
c9. false$^{[1]}$ $\leftarrow$ false$^{(0)}$.
c10. false$^{[0]}$ $\leftarrow$ false$^{(0)}$.
c11. fib$^{[1]}$(X,Y) $\leftarrow$ fib$^{(1)}$(X,Y).
c12. fib$^{[1]}$(X,Y) $\leftarrow$ fib$^{(0)}$(X,Y).
c13. fib$^{[0]}$(X,Y) $\leftarrow$ fib$^{(0)}$(X,Y).
c14. false $\leftarrow$ false$^{[1]}$.

This example shows the at-most-1-dimension version of the CLP given in Example 2.2.4. The reader can check that any derivation tree with subgoal fib$^{(1)}$(X,Y) in its root has dimension exactly 1. Similarly, any derivation tree with subgoal fib$^{[1]}$(X,Y) in its root has dimension 1 or 0. In other words, it is not possible to generate a tree of dimension 2 or greater if the subgoal from which we start the derivation steps contains a predicate of dimension 1. Also note that now false$^{(1)}$, false$^{(0)}$, false$^{[1]}$ and false$^{[0]}$ are new predicates in the dimension bounded program. Clause c14. is not the result of applying any of the rules in Definition 3.3.1. We have added this clause in order to preserve the interpretation of predicates false$^{(1)}$, false$^{(0)}$, false$^{[1]}$ and false$^{[0]}$ as FALSE. Note that false in the head of clause c14. is *not* an atom but the logic symbol interpreted as FALSE. Finally, notice that the at-most-0-dimension version of the program is included in the at-most-1-dimension version of the program. In general, $P^{[k_1]} \subset P^{[k_2]}$ where $k_1 < k_2$.

We stated at the end of the previous section that $P^{[k]}$ represents an *under-approximation* of the original set of clauses P. We used the term *under-approximation* in the sense that this program transformation gives as a result a new program where some sets of derivation trees have been possibly eliminated. As a result, finding a solution for $P^{[k]}$ does not guarantee that P has a solution. The next example illustrates this.

Example 3.3.3. Recall the CLP P given in Example 2.2.11:

c1. triangular(X) $\leftarrow$ Y$_1 \geq 0$, $Y_2 \geq 0$, Y$_2$ = Y$_1$ + 1, X = (Y$_1$*Y$_2$)/2.
c2. square(X) $\leftarrow$ X$\geq$ 0.
c3. square(X) $\leftarrow$ Y$_1$ $\geq$ 0, Y_2 $\geq$ 0, Y$_2$ = Y$_1$+1, triangular(Y$_1$), triangular(Y$_2$), $X = Y_1 + Y_2$.

We build $P^{[0]}$ following Definition 3.3.1:

c1. triangular$^{(0)}$(X) $\leftarrow$ Y$_1 \geq 0$, $Y_2 \geq 0$, Y$_2$ = Y$_1$ + 1, X = (Y$_1$*Y$_2$)/2.
c2. square$^{(0)}$(X) $\leftarrow$ X$\geq$0.
c3. triangular$^{[0]}$(X) $\leftarrow$ triangular$^{(0)}$(X).
c4. square$^{[0]}$(X) $\leftarrow$ square$^{(0)}$(X).

In this case a solution for $P^{[0]}$ is {triangular$^{(0)}$(1), triangular$^{[0]}$(1), square$^{(0)}$(2), square$^{[0]}$(2)}. The reader may wonder why the ground atoms square$^{(0)}$(2) and square$^{[0]}$(2) belong to the solution if 2 is not the square of any natural number. The reason is that square$^{(0)}$(X) and square$^{[0]}$(X) are true in $P^{[0]}$ if X$\geq$ 0, without any other restriction on X.

This solution for $P^{[0]}$ yields to the following solution for P: {triangular(1), square(2)}. However, the reader can check that this is *not* a solution for P since 2 does not verify the property of being the sum of two consecutive triangular numbers.

As we stated in the previous section, every dimension bounded program is piecewise linear. Before proving this result, we present a lemma that will be useful in the proof.

Lemma 3.3.4. *Given a dimension bounded program $P^{[k]}$, all predicates (represented by nodes in the dependence graph) contained in the same SCC have the same dimension.*

Proof. The reader can observe in Definition 3.3.1 that a clause in $P^{[k]}$ with a predicate of dimension d in its head cannot contain a predicate of dimension greater than d in any of its subgoals, where $0 \leq d \leq k$. Therefore, the SCCs of the dependence graph of a given $P^{[k]}$ will always contain nodes representing predicates of the same dimension. □

Lemma 3.3.5. *If P is a CLP and $k \in \mathbb{N}$, then $P^{[k]}$ is piecewise linear.*

Proof. Recall the definition of *piecewise linear program* (Definition 2.3.9) given in the previous chapter. Now, we will see that, regarding the way we defined the rules in Definition 3.3.1 to build the at-most-k-dimension version of any CLP given, the resulting program $P^{[k]}$ verifies Definition 2.3.9.

- Linear clauses: Clauses in $P^{[k]}$ of the form 1. of Definition 3.3.1 verify Definition 2.3.9, as they only contain 0 or 1 subgoal.
- Non-linear clauses: Clauses in $P^{[k]}$ of the form 2a. and 2b. of Definition 3.3.1 verify Definition 2.3.9. The reason is that, considering lemma 3.3.4, every recursive subgoal, i.e., every subgoal whose predicate is contained in the same SCC as the predicate in the head, in this case $H^{(d)}$, has to have dimension d, with $1 \leq d \leq k$. In clauses of the form 2a., there is only 1 subgoal of dimension d, while in clauses of the form 2b., there are no subgoals of that form. Thus, clauses of the form 2a. and 2b. always verify Definition 2.3.9.

- e-clauses: Clauses in $P^{[k]}$ of the form 3. of Definition 3.3.1 verify Definition 2.3.9, as they only contain 1 subgoal.

$\square$

In the following sections, we will use *dimension bounded program* and *program* interchangeably.

Chapter 4

Linearisation Procedure

4.1 Preliminaries

The *ELP procedure* or *Program Linearisation Procedure* is an algorithm that automatically tranforms a dimension bounded set of Horn clauses into a linear program preserving the equivalence between them. The set of transformations performed by the procedure includes the *introduction of new predicates* and its corresponding *definition clauses*, and the *unfold/fold* transformation rules . As we shall see at the end of this chapter, this set of transformations preserves the meaning between the original and the transformed program (according to the definition of meaning preservation given in Definition 2.2.12).

The ELP procedure repeatedly applies the *CL procedure* or *Clause Linearisation Procedure*. The CL procedure replaces by linear clauses, non-linear clauses of a special kind, referred to as *minimally non-linear clauses* (see Definition 4.1.3). Minimally non-linear clauses are always guaranteed to exist in dimension bounded programs containing non-linear clauses.

Before giving a definition of minimally non-linear clause, we need to give the notion of *transitive closure*.

Definition 4.1.1. *The* ***transitive closure of a predicate*** p ***w.r.t. a program*** P *is the subset of clauses of* P *where each clause* c *satisfies (at least) one of the following conditions:*

1. *the predicate in the head of* c *is* p*.*
2. *the predicate in the head of* c *is a predicate whose node in the dependence graph of* P *is reachable from node* p*.*

The reader can observe that the transitive closure of a given predicate depends on the predicate p and also on the program P. However, for simplicity, we will denote the transitive closure of predicate p in P as $tc(p)$.

For a given P, if all clauses in $tc(p)$ are linear, then p has a *linear transitive closure*, otherwise p has a *non-linear transitive closure*.

The reader may notice that regarding the rules given in Definition 3.3.1, the transitive closure of predicates of dimension exactly 0 is always *linear*. As a result, so does the transitive closure of predicates of dimension at most 0.

Example 4.1.2. Transitive closure of predicate fib$^{(1)}$ w.r.t. program in Example 3.3.2

c1. fib$^{(0)}$(0,0).
c2. fib$^{(0)}$(1,1).
c3. fib$^{(1)}$(X,Y) $\leftarrow$ X > 1, X2 = X - 2, fib$^{[0]}$(X2, Y2), X1 = X - 1, fib$^{(1)}$(X1, Y1),

Y = Y1 + Y2.
c4. $fib^{(1)}$(X,Y) ← X > 1, X2 = X - 2, $fib^{(1)}$(X2, Y2), X1 = X - 1, $fib^{[0]}$(X1, Y1), Y = Y1 + Y2.
c5. $fib^{(1)}$(X,Y) ← X > 1, X2 = X - 2, $fib^{(0)}$(X2, Y2), X1 = X - 1, $fib^{(0)}$(X1, Y1), Y = Y1 + Y2.
c13. $fib^{[0]}$(X,Y) ← $fib^{(0)}$(X,Y).

Clauses c3., c4., c5. verify condition 1. in Definition 4.1.1 while clauses c13., c1. and c2. verify condition 2. As clauses c3., c4. and c5. are non-linear, the transitive closure of $fib^{(1)}$ is non-linear. The reader can check that, for the same reason, the transitive closure of $fib^{[1]}$ is non-linear as well.

Definition 4.1.3. ***Minimally non-linear clause.*** *A non-linear clause c in a program P is minimally non-linear if for every predicate p ocurring in a non-recursive subgoal in c, $tc(p)$ is linear.*

Example 4.1.4. At-most-2-dimension program of Fibonacci

c1. $fib^{(0)}$(0,0).
c2. $fib^{(0)}$(1,1).
c3. $fib^{(2)}$(X,Y) ← X > 1, X2 = X - 2, $fib^{[1]}$(X2, Y2), X1 = X - 1, $fib^{(2)}$(X1, Y1), Y = Y1 + Y2.
c4. $fib^{(2)}$(X,Y) ← X > 1, X2 = X - 2, $fib^{(2)}$(X2, Y2), X1 = X - 1, $fib^{[1]}$(X1, Y1), Y = Y1 + Y2.
c5. $fib^{(2)}$(X,Y) ← X > 1, X2 = X - 2, $fib^{(1)}$(X2, Y2), X1 = X - 1, $fib^{(1)}$(X1, Y1), Y = Y1 + Y2.
c6. $fib^{(1)}$(X,Y) ← X > 1, X2 = X - 2, $fib^{[0]}$(X2, Y2), X1 = X - 1, $fib^{(1)}$(X1, Y1), Y = Y1 + Y2.
c7. $fib^{(1)}$(X,Y) ← X > 1, X2 = X - 2, $fib^{(1)}$(X2, Y2), X1 = X - 1, $fib^{[0]}$(X1, Y1), Y = Y1 + Y2.
c8. $fib^{(1)}$(X,Y) ← X > 1, X2 = X - 2, $fib^{(0)}$(X2, Y2), X1 = X - 1, $fib^{(0)}$(X1, Y1), Y = Y1 + Y2.
c9. $false^{(2)}$ ← X > 5, $fib^{(2)}$(X,Y), X > Y.
c10. $false^{(1)}$ ← X > 5, $fib^{(1)}$(X,Y), X > Y.
c11. $false^{(0)}$ ← X > 5, $fib^{(0)}$(X,Y), X > Y.
c12. $false^{[2]}$ ← $false^{(2)}$.
c13. $false^{[2]}$ ← $false^{(1)}$.
c14. $false^{[2]}$ ← $false^{(0)}$
c15. $false^{[1]}$ ← $false^{(1)}$.
c16. $false^{[1]}$ ← $false^{(0)}$.
c17. $false^{[0]}$ ← $false^{(0)}$.
c18. $fib^{[2]}$(X,Y) ← $fib^{(2)}$(X,Y).
c19. $fib^{[2]}$(X,Y) ← $fib^{(1)}$(X,Y).
c20. $fib^{[2]}$(X,Y) ← $fib^{(0)}$(X,Y).
c21. $fib^{[1]}$(X,Y) ← $fib^{(1)}$(X,Y).
c22. $fib^{[1]}$(X,Y) ← $fib^{(0)}$(X,Y).
c23. $fib^{[0]}$(X,Y) ← $fib^{(0)}$(X,Y).
c24. false ← $false^{[2]}$.

The minimally non-linear clauses w.r.t. this program are c6., c7. and c8. They are minimally non-linear because they are non-linear and every predicate ocurring in a non-recursive subgoal has a linear transitive closure. In the case of c6. and c7., there is one predicate ocurring in a non-recursive subgoal, $\text{fib}^{[0]}$, and $\text{tc}(\text{fib}^{[0]})$ is linear. In the case of c8., there are two predicates ocurring in a non-recursive subgoal, $\text{fib}^{(0)}$ in both cases, and $\text{tc}(\text{fib}^{(0)})$ is linear. Clauses c3., c4., and c5. are non-linear but they are *not* minimally non-linear. The reason is that in c3. and c4. there is one predicate ocurring in a non-recursive subgoal, $\text{fib}^{[1]}$, but $\text{tc}(\text{fib}^{[1]})$ is non-linear. Finally, in the case of cluase c5., there are two predicates ocurring in a non-recursive subgoal, $\text{fib}^{(1)}$ in both cases, but $\text{tc}(\text{fib}^{(1)})$ is non-linear.

Basically, the ELP procedure, formally defined in section 4.2.2, selects in turns minimally non-linear clauses in the given program and replaces them by a set of linear clauses as given by the CL procedure, formally described in section 4.2.1. In other words, the CL procedure is a subroutine inside the ELP procedure that linearises a minimally non-linear clause c in each ELP iteration and gives as a result a set of linear clauses. This new set of clauses replaces c in the program.

Every dimension bounded program which is not linear is guaranteed to contain at least one minimally non-linear clause. This means that, in each iteration of the ELP procedure, either the non-linear set of clauses in the program is non-empty, in which case at least we can find a minimally non-linear clause to linearise in the set; or it is empty, in which case the program is already linear and the ELP execution stops. The following lemma formally describes this result.

Lemma 4.1.5. *Let P be a dimension bounded program and let N be the set of non-linear clauses in P. Then either N is empty or there is (at least) one minimally non-linear clause in N.*

Proof. We define a strict ordering relation $<$ over the set N, as follows: $c_1 < c_2$ iff c_1 belongs to the transitive closure of some predicate of a subgoal in clause c_2 other than a recursive subgoal (if exists). Let's prove that this ordering relation verifies the irreflexive, transitive and assymetric properties.

- not $c < c$ (Irreflexive)

  Let c be a non-linear clause in N. Let us assume $c < c$, i.e., c belongs to the transitive closure of some predicate of a subgoal in c other than a recursive subgoal. According to the definition of transitive closure (Definition 4.1.1), this means that either:

  1. there exists a nonrecursive subgoal in c whose predicate is the same as the predicate in the head of c or,
  2. the predicate in the head of c is a predicate whose node in the dependence graph of P is reachable from a predicate in a nonrecursive subgoal of c.

  1. is impossible since if such a subgoal exists, it would be a *recursive* subgoal and this contradicts the hypothesis $c < c$. Also 2. is impossible for the same reason: if the predicate in the head of c is a node in the dependence graph of P reachable from a predicate in a nonrecursive subgoal of c, then this subgoal would be *recursive*, which again is a contradiction with the hypothesis $c < c$. Therefore, the *irreflexive* property is satisfied.

- if $c_1 < c_2$ and $c_2 < c_3$ then $c_1 < c_3$ (Transitive)

  Let c_1, c_2 and c_3 be non-linear clauses in N. Let us assume $c_1 < c_2$ and $c_2 < c_3$. This means the following:

  - c_1 is in the transitive closure of some predicate ocurring in a nonrecursive subgoal in c_2, let's say p.
  - c_2 is in the transitive closure of some predicate ocurring in a nonrecursive subgoal in c_3, let's say q.

  As the transitive closure of q includes clause c_2, it also includes the transitive closure of predicates ocurring in each subgoal in c_2, in particular the transitive closure of p, $tc(p)$. As $tc(p)$ includes c_1, then we can conclude $c_1 < c_3$.

- if $c_1 < c_2$ then not $c_2 < c_1$ (Assymetric)

  Let c_1 and c_2 be non-linear clauses in N. Let us assume $c_1 < c_2$ and $c_2 < c_1$. Then, using the transitive property, we would have $c_1 < c_1$ which contradicts the irreflexive property, proved just above. Therefore, from the irreflexive and the transitive properties, we can prove the assymetric property.

The minimally non-linear clauses of P are the minimal elements of N under this ordering relation. □

4.2 Linearisation procedure applied to dimension bounded programs

Definition 4.2.1. ***Unfolding rule.*** *Given a dimension bounded program P and a clause c in P of the form $Q \leftarrow C, L, A, R$ where C is a constraint, Q and A are atoms, and L and R are (possibly empty) conjunctions of atoms. We consider the set of clauses $\{H_i \leftarrow C_i, B_i | i = 1, ..., n\}$ in P made out of the (renamed apart) clauses such that, for $i = 1, ..., n$, H_i is* unifiable *with A, via the most general unifier v_i and $(C, C_i)v_i$ is satisfiable. By unfolding c w.r.t. A, we derive the set of clauses $\{(Q \leftarrow C, C_i, L, B_i, R)v_i | i = 1, ..., n\}$.*

Example 4.2.2. Example of use of the unfolding rule

Given the following set of clauses:

c1. path(X,Y) ← X>0, Y>0, arc(X,Y).
c2. path(X,Y) ← double_arc(X,Y).
c3. path(X,Y) ← path(X,Z), path(Z,Y).

the result of unfolding clause c3 w.r.t. atom path(X,Z) are the following three clauses:

c4. path(X,Y) ← X>0, Z>0, arc(X,Z), path(Z,Y).
c5. path(X,Y) ← double_arc(X,Z), path(Z,Y).
c6. path(X,Y) ← path(X,W), path(W,Z), path(Z,Y).

Definition 4.2.3. ***Folding rule.*** *Given a dimension bounded program P and a clause c in P of the form $Q \leftarrow C, L, S, R$ where Q is an atom, C is a constraint, S is a non-empty conjunction of atoms, and L and R are (possibly empty) conjunctions of atoms; and a clause c' of the form $N \leftarrow D$, where N is an atom and D is a non-empty conjunction of* *atoms such that, for some substitution v: $(D)v = S$ then by folding c using c' we derive the clause $Q \leftarrow C, L, (D)v, R$.*

Example 4.2.4. Example of use of the folding rule

Given the following set of clauses:

c1. new(X,Z) ← path(X,Y), path(Y,Z).
c2. path(X,Y) ← path(X,Z), path(Z,Y).

the result of folding clause c2 using clause c1 is the following clause:

c3. path(X,Y) ← new(X,Y).

Definition 4.2.5. ***Introduction of new clauses****. Applying this rule we introduce a new clause (not ocurring before in the program) defining a new predicate. This clause is of the form* new*(X)← J where* new *is a new predicate symbol, X is the tuple of variables ocurring in J and J is a non-empty conjunction of atoms.*

These new clauses defining new predicates are called *Eureka Definitions*. The introduction of Eureka Definitions is a step in the linearisation procedure that we present in the following section.

Definition 4.2.6. *An* ***unfolding selection rule*** *is a rule that maps clauses to atoms. Given a clause, it returns a selected atom in its body.*

We will write U-rule to denote an unfolding selection rule.

Definition 4.2.7. ***Linear unfolding selection rule****. Given a clause c in a dimension bounded program P, a linear unfolding selection rule is an U-rule that returns a subgoal in c whose predicate has a linear transitive closure. This rule is undefined if there is not such a subgoal in c.*

We will write L-rule to denote a linear unfolding selection rule. If there are two subgoals in c verifying the previous condition, the L-rule selects nondeterministically one of them.

Definition 4.2.8. ***Linear lowest-dimension-first unfolding selection rule****. Given a clause c in a dimension bounded program P, a linear lowest-dimension-first unfolding selection rule is an L-rule that returns a subgoal in c whose predicate has the lowest dimension.*

We will write I-rule to denote a linear lowest-dimension-first unfolding selection rule. In the case of occurring two subgoals in c verifying the previous condition, the I-rule selects nondeterministically one of them.

Example 4.2.9. Example of linear lowest-dimension-first unfolding selection rule

Given the program in the Example 4.1.4 and given a linear lowest-dimension-first unfolding selection rule S:

1. S applied to clause c6. in the program returns the subgoal fib$^{[0]}$(X2, Y2) because it is the only subgoal in c6. whose predicate has a linear transitive closure.

2. S applied to c8. returns the subgoal fib$^{(0)}$(X2, Y2). Note that the predicates of both subgoals in c8. have a linear transitive closure and have dimension 0. Therefore, S selects nondeterministically one of them, in this case fib$^{(0)}$(X2, Y2).

3. S is not defined for clause c3., for instance, because neither fib$^{[1]}$(X2, Y2) nor fib$^{(2)}$(X1, Y1) has a linear transitive closure.

Lemma 4.2.10. *An I-rule is always defined for minimally non-linear clauses in bounded dimension programs.*

Proof. From definition 4.1.3, we have that every minimally non-linear clause in a bounded dimension program P has a subgoal B whose predicate's transitive closure w.r.t. P is linear. For more than one subgoal verifying the previous condition, it is possible to select the atom whose predicate has the lowest dimension (in case of having more than one subgoal verifying the two previous conditions, the I-rule will select nondeterministically one of them). □

Definition 4.2.11. ***Unfolding tree***. *Let P be a dimension bounded program, c a clause in P and S an I-rule. An* ***unfolding tree*** *T w.r.t. c in P via S is a labeled tree with clauses, constructed as follows:*

- *c is the root label of T.*
- *If M is a node labeled by a clause c_m and B is the atom selected by S in c_m then for each clause c_n that results from unfolding c_m w.r.t. atom B, there is a child node N of M labeled by c_n.*

We will write I-tree to denote an unfolding tree w.r.t. to an I-rule.

Example 4.2.12. Example of unfolding tree

Given the program in the Example 4.1.4, we build the unfolding tree w.r.t. c6. vía an I-rule S (as the one given in the Example 4.2.9):

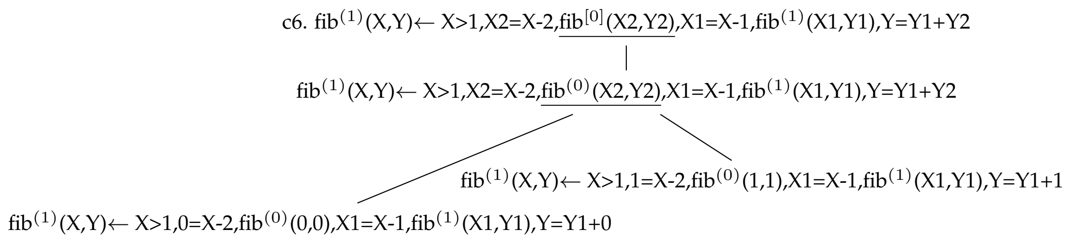

c6. fib$^{(1)}$(X,Y)← X>1,X2=X-2,$\underline{\text{fib}^{[0]}\text{(X2,Y2)}}$,X1=X-1,fib$^{(1)}$(X1,Y1),Y=Y1+Y2
fib$^{(1)}$(X,Y)← X>1,X2=X-2,$\underline{\text{fib}^{(0)}\text{(X2,Y2)}}$,X1=X-1,fib$^{(1)}$(X1,Y1),Y=Y1+Y2
fib$^{(1)}$(X,Y)← X>1,1=X-2,fib$^{(0)}$(1,1),X1=X-1,fib$^{(1)}$(X1,Y1),Y=Y1+1
fib$^{(1)}$(X,Y)← X>1,0=X-2,fib$^{(0)}$(0,0),X1=X-1,fib$^{(1)}$(X1,Y1),Y=Y1+0

FIGURE 4.1: Unfolding tree w.r.t. c6. vía S

We have underlined the atoms selected by S at each parent node.

Definition 4.2.13. ***Upper portion of a tree***. *A non-empty tree T' is called an* ***upper-portion*** *of a tree T if it verifies both conditions:*

1. *The root node of T' is also the root node of T.*
2. *For every node N of T', N is also a node of T and either N is a leaf node of T' or all child nodes of N in T are also child nodes of N in T'.*

An upper portion of a tree T is *trivial* if it only consists of a single node. In any other case, we say it is *non-trivial*.

Example 4.2.14. Example of upper portion of a tree

T is a tree where each label N_i, with $i \in \mathbb{N}$, represents a node.

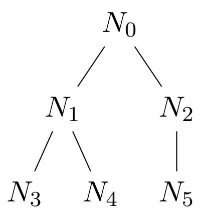
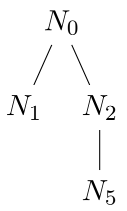
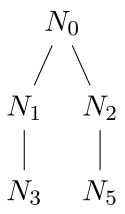


FIGURE 4.2: Tree T



(A) This is an upper portion of T



(B) This is *not* an upper portion of T since it does not verify condition 2 in Definition 4.2.13

The following lemma implies that when we unfold a minimally non-linear clause in a bounded dimension program via an I-rule S, then S is also defined for all the non-linear clauses that result from this unfolding as these clauses are also minimally non-linear.

Lemma 4.2.15. *Let c be a minimally non-linear clause in a bounded dimension program P, S an I-rule and T an I-tree w.r.t. clause c in P via S. Then every non-linear clause in the set of leaves L of any finite upper-portion of T is minimally non-linear in $(P \backslash \{c\}) \cup L$.*

Proof. We will give a proof by contradiction. Let p be the predicate of the atom selected by S in the body of c. Now suppose that a clause $c_n \in L$ is not minimally non-linear. From definition 4.1.3, it follows that there is a non-recursive subgoal in c_n whose predicate's transitive closure is non-linear. Let q be the predicate of this non-recursive goal in c_n. The clauses in the transitive closure of q are also in the transitive closure of the predicate p. This means that the transitive closure of p is non-linear which is a contradiction since c is minimally non-linear and S is an I-rule. □

The following definitions introduce two kinds of upper-portion of I-trees that will be built by the procedure in order to decide when to stop the unfolding, which Eureka Definitions to introduce, and when to start the folding.

Definition 4.2.16. *Let P be a dimension bounded program, c a clause in P, S an I-rule, and T an I-tree w.r.t. c and P via S. A finite upper-portion U of T is said to be* ***F-linearisable w.r.t. a set of Eureka Definitions ED*** *if each leaf of U*

- *either can be folded using as folding clause a definition in the set ED and giving as a result a* linear *clause,*
- *or is a* linear *clause.*

Definition 4.2.17. *Let P be a dimension bounded program, c a clause in P, S an I-rule, and T an I-tree w.r.t. c and P via S. A finite upper-portion U of T is said to be* ***E-linearisable*** *if each leaf of U is*

- *either a* linear *clause,*
- *or an* Eurekable clause*. An* eurekable clause *is a clause c_n in a node of a I-tree T w.r.t. c in P via S such that, there is an ancestor c_m of c_n in T and a tuple J of atoms such that the tuples of all the subgoals in c_n and c_m are instances of J.*

We will say that U is *minimal* F-linearisable iff there exists no F-linearisable upper portion U' of T, with $U' \neq U$, that is also an upper portion of U. Likewise, U is *minimal* E-linearisable iff there exists no E-linearisable upper portion U' of T, with $U' \neq U$, that is also an upper portion of U.

The detection of an Eurekable clause in an E-linearisable upper portion tell us when to stop unfolding in the corresponding branch of the I-tree and introduce a new Eureka Definition. The body of the Eureka Definition consists of the tuple J. Further details of the way Eureka Definitions are constructed is given in Algorithm 1.

Example 4.2.18. Example of a F-linearisable upper portion of an I-tree w.r.t. a set of Eureka Definitions
In section 4.2.3, we present an example of the ELP procedure applied to a dimension bounded program. Figures 4.6, 4.7, 4.9, 4.10, 4.12 and 4.13 are examples of F-linearisable upper portions of an I-tree w.r.t. a given set of Eureka Definitions.

Example 4.2.19. Example of a E-linearisable upper portion of an I-tree w.r.t. a clause in a program
In the example of section 4.2.3, Figures 4.4, 4.5, 4.8 and 4.11 are examples of E-linearisable upper portions of an I-tree w.r.t. a clause in the given program.

The following lemma implies that given a minimally non-linear clause in a bounded dimension program P, an I-rule S and an I-tree T w.r.t. c in P via S, it is always possible to build a *finite* E-linearisable upper portion of T.

Lemma 4.2.20. *Let P be a bounded dimension program, c a minimally non-linear clause in P, S an I-rule, T an I-tree w.r.t. P and c via S. Then there exists at least an E-linearisable upper portion of T.*

Proof. The main idea of the proof is to show that the definition given in 4.2.17 of an E-linearisable upper portion of T w.r.t. P and c via S leads to a finite structure in a finite number of steps. The I-rule S always selects an atom whose transitive closure is linear. Thus, the number of subgoals in the clauses that result from unfolding clause c (any descendant of c in T) is less than or equal to the number of subgoals in c. As the number of predicates in P is finite (P is finite), it is evident that in a finite number of unfolding steps from the root of T we can find either a linear clause or a clause c_n that is eurekable. □

4.2.1 Clause Linearisation Procedure

The Clause Linearisation Procedure or CL procedure is an algorithm that, given a bounded dimension program P, a minimally non-linear clause c in P and an I-rule S, replaces c by a set of linear clauses LC in P, so that the equivalence between program P and program $(P \setminus \{c\}) \cup LC$ is preserved (recall Definition 2.2.12).

Algorithm 1 Clause Linearisation Procedure (CL procedure)

```
1: procedure CLPROCEDURE($P$, $c$, $S$)
   Input: A bounded dimension program $P$, a minimally non-linear clause $c$ in $P$ and
   an I-rule $S$.
   Output: A set of linear clauses $LC$ and a set of Eureka Definitions $ED$.
2: Construct the minimal E-linearisable upper portion $U$ of a I-tree $T$ w.r.t. $P$ and $c$ via
   $S$.
3: For every leaf $c_n$ in $U$ which is eurekable via an ancestor $c_m$ introduce a fresh predi-
   cate symbol $new^{(d)}$ and construct a clause:
   $e$: new$^{(d)}(X_1,...X_n) \leftarrow J$ where
      • $J$ is a conjunction of atoms such that both the conjunction $J_n$ of all subgoals in
        clause $c_n$ and the conjunction $J_m$ of all subgoals in clause $c_m$ are instances of
        $J$. We will use as $J$ the most specific generalisation (recall the examples given in
        Example 3.1.7) of $J_n$ and $J_m$.
      • $\{X_1, ..., X_n\}$ is the minimal subset of the set of all variables in $J$ such that both
        clauses $c_n$ and $c_m$ can be folded using $e$.
      • The natural number $d$ must be chosen accordingly to the rules specified in Def-
        inition 3.3.1.
4: Let $ED$ be the set consisting of all clauses of the form $e$ constructed as above after
   having eliminated "copies", differing from other clauses in the set only in the names
   of the predicates they define, and in the order of the variables in the heads.
5: Select the minimal F-linearisable upper portion $U'$ of $U$ w.r.t. $ED$.
6: for each clause $e$ in $ED$:
7:     Construct the minimal non-trivial F-linearisable upper portion $U_e$ w.r.t. $ED$ of an
   I-tree w.r.t. $e$ in $P$ via $S$.
8: Let $LC$ be the set of all linear clauses in the leaves of $U'$ and $U_e$ together with the set
   of all clauses that result from the folding of the non-linear clauses in the leaves of $U'$
   and $U_e$ using the clauses in $ED$.
```

Note that U' in step 5 can be a trivial upper portion of U while U_e in step 7 is necessarily non-trivial, i.e., it is not allowed to fold the corresponding e using e itself. This is called *self-folding* and it does not preserve the meaning of programs.

The following lemma implies that an I-rule is always defined for all the clauses in ED and, consequently, for all the clauses in U_e, for all $e \in ED$. This guarantees that step 7 is always feasible.

Lemma 4.2.21. *Let P be a dimension bounded program, c a clause in P, S an I-rule, and (LC, ED) be the output of the CL procedure applied to input (P, c, S). Then every clause in ED is minimally non-linear in $P \cup ED$.*

Proof. From Lemma 4.2.15 and the way we have constructed Eureka Definitions in step 3, it follows that if c is minimally non-linear in P then so does every clause in the set ED. □

4.2.2 Program Linearisation Procedure

Finally, we describe an algorithm that linearises bounded dimension programs, namely the Program Linearisation Procedure or ELP procedure. This algorithm repeatedly applies the CL procedure replacing a minimally non-linear clause by the set of linear clauses that the CL procedure generates for that clause.

Algorithm 2 Program Linearisation Procedure (ELP procedure)

```
1: procedure ELPROCEDURE($P$, $S$)
   Input: A bounded dimension program $P$ and an I-rule $S$
   Output: A set of linear clauses $LC$ and a set of Eureka Definitions $ED$
2: Let $NLC$ be the set of all non-linear clauses in $P$
3: $i \leftarrow 0$
4: $P_i \leftarrow P$
5: $dim \leftarrow 1$
6: while $NLC$ is non-empty do:
7:     Select a minimally non-linear clause $C$ from $NLC$ whose predicate in the head
   has dimension $dim$
8:     $(LC_i, ED_i) \leftarrow$ CLPROCEDURE$(P_i, c, S)$
9:     $P_{i+1} \leftarrow (P_i \setminus \{c\}) \cup LC_i$
10:    $NL \leftarrow NL \setminus \{c\}$
11:    $i \leftarrow i + 1$
12:    if for each clause in $NLC$, the predicate in its head has dimension $> dim$, then:
13:    $dim \leftarrow dim + 1$
14: $ED \leftarrow \cup_i ED_i$
15: $LC \leftarrow \cup_i LC_i$
```

In each iteration, the algorithm selects a minimally non-linear clause c in the program to replace it by a set of linear clauses. The order in which each clause c is selected depends on the dimension of the predicate in its head. For short, we will write that a clause has dimension d iff the dimension of the predicate in its head is d. Thus, first are chosen those minimally non-linear clauses of dimension 1. Once every minimally non-linear clause of dimension 1 has been linearised, the algorithm selects those of dimension 2, and so on. This order of selection is managed by means of the variable dim whose value is updated to $d+1$ each time that all the non-linear clauses of dimension d have been linearised, with $d > 0$. This guarantees that if the algorithm is linearising a clause of dimension $d + 1$ then all clauses of dimension d are already linear.

4.2.3 Example

Let P be the following dimension bounded program:

c1. arc$^{(0)}$(0,0)$\leftarrow$ true.
c2. path$^{(0)}$(X,Y)$\leftarrow$ Z=Y, path$^{(0)}$(X,Z).
c3. path$^{(1)}$(X,Y)$\leftarrow$ arc$^{(0)}$(X,Z), arc$^{(0)}$(Z,Y).
c4. path$^{(1)}$(X,Y)$\leftarrow$ Z=Y, path$^{(1)}$(X,Z).
c5. double$^{(1)}$(X,Y) $\leftarrow$ X>0, Y>0, path$^{(0)}$(X,Z), path$^{(0)}$(Z,Y).
c6. double$^{(1)}$(X,Y) $\leftarrow$ X>0, Y>0, path$^{[0]}$(X,Z), path$^{(1)}$(Z,Y).
c7. double$^{(1)}$(X,Y) $\leftarrow$ X>0, Y>0, path$^{(1)}$(X,Z), path$^{[0]}$(Z,Y).

The set of minimally non-linear clauses in the first iteration of ELP is $\{c3, c5\}$. Both clauses have dimension 1 so ELP chooses arbitrarily clause $c3$ to be linearised.

c3. path$^{(1)}$(X,Y)← arc$^{(0)}$(X,Z), arc$^{(0)}$(Z,Y)

|

c8. path$^{(1)}$(0,Y)← true, arc$^{(0)}$(0,Y)

FIGURE 4.4: E-linearisable upper portion U of a I-tree w.r.t. P and clause $c3$

The result of performing step 2 of Algorithm 1 is shown in Figure 4.4. Since all the leaves of U are linear, the CL procedure for iteration $i = 0$ returns $LC_0 = \{c8\}$ and $ED_0 = \{\emptyset\}$. P_1 in the next iteration of ELP is:

c1. arc$^{(0)}$(0,0)← true.
c2. path$^{(0)}$(X,Y)← Z=Y, path$^{(0)}$(X,Z).
c8. path$^{(1)}$(0,Y)← true, arc$^{(0)}$(0,Y).
c4. path$^{(1)}$(X,Y)← Z=Y, path$^{(1)}$(X,Z).
c5. double$^{(1)}$(X,Y) ← X>0, Y>0, path$^{(0)}$(X,Z), path$^{(0)}$(Z,Y).
c6. double$^{(1)}$(X,Y) ← X>0, Y>0, path$^{[0]}$(X,Z), path$^{(1)}$(Z,Y).
c7. double$^{(1)}$(X,Y) ← X>0, Y>0, path$^{(1)}$(X,Z), path$^{[0]}$(Z,Y).

Now path$^{(1)}$ is a predicate with a linear transitive closure. Therefore, the set of minimally non-linear clauses is $\{c5, c6, c7\}$. The next clause to be linearised is $c5$.

c5. double$^{(1)}$(X,Y)← X>0,Y>0,$\underline{\text{path}^{(0)}\text{(X,Z), path}^{(0)}\text{(Z,Y)}}$

|

e1. double$^{(1)}$(X,Y)← X>0,Y>0,Z1=Z, $\underline{\text{path}^{(0)}\text{(X,Z1), path}^{(0)}\text{(Z,Y)}}$

FIGURE 4.5: E-linearisable upper portion U of a I-tree w.r.t. P and clause $c5$

The result of performing step 2 of Algorithm 1 is shown in Figure 4.5. Clause $e1$ is eurekable via clause $c5$. In this case J =path$^{(0)}$(X,Y), path$^{(0)}$(Z,Z1) such that J_{c5} =path$^{(0)}$(X,Z), path$^{(0)}$(Z,Y) (underlined) and J_{e1} =path$^{(0)}$(X,Z1), path$^{(0)}$(Z,Y) (underlined) are instances of J. Therefore, we introduce the Eureka Definition following the instructions given in step 3 of the CL procedure:

d1. new1$^{(1)}$(X,Y,Z,Z1)← path$^{(0)}$(X,Y), path$^{(0)}$(Z,Z1).

Now, we select the minimal F-linearisable upper portion U' of U (Figure 4.5) w.r.t. $ED = \{d1\}$. This means that we fold clause e1 using Eureka Definition d1, and give as a result linear clause c9:

c5. double$^{(1)}$(X,Y)← X>0,Y>0,path$^{(0)}$(X,Z), path$^{(0)}$(Z,Y)

|

c9. double$^{(1)}$(X,Y)← X>0, Y>0, Z1=Z, new1$^{(1)}$(X,Z1,Z,Y)

FIGURE 4.6: F-linearisable upper portion U' of U w.r.t. ED

Now we proceed to construct the minimal F-linearisable upper portion U_{d1} w.r.t. ED. This means that we unfold clause d1 at path$^{(0)}$(X,Y), and fold the resulting clause using Eureka Definition d1. The result is linear clause c10.

d1. new1$^{(1)}$(X,Y,Z,Z1)← path$^{(0)}$(X,Y), path$^{(0)}$(Z,Z1)

|

c10. new1$^{(1)}$(X,Y,Z,Z1)← Z2=Y, new1$^{(1)}$(X,Z2,Z,Z1)

FIGURE 4.7: F-linearisable upper portion U_{d1} w.r.t. ED

The CL procedure for iteration $i = 1$ returns $LC_1 = \{c9, c10\}$ and $ED_1 = \{d1\}$. P_2 in the next iteration of ELP is:

c1. arc$^{(0)}$(0,0)← true.
c2. path$^{(0)}$(X,Y)← Z=Y, path$^{(0)}$(X,Z).
c8. path$^{(1)}$(0,Y)← true, arc$^{(0)}$(0,Y).
c4. path$^{(1)}$(X,Y)← Z=Y, path$^{(1)}$(X,Z).
c9. double$^{(1)}$(X,Y)← X>0, Y>0, Z1=Z, new1$^{(1)}$(X,Z1,Z,Y).
c10. new1$^{(1)}$(X,Y,Z,Z1)← Z2=Y, new1$^{(1)}$(X,Z2,Z,Z1).
c6. double$^{(1)}$(X,Y) ← X>0, Y>0, path$^{[0]}$(X,Z), path$^{(1)}$(Z,Y).
c7. double$^{(1)}$(X,Y) ← X>0, Y>0, path$^{(1)}$(X,Z), path$^{[0]}$(Z,Y).

Now the set of minimally non-linear clauses is $\{c6, c7\}$. The next clause to be linearised is $c6$.

c6. double$^{(1)}$(X,Y)← X>0,Y>0,path$^{[0]}$(X,Z), path$^{(1)}$(Z,Y)

|

a1. double$^{(1)}$(X,Y)← X>0,Y>0, $\underline{\text{path}^{(0)}\text{(X,Z), path}^{(1)}\text{(Z,Y)}}$

|

e2. double$^{(1)}$(X,Y)← X>0,Y>0,Z1=Z, $\underline{\text{path}^{(0)}\text{(X,Z1), path}^{(1)}\text{(Z,Y)}}$

FIGURE 4.8: E-linearisable upper portion U of a I-tree w.r.t. P and clause $c6$

The result of performing step 2 of Algorithm 1 is shown in Figure 4.8. As clause $e2$ is eurekable via clause a1 (in this case J =path$^{(1)}$(X,Y), path$^{(0)}$(Z,Z1) such that J_{a1} =path$^{(0)}$(X,Z), path$^{(1)}$(Z,Y) and J_{e2} =path$^{(0)}$(X,Z1), path$^{(1)}$(Z,Y) are instances of J). Therefore, we introduce the Eureka Definition following the instructions given in step 3 of the CL procedure:

d2. new2$^{(1)}$(X,Y,Z,Z1)← path$^{(0)}$(X,Y), path$^{(1)}$(Z,Z1).

Now we select the minimal F-linearisable upper portion U' of U (Figure 4.8) w.r.t. $ED = \{d1, d2\}$. This means that we fold clause e2 using Eureka Definition d2, and give as a result linear clause c11.

c6. double$^{(1)}$(X,Y)$\leftarrow$ X>0,Y>0, path$^{[0]}$(X,Z), path$^{(1)}$(Z,Y)

|

double$^{(1)}$(X,Y)$\leftarrow$ X>0,Y>0, path$^{(0)}$(X,Z), path$^{(1)}$(Z,Y)

|

c11. double$^{(1)}$(X,Y)$\leftarrow$ X>0,Y>0,Z1=Z, new2$^{(1)}$(X,Z1,Z,Y)

FIGURE 4.9: F-linearisable upper portion U' of U w.r.t. ED

Then we proceed to construct the minimal F-linearisable upper portion U_{d2} w.r.t. ED. This means that we unfold clause d2 at path$^{(0)}$(X,Y), and fold the resulting clause using Eureka Definition d2. The result is linear clause c12.

d2. new2$^{(1)}$(X,Y,Z,Z1)$\leftarrow$ path$^{(0)}$(X,Y), path$^{(1)}$(Z,Z1)

|

c12. new2$^{(1)}$(X,Y,Z,Z1)$\leftarrow$ Z2=Y, new2$^{(1)}$(X,Z2,Z,Z1)

FIGURE 4.10: F-linearisable upper portion U_{d2} w.r.t. ED

The CL procedure for iteration $i = 2$ returns $LC_2 = \{c11, c12\}$ and $ED_2 = \{d2\}$. P_2 in the next iteration of ELP is:

c1. arc$^{(0)}$(0,0)$\leftarrow$ true.
c2. path$^{(0)}$(X,Y)$\leftarrow$ Z=Y, path$^{(0)}$(X,Z).
c8. path$^{(1)}$(0,Y)$\leftarrow$ true, arc$^{(0)}$(0,Y).
c4. path$^{(1)}$(X,Y)$\leftarrow$ Z=Y, path$^{(1)}$(X,Z).
c9. double$^{(1)}$(X,Y)$\leftarrow$ X>0, Y>0, Z1=Z, new1$^{(1)}$(X,Z1,Z,Y).
c10. new1$^{(1)}$(X,Y,Z,Z1)$\leftarrow$ Z2=Y, new1$^{(1)}$(X,Z2,Z,Z1).
c11. double$^{(1)}$(X,Y)$\leftarrow$ X>0, Y>0, Z1=Z, new2$^{(1)}$(X,Z1,Z,Y).
c12. new2$^{(1)}$(X,Y,Z,Z1)$\leftarrow$ Z2=Y, new2$^{(1)}$(X,Z2,Z,Z1).
c7. double$^{(1)}$(X,Y) $\leftarrow$ X>0, Y>0, path$^{(1)}$(X,Z), path$^{[0]}$(Z,Y).

Finally, the set of minimally non-linear clauses is $\{c7\}$, thus $c7$ is the last clause to be linearised in P.

c7. double$^{(1)}$(X,Y)$\leftarrow$ X>0,Y>0,path$^{(1)}$(X,Z), path$^{[0]}$(Z,Y)

|

a2. double$^{(1)}$(X,Y)$\leftarrow$ X>0,Y>0, $\underline{\text{path}^{(1)}\text{(X,Z), path}^{(0)}\text{(Z,Y)}}$

|

e3. double$^{(1)}$(X,Y)$\leftarrow$ X>0,Y>0, $\underline{\text{path}^{(1)}\text{(X,Z)}}$,Z1=Y, $\underline{\text{path}^{(0)}\text{(Z,Z1)}}$

FIGURE 4.11: E-linearisable upper portion U of a I-tree w.r.t. P and clause $c7$

The result of performing step 2 of Algorithm 1 is shown in Figure 4.11. As clause $e3$ is eurekable via clause a2 (in this case J =path$^{(1)}$(X,Y), path$^{(0)}$(Y,Z) such that J_{a2} =path$^{(1)}$(X,Z), path$^{(0)}$(Z,Y) and J_{e3} =path$^{(1)}$(X,Z), path$^{(0)}$(Z,Z1) are instances of J). Therefore, we introduce the Eureka Definition following the instructions given in step 3 of the CL procedure:

d3. new3$^{(1)}$(X,Y)$\leftarrow$ path$^{(1)}$(X,Z), path$^{(0)}$(Z,Y).

Now, we select the minimal F-linearisable upper portion U' of U (Figure 4.11) w.r.t. $ED = \{d1, d2, d3\}$. This means that we fold clause e3 using Eureka Definition d3, and give as a result linear clause c13:

c7. double$^{(1)}$(X,Y)← X>0,Y>0,path$^{(1)}$(X,Z), path$^{[0]}$(Z,Y)

|

double$^{(1)}$(X,Y)← X>0,Y>0,path$^{(1)}$(X,Z), path$^{(0)}$(Z,Y)

|

c13. double$^{(1)}$(X,Y)← X>0,Y>0,Z1=Y, new3$^{(1)}$(X,Z1)

FIGURE 4.12: F-linearisable upper portion U' of U w.r.t. ED

Now we proceed to construct the minimal F-linearisable upper portion U_{d3} w.r.t. ED. This means that we unfold clause d3 at path$^{(0)}$(Z,Y), and we fold the resulting clause using Eureka Definition d3. The result is linear clause c14.

d3. new3$^{(1)}$(X,Y)← path$^{(1)}$(X,Z), path$^{(0)}$(Z,Y)

|

c14. new3$^{(1)}$(X,Y)← Z1=Y, new3$^{(1)}$(X,Z1)

FIGURE 4.13: F-linearisable upper portion U_{d3} w.r.t. ED

The CL procedure for iteration $i = 3$ returns $LC_3 = \{c13, c14\}$ and $ED_3 = \{d3\}$. As the set of minimally non-linear clauses in P_3 is empty, ELP terminates and returns the set $ED = \{d1, d2, d3\}$ and the set $LC = \{c8, c9, c10, c11, c12, c13, c14\}$. The resulting linearised program is:

c1. arc$^{(0)}$(0,0)← true.
c2. path$^{(0)}$(X,Y)← Z=Y, path$^{(0)}$(X,Z).
c8. path$^{(1)}$(0,Y)← true, arc$^{(0)}$(0,Y).
c4. path$^{(1)}$(X,Y)← Z=Y, path$^{(1)}$(X,Z).
c9. double$^{(1)}$(X,Y)← X>0,Y>0, Z1=Z, new1$^{(1)}$(X,Z1,Z,Y).
c10. new1$^{(1)}$(X,Y,Z,Z1)← Z2=Y, new1$^{(1)}$(X,Z2,Z,Z1).
c11. double$^{(1)}$(X,Y)← X>0,Y>0, Z1=Z, new2$^{(1)}$(X,Z1,Z,Y).
c12. new2$^{(1)}$(X,Y,Z,Z1)← Z2=Y, new2$^{(1)}$(X,Z2,Z,Z1).
c13. double$^{(1)}$(X,Y)← X>0,Y>0, Z1=Y, new3$^{(1)}$(X,Z1).
c14. new3$^{(1)}$(X,Y)← Z1=Y, new3$^{(1)}$(X,Z1).

4.2.4 Correctness of ELP

Now we will give two theorems that imply the correctness of the Program Linearisation Procedure. First, we prove the CL procedure correctness. Second, relying on the CL procedure correctness, we prove that the ELP procedure is also correct.

Theorem 4.2.22. *Let P be a dimension bounded program, c a clause in P, S an I-rule, LC the set of linear clauses returned by the CL procedure applied to input (P, c, S), and $pred(P)$ the set of predicates occurring in P. Then*

- *the CL procedure applied to (P, c, S)* terminates,
- *$M(P) = M_{pred(P)}((P \setminus \{c\}) \cup LC)$.*

Proof. The proof will be divided into two parts. First, we will prove termination of the CL procedure and second, we will prove the equivalence (recall Definition 2.2.12) between the original program and the transformed program after linearising one individual clause by means of the CL procedure.

1. **Termination:**

   We will prove that the upper portion of the three unfolding trees constructed in the Algorithm 1 can be constructed in a finite number of steps.

   (a) **E-linearisable upper portion U of a I-tree w.r.t. P and c via S, constructed in step 2 of the procedure.**
   Relying on Lemma 4.2.20, the construction of U can be performed in a finite number of steps.

   (b) **F-linearisable upper portion U' of U w.r.t. ED, constructed in step 5 of the procedure**.
   Relying on the construction of Eureka Definitions described in step 3 of the procedure, it is easy to see that the construction of U' consists in folding each leaf of U using the proper E_i in ED and this can be performed in a finite number of steps (U was a finite structure).

   (c) **F-linearisable upper portion of U_{E_i} of a I-tree w.r.t. P and c via S, for each E_i in ED, constructed in step 7 of the procedure**.
   Let's assume that the I-rule S used in step 2 is the same as the I-rule used to build U_{E_i}. Then, as S is uniquely determined by the set of subgoals appearing in a clause, U_{E_i} will be constructed in the same way as the I-tree for the clause which led to the introduction of E_i. Let's assume that U is the E-linearisable upper portion of a I-tree such that L is the clause in a leaf of U for which E_i was introduced. Let A be the ancestor clause that makes L eurekable. Relying on step 3 of the CL procedure, clause E_i has the same subgoals as clause L (and A). Therefore, we can put in one-to-one correspondence the nodes in U_{E_i} with the nodes of the subtree of U whose root is A. The clause in a node of U_{E_i} has the same subgoals as the clause in the corresponding node of U and both clauses only (possibly) differ in the set of constraints. Thus, U_{E_i} will be constructed in a finite number of steps.

2. **Equivalence:**

   Our aim is to prove that $M(P) = M_{pred(P)}((P \setminus \{c\}) \cup LC)$, where $pred(P)$ is the set of predicates ocurring in P.

   Let's assume that, initially, our program P consists of every clause in P together with the set of definitions ED that will be introduced during the ELP procedure. Thus, $M_{pred(P)}(P \cup ED) = M(P)$. Relying on the correctness of the transformation system described by Definitions 4.2.1 and 4.2.3, we have that $M_{pred(P)}(P \cup ED) = M_{pred(P)}((P \setminus \{c\}) \cup LC)$. Therefore, $M(P) = M_{pred(P)}((P \setminus \{c\}) \cup LC)$.

□

Theorem 4.2.23. *Let P be a dimension bounded program, S an I-rule, LC the set of all linear clauses returned by the ELP procedure applied to input (P, S), NLC the set of all non-linear clauses in P and $pred(P)$ the set of predicates occurring in P. Then*

- *the ELP procedure applied to (P, S)* terminates,
- $M(P) = M_{pred(P)}((P \setminus \{NLC\}) \cup LC)$.

Proof. Again, we will prove first termination, and second, the equivalence between the original program and the transformed program by means of the ELP procedure.

1. **Termination:**
   The ELP procedure always terminates for the following reasons:
   (a) Given P, the set NLC is finite.
   (b) In each iteration of ELP, one clause in NLC is transformed into linear by means of the CL procedure.
   (c) The CL procedure terminates in a finite number of steps.
2. **Equivalence:**
   The correctness of the CL procedure leads to the correctness of the ELP procedure.

$\square$

Chapter 5

Conclusions

A majority of solvers handle non-linear Horn clauses but there are notable exceptions like VeriMAP ([2]) or Sally[1]. With this work, we conclude that determining the unsatisifiability of a given non-linear program P, i.e., the lack of solution for P, can be reduced to determine the unsatisfiability of a linear program, which is obtained by means of a set of transformations applied to P. The resulting linear program can be computed using a linear solver. On the other hand, determining the satisfiability of a given non-linear program P by solving the corresponding linear program is not always guaranteed.

In [1] it is proved that piecewise linear programs are linearisable, i.e. there exists an algorithm that transforms them into linear programs which terminates and preserves the equivalence between the original and the linearised program. We have proved that dimension bounded programs are piecewise linear and therefore, such an algorithm exists for them. These dimension bounded programs are the result of choosing a natural value k and applying the transformation rules given in Chapter 3 to a set of Horn clauses P with no syntactic restrictions of any kind. $P^{[k]}$ represents an under-approximation of program P in the sense that the set of all possible derivations trees of $P^{[k]}$ are a subset of P's derivation trees where we only consider those of dimension at-most k. Thus, if $P^{[k]}$ is unsatisfiable then we can affirm that P is also unsatisfiable. On the other hand, if we find a solution for $P^{[k]}$, nothing can be said about the satisifiability of P. In addition, we have proved that applying ELP procedure to $P^{[k]}$ preserves the equivalence between the later and the resulting linear program. This means that there exists a one-to-one correspondence between the set of solutions for $P^{[k]}$ and the set of solutions for the resulting linear program.

We conclude that, if the linear program that results from applying these transformations to P is unsatisfiable then P is also unsatisfiable. If a solution is found for the linear program, nothing can be said about the satisfiability of P. Although dealing with the later case is outside the scope of this work, the notion of proof decomposition given in [3] can be applied. Thus, if a solution found for the linear program does not yield to a solution for P then we can increase the value k to $k + 1$, obtaining $P^{[k+1]}$. If the solution for the linear program resulting from applying ELP procedure to $P^{[k+1]}$ does not yield to a solution yet, the same steps can be repeated increasing consecutively the value of k until finding a solution for P, or until resources are exhausted.

Finally, as a part of the work that I carried out in the research institute IMDEA Software I implemented the ELP procedure in the logic programming language Prolog. This procedure takes as input a dimension bounded program specified

[1]https://github.com/SRI-CSL/sally

in Prolog syntax and gives as a result a linear program preserving the equivalence between the input program and the output program, and a set of Eureka Definitions built during the procedure. This code is public in a Git repository.[2]

[2]https://github.com/elenagutiv/Linearisation-2015

Bibliography

[1] F. N. Afrati, M. Gergatsoulis, and F. Toni. Linearisability on datalog programs. *Theoretical Computer Science*, 308(1-3):119–226, 2003.

[2] E. D. Angelis, F. Fioravanti, A. Pettorosi, and M. Proietti. Verimap: A tool for verifying programs through transformations. *In E. Abraham and K. Havelund, editors, TACAS*, 8413 of LNCS:568–574, 2014.

[3] B. Kafle, J. P. Gallagher, and P. Ganty. Decomposition by tree dimension in Horn clause verification. 2015.

[4] J.-L. Lasser, M. Maher, and K. Marriott. Unification revisited, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming. *Morgan Kaufmann Publishers*, pages 587–625, 1988.

[5] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1994.