Final Thesis

Efficient Bottom-up Evaluation of Magic Programs

by

Jan Carlson

LiTH-IDA-Ex-00/1

2000-01-20

Supervisor: Jan Maluszynski
Examiner: Jan Maluszynski
Abstract

Bottom-up evaluation of logic programs is a well-known technique typically used in deductive databases and in abstract interpretation. In these applications, bottom-up evaluation is often preceded by a transformation of the program into a so called magic program. The problem addressed in this report is how to optimize bottom-up evaluation for this special case. The report presents a new bottom-up evaluation method for magic programs, based on a combination of two known optimizations, induced magic and the use of strongly connected components.

Induced magic takes advantage of the structure of magic programs to avoid some recomputation. Magic programs are characterized by clause bodies with a high degree of overlapping, and the method uses this information to avoid some recomputation.

Using strongly connected components, the program is statically divided into partitions that can be handled separately. This results in smaller data structures, decreasing the computation time.

The report discusses the problem of combining these two methods, and presents a method where the basic ideas of both are used. It also outlines an efficient implementation of the method. Finally, it describes a system where bottom-up evaluation is used to compute regular approximations of magic programs. It is shown that the method can be incorporated into this system, and that this results in improved performance for the benchmark programs.
People who use magic without knowing what they are doing usually come to a sticky end. All over the entire room, sometimes.

Terry Pratchett, Moving Pictures
Contents

1 Introduction ............................................. 1
  1.1 Background ........................................ 2
  1.2 The Contributions ................................ 2
  1.3 Report Overview .................................. 2
  1.4 Acknowledgments .................................. 3

2 Preliminaries .......................................... 5
  2.1 Fixed Points ....................................... 5
  2.2 Logic Programming ................................ 6
  2.3 Bottom-up Evaluation of Logic Programs ............ 7
  2.4 Naive and Semi-naive Implementation ............... 10
  2.5 Strongly Connected Components ..................... 11
  2.6 Magic Transformation ............................. 13
  2.7 Induced Magic .................................... 14

3 Combining Induced Magic and SCCs .................. 17
  3.1 The Problem ...................................... 17
  3.2 Solutions ......................................... 18

4 Compressed Magic ...................................... 21
  4.1 Overlapping Clauses and Compressed Sets .......... 21
  4.2 Evaluation of Compressed Programs ................. 23

5 Implementation Issues .................................. 25
  5.1 Sorting Clauses .................................... 25
  5.2 Compressing a Program ............................ 27
  5.3 Data Structures ................................... 29

6 Properties of SCCs in Magic Programs ............... 31
  6.1 Dependency Loops in Magic Programs ............... 31
  6.2 Overlapping Modulo Variable Renaming ............. 34
7 Regular Approximations, an Application 37
  7.1 Regular Term Grammars 37
  7.2 Approximation as Bottom-up Evaluation 40
  7.3 Avoiding Non-termination 41
  7.4 Benchmarks 42

8 Conclusions 45
  8.1 Efficiency Discussion 45
  8.2 Future Research 46
  8.3 Optimized Compressed Magic 46
Chapter 1

Introduction

In 1998, Pawel Pietrzak published a thesis [Pie98] on the subject of locating errors in constraint logic programs by static program analysis, including a prototype diagnosis tool for the programming language CHIP. An important part of this diagnoser is a type inference system that computes an approximation of a given program, based on the regular approximation algorithm presented in [GdW92, GdW94].

Originally, the objective of the work was to investigate the possibility of optimizing this approximation algorithm by using the fact that the programs that are being approximated in the system contain many similar clauses. A similar idea for general bottom-up evaluation had been presented by Michael Codish [Cod99], but studying his work showed that his method was difficult to combine with some of the optimizations used in the current implementation.

It turned out to be possible, however, to construct a method based on a trade-off between two conflicting optimization techniques. This method was proven to be correct and later implemented as a part of the static analyzer. This implementation was also used to gather some statistical data to support the claim that this new method was in fact an improvement.

When looking for a nice way to present this new algorithm for regular approximation of magic programs it became clear that the basic concept of the algorithm was not restricted to approximation, but would work for other bottom-up techniques as well. This resulted in a definition of a general bottom-up evaluation framework for which different improvements, including this new algorithm, could be explained.

Having said this about the work behind the material presented in this report, we will now focus only on the issue of bottom-up evaluating magic programs in an efficient way.
1.1 Background

Bottom-up evaluation of logic programs is a well-known technique typically used in deductive databases and in abstract interpretation. Its fundamental operation is the application of a logic program to a set of facts to generate new facts. This relates to the lattice theory issue of computing fixed points of monotonic mappings. Many variants of the basic bottom-up technique have been proposed, as well as ideas of how to implement them efficiently.

A specific application of the bottom-up scheme concerns the magic transformation. This transformation, applied to a logic program, produces a magic program that contains information about what a top-down evaluation of the original program would look like. By bottom-up evaluating the magic program we explore the properties of such a top-down evaluation.

This report presents an optimization of the standard bottom-up evaluation implementation for the special case when it is applied to the result of the magic transformation. The method is based on a combination of two known optimizations. The first is to statically divide the program into partitions that can be handled separately [MUV86]. The second is to take advantage of the structure of magic programs to avoid some recomputation [Cod99].

1.2 The Contributions

The main contributions of the report are:

- A general framework for bottom-up evaluation of logic programs.

- An efficient method for bottom-up evaluation of magic programs.

- An implementation of the method.

1.3 Report Overview

The report is organized as follows. Chapter 2 introduces some basic concepts and presents the bottom-up evaluation framework. It also discusses some existing optimizations that are used as the basis for our method. Chapter 3 defines the problem and outlines the proposed approach. The outline is concretized in Chapter 4 where the method is presented. Chapter 5 surveys some implementation issues, and Chapter 6 discusses some properties of magic programs that influence the efficiency of our method. Chapter 7 describes how the method can be applied to the task of computing regular approximations. A final discussion including suggestions of future research is contained in Chapter 8.
1.4 Acknowledgments

I would like to express my warmest gratitude to my supervisor and examiner Jan Małuszyński, for his help during the project. I would also like to thank Andreas Sjöstedt and my opponent Tomas Nyström for reading my drafts and for their comments and suggestions. Finally, many thanks to the members of TCSLAB for the help, the inspiration and the bombastic coffee break discussions.
Chapter 2

Preliminaries

In this chapter we present the background theory on which the new evaluation method is based. This includes some basic definitions and results from lattice theory and from the area of logic programming. The main part of the chapter is a presentation of a general framework for bottom-up evaluation of logic programs, and a survey of some existing optimizations that have influenced our method.

2.1 Fixed Points

This section sketches some concepts from lattice theory that will be needed for the definition of bottom-up evaluation. A more extensive survey of these issues can be found in [Llo87, Nil92]. These books are also the source of the notation and definitions of the basic concepts we use.

Definition 2.1.1 Let \( D \) be a non-empty set, and \( \sqsubseteq \) be a relation on \( D \). Then \( \langle D; \sqsubseteq \rangle \) is a complete lattice iff

- \( \sqsubseteq \) is reflexive, antisymmetric and transitive.
- For any non-empty \( X \subseteq D \), \( D \) contains a least upper bound and a greatest lower bound of \( X \) with respect to \( \sqsubseteq \).

We say that \( \langle D; \sqsubseteq \rangle \) is of finite length if there exist no infinite increasing sequence \( x_0 \sqsubseteq x_1 \sqsubseteq \ldots \) of unique elements in \( D \).

Definition 2.1.2 Let \( \langle D; \sqsubseteq \rangle \) be a complete lattice and \( T : D \to D \) be a mapping. A fixed point of \( T \) is an element \( d \in D \) such that \( T(d) = d \). The least fixed point of \( T \) is a fixed point \( a \) of \( T \) such that \( a \sqsubseteq b \) for all fixed points \( b \) of \( T \).

Our interest in fixed points and how they might be computed arises from the fact that bottom-up evaluation of magic programs can be viewed as an instance of fixed point computation. Therefore, it would be of interest to know under what conditions we can be sure that a fixed point exists. Another important question is how to compute fixed points.
In lattice theory these issues are addressed in the following theorems. Proofs are omitted but can be found in [Nil92].

**Theorem 2.1.1** Let \( \langle D; \sqsubseteq \rangle \) be a complete lattice and \( T : D \to D \) be a monotonic mapping. Then \( T \) has a least fixed point.

**Definition 2.1.3** We define \( T \uparrow n \) as follows (\( \bot \) denotes the bottom element of the lattice).

\[
T \uparrow 0 := \bot \\
T \uparrow n := T(T \uparrow (n - 1)) \quad \text{for } n \geq 1
\]

**Theorem 2.1.2** Let \( \langle D; \sqsubseteq \rangle \) be a complete lattice of finite length, and let \( T : D \to D \) be a monotonic mapping. Then there exist a finite \( n \) such that \( T \uparrow n = T \uparrow (n + 1) \).

Theorem 2.1.2 provides us with an algorithm for computing the least fixed point of a monotonic mapping. We start with the bottom element of the lattice, and iteratively apply \( T \) until two consecutive iterations produce the same result. This result is the least fixed point of \( T \). Provided that the lattice is of finite length we are guaranteed that this algorithm will terminate.

### 2.2 Logic Programming

We also need to define some concepts from logic programming that will be used in the report.

**Definition 2.2.1** If \( t_1, \ldots, t_n \) are terms and \( p \) is a predicate of arity \( n \), then \( p(t_1, \ldots, t_n) \) is an *atomic formula* (or simply an *atom*).

**Definition 2.2.2** A *definite clause* is a logic formula on the form

\[
\forall x_1 \ldots \forall x_n (A \lor \neg B_1 \lor \ldots \lor \neg B_n)
\]

where \( A \) and \( B_i \) are atoms, and \( x_1, \ldots, x_n \) are all the variables occurring in \( (A \lor \neg B_1 \lor \ldots \lor \neg B_n) \).

In logic programming we normally write a definite clause as

\[
A \leftarrow B_1, \ldots, B_n
\]

Also, the clause \( A \leftarrow \text{true} \) is often written simply as \( A \).

**Definition 2.2.3** A *definite logic program* is a finite set of definite clauses.

**Definition 2.2.4** A *definite goal* is a formula on the form

\[
B_1, \ldots, B_n
\]

Throughout the report we will refer to definite clauses simply as clauses. We will also abbreviate definite programs and definite goals in the same way.
2.3 Bottom-up Evaluation of Logic Programs

In Section 2.1, we addressed the question of computing a fixed point of a mapping. Now, we will focus our attention on the case when the mapping is defined by a logic program.

The term bottom-up evaluation sometimes refers solely to the task of generating all the logical consequences of a given program, called the least Herbrand model of the program. We will define bottom-up evaluation in a more general way, making the construction of least Herbrand models an instance of bottom-up evaluation. An example of another instance is the method of computing a regular approximation of a given program described in Chapter 7.

The fundamental idea of bottom-up evaluation is that we see each clause \( C \) of a logic program as an operator \( O^C : D \to D \) on a certain domain \( D \). Similarly, the whole program \( P \) can be seen as the operator \( O^P : D \to D \), defined as \( O^P(d) := \bigcup_{C \in P} O^C(d) \).

Different instances of bottom-up evaluation use different domains and different definitions of the operations, but the goal is always to find a fixed point of \( O^P \). The evaluation consist of starting from the bottom element of the lattice \( d_0 = \bot \) and iteratively computing \( d_i = O^P(d_{i-1}) \) until we find a fixed point of \( O^P \), that is \( d_i \in D \) for which \( d_i = O^P(d_i) \).

Section 2.1 provided simple requirements for the existence of fixed points and the possibility to compute them. This means that if \( \langle D; \subseteq \rangle \) is a complete lattice of finite length and \( O^P \) is monotonic, the bottom-up algorithm is guaranteed to terminate.

**Example 2.3.1** Let us consider the task of computing the least Herbrand model of a datalog program. In this instance of bottom-up evaluation, \( D \) is the powerset of all ground atomic formulas over the alphabet used in the program, and \( O^P \) is the well-known immediate consequence operator \( T_P \). Datalog programs contain no functors, so \( D \) is finite, making \( \langle D; \subseteq \rangle \) a complete lattice of finite length. Since \( T_P \) is monotonic, this instance of bottom-up evaluation is guaranteed to terminate.

With \( P \) being the datalog program

\[
\begin{align*}
C_1 & : \text{married}(a, b) \quad \leftarrow \quad \text{true}. \\
C_2 & : \text{married}(X, Y) \quad \leftarrow \quad \text{married}(Y, X).
\end{align*}
\]

we would have the following evaluation

\[
\begin{align*}
\quad d_0 & = \{ \} \\
\quad d_1 & = O^P(d_0) = O^{C_1}(d_0) \cup O^{C_2}(d_0) = \\
\quad & = \{ \text{married}(a, b) \} \cup \{ \} = \\
\quad & = \{ \text{married}(a, b) \} \\
\quad d_2 & = O^P(d_1) = O^{C_1}(d_1) \cup O^{C_2}(d_1) = \\
\quad & = \{ \text{married}(a, b) \} \cup \{ \text{married}(b, a) \} = \\
\quad & = \{ \text{married}(a, b), \text{married}(b, a) \} \\
\quad d_3 & = O^P(d_2) = O^{C_1}(d_2) \cup O^{C_2}(d_2) = \\
\quad & = \{ \text{married}(a, b) \} \cup \{ \text{married}(b, a), \text{married}(a, b) \} = \\
\quad & = \{ \text{married}(a, b), \text{married}(b, a) \}
\end{align*}
\]
Looking closer at the $O^C$ operator, we find that the computation of $O^C(d)$ can be divided into three parts. First each body atom in $C$ is solved with respect to the facts in $d$, resulting in sets of variable bindings. In the next step these results are combined, producing a single set of variable bindings corresponding to the solution of the whole body. Finally, these bindings are used together with the head of $C$ to produce new facts.

**Example 2.3.2** Consider the computation of $O^{C_2}(d_2)$ in the example above. The first step results in the bindings $\{X = a, Y = b\}, \{X = b, Y = a\}$. The second step is invisible, since $C_2$ does not have more than one body atom. In the third step, the bindings are used to instantiate the head, resulting in $\{\text{married}(b, a), \text{married}(a, b)\}$.

The possibility of dividing the computation into these three steps is not specific for this instance of bottom-up evaluation. On the contrary, we will use it to further define $O^C$.

Let $D$ be any complete lattice, and let $E$ contain variable bindings. Finally, let $T$ be the set of all atomic formulas over the current language. Now, the three steps of computing $O^C$ can be described using the following three operators:

$$
\Delta : T \times D \rightarrow E \\
\oplus : E \times E \rightarrow E \\
\forall : T \times E \rightarrow D
$$

The role of $\Delta$ can be described informally as collecting information about the variables of a body atom with respect to an element from $D$. The operator $\oplus$ combines the information gained from the atoms, and finally $\forall$ constructs an element in $D$ from the head of the clauses and the bindings from the whole body. A graphical presentation of this can be found in Figure 2.1.

**Definition 2.3.1** If $C$ is the clause $h \leftarrow b_1, \ldots, b_n$ we define $O^C(d)$ to be

$$
O^C(d) = \forall(h, \bigoplus_{i=1}^{n} (\Delta (b_i, d)))
$$

**Example 2.3.3** When bottom-up evaluation is used to compute the least Herbrand model, $D$ is the powerset of all ground atomic formulas, and $E$ is the powerset of all substitutions binding variables to ground terms. The three operations are defined as follows.

$$
\Delta (t, d) = \{\theta \mid \theta \text{ is a mgu of } t \text{ and an element in } d\}
$$

$$
\oplus(e_1, e_2) = \{\theta_1 \cup \theta_2 \mid \theta_1 \in e_1 \land \theta_2 \in e_2 \land
\land [(X/t_1 \in \theta_1 \land X/t_2 \in \theta_2) \Rightarrow (t_1 = t_2)]\}
$$

$$
\forall(t, e) = \{\theta \mid \theta \in e \land \theta \text{ is ground}\}
$$

Figure 2.2 shows how these operations are used during the computation of $O^C(d)$. 

8
Figure 2.1: Computation of $O^C(d)$, where $C$ is the clause $h \leftarrow b_1, \ldots, b_n$

Figure 2.2: Computation of $O^C(d)$, for the case when bottom-up evaluation is used to compute least Herbrand models. $C$ is the clause $p(X) \leftarrow q(X, Y)$, $r(Y)$ and $d = \{q(a, a), q(b, b), q(c, d), r(b), r(c), r(d)\}$.

**Example 2.3.4** Another example of bottom-up evaluation is the computation of types for a logic program. By a type, we mean a restriction on the values the arguments of a predicate can take. In this example, we will use the following types:

$$T = \{\varepsilon, int, nat, any\}$$

The type $\varepsilon$ denote the empty set. The remaining types denote the set of all integers, natural numbers and terms, respectively.

For a predicate $p$ of arity $n$, the specification $p(\tau_1, \ldots, \tau_n)$, where $\tau_i \in T$, means that the $i$-th argument of $p$ is in $\tau_i$, for $1 \leq i \leq n$. The elements of the domain $D$ are sets of such specifications, in which no predicate occurs more than once. We will not define the three operations for this instance of bottom-up evaluation, but Figure 2.3 sketches their use.
In this example we used a fixed, and very small, set of types. Chapter 7 will present a more general way of constructing types for logic programs.

Figure 2.3: Computation of $O^C(d)$, for the case when bottom-up evaluation is used to construct types. $C$ is the clause $p(X) \leftarrow q(X, Y), r(Y)$ and $d = \{q(int, int), r(nat)\}$.

2.4 Naive and Semi-naive Implementation

An algorithm for bottom-up evaluation, based directly on the definition in the previous section, is outlined in Figure 2.4. This algorithm is often referred to

\[
\begin{align*}
\text{facts} & := \bot \\
\text{repeat} & \\
& \quad \text{old} := \text{facts} \\
& \quad \text{facts} := O^P(\text{facts}) \\
\text{until} & \quad \text{old} = \text{facts}
\end{align*}
\]

as the naive algorithm, reflecting the fact that it is rather inefficient.

The main drawback of the naive algorithm is that once a specific fact is inferred, it will be inferred again in every following iteration, leading to a lot of recomputation.

An improved version, called the semi-naive algorithm, has been suggested. It is based on the idea that we only infer a new fact if at least one of the facts used to infer it was inferred in the very previous step. For this purpose, we define the operator $\Delta O^P(f, d)$ to be the set of the facts in $O^P(f)$ meeting the additional restriction that at least one fact from $d$ should have been used when inferring them. The semi-naive algorithm is presented in Figure 2.5.
facts := $O^P(\bot)$
delta := facts
repeat
  delta := $\Delta O^P$(facts, delta)
facts := facts $\cup$ delta
until delta = {} 

Figure 2.5: Semi-naive algorithm

2.5 Strongly Connected Components

Another way of improving the bottom-up evaluation is by dividing the iteration into several smaller iterations, each of them considering a subset of the program. For this method to work, we need to be sure that the iteration is divided in such a way that during the computation we never need information that will be computed later.

To assure this, we look at dependencies between predicates in the program, using the following definitions.

Definition 2.5.1 A predicate $a$ is defined by a predicate $b$, in a program $P$ iff $P$ contains a clause with $a$ in its head and $b$ in its body. We also say that $a$ depends on $b$ iff $\langle a, b \rangle$ is in the transitive closure of "is defined by".

Definition 2.5.2 The predicate-dependency graph of a program $P$ is defined as follows. The vertices of the graph are the predicates appearing in $P$, and $\langle p_1, p_2 \rangle$ is a directed edge in the graph iff $p_2$ is defined by $p_1$ in $P$.

Example 2.5.1 Let $P$ be this rather peculiar program

\[
\begin{align*}
  a([],[]) &\quad c([],1) \\
  a([X][Xs], [Y][Ys]) &\leftarrow c([X][Xs], Y) &\leftarrow c[X][Xs], Y) \\
  b(X, Y) &\leftarrow e(X) \\
  a(Xs, Ys) &\leftarrow b(Xs, Y) \\
  b([],0) &\leftarrow d(a) \\
  b([X][Xs], Y) &\leftarrow e(e) \\
  d(X), &
\end{align*}
\]

The predicate-dependency graph of $P$ is

\[
\begin{array}{c}
  e \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \\
  \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \\
  d \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \\
\end{array}
\]

Definition 2.5.3 A strongly connected component (SCC) in a graph $G$ is a maximal subgraph of $G$ such that for every pair of vertices $u$ and $v$ in the subgraph, there is a directed path from $u$ to $v$ and a directed path from $v$ to $u$.
The SCCs of a graph form an acyclic graph that can be topologically sorted to give a sequence of SCCs.

If we look at the SCCs in the predicate-dependency graph of a program, the predicates in a single component all depend on each other. Looking at a topologically sorted sequence of such SCCs, we can see that a predicate \( p \) depend on all predicates in the same and previous SCCs. Also, all predicates in the same and subsequent SCCs depend on \( p \).

**Example 2.5.2** The SCCs of the graph from the previous example form the following graph

\[
\begin{align*}
\{e\} & \rightarrow \{b, c\} \rightarrow \{a\} \\
\{d\} & \rightarrow \{b, c\} \rightarrow \{a\}
\end{align*}
\]

which can be topologically sorted into one of the following sequences

\[
\begin{align*}
\{e\} & \rightarrow \{d\} \rightarrow \{b, c\} \rightarrow \{a\} \\
\{d\} & \rightarrow \{e\} \rightarrow \{b, c\} \rightarrow \{a\}
\end{align*}
\]

Having computed a sequence of SCCs from the predicate-dependency graph of a program, we can traverse the sequence performing one fixpoint iteration on each component. We start with the first SCC in the sequence, using the naive or semi-naive algorithm to compute the fixed point of the subset of the program defining the predicates of that SCC. The inferred facts from each iteration are used as the initial value when iterating over the next SCC.

The correctness of the SCC method is guaranteed by showing that all clauses of the program will be considered, and that during the iteration over a component, no facts inferred in subsequent components would contribute to the result. Assuming the opposite implies that a predicate in the current component depends on a predicate in a subsequent component, contradicting the fact that the components are topologically sorted.

The reason why the SCC method indeed is more efficient than the original version, even though it introduces the overhead cost of computing SCCs, is that the size of the data structures used in the iterations are considerably smaller than if the whole program is computed together, with more efficient searching and data handling as a result. A linear algorithm for computing the SCCs of a graph is available in [CLR90].

The SCCs can also be used to identify non-recursive predicates. This is useful because when processing the SCC containing (only) a non-recursive predicate no fixpoint iteration is needed. After applying the operator once we know that a fixed point is reached.
2.6 Magic Transformation

Bottom-up evaluation of a program computes the least fixed point of the mapping $O^P$. In many applications we would like the possibility to restrict this computation. For example, imagine that we add to the program in Example 2.3.1 a lot of facts about who is married to whom. Computing the least Herbrand model is far too much work if we only want to know if $a$ and $b$ are married, or to whom $c$ is married.

Magic transformation, described in [DR94, Nil92], introduces a possibility of goal-directed bottom-up evaluation. From the original program $P$, a new program $M^P$ is constructed using the following transformation.

**Definition 2.6.1** $M^P$ is the smallest program such that for each clause

$$h \leftarrow b_1, \ldots, b_n$$

in $P$, $M^P$ contains the clauses

$$\begin{align*}
\cdot b_1 & \leftarrow \cdot h \\
\cdot b_2 & \leftarrow \cdot h, \cdot b_1^* \\
\cdot b_3 & \leftarrow \cdot h, \cdot b_1^*, \cdot b_2^* \\
& \vdots \\
\cdot b_n & \leftarrow \cdot h, \cdot b_1^*, \cdot b_2^*, \ldots, \cdot b_{n-1}^* \\
h^* & \leftarrow \cdot h, \cdot b_1^*, \cdot b_2^*, \ldots, \cdot b_{n-1}^*, \cdot b_n^*
\end{align*}$$

Constructing $M^P$ is a way of restricting the bottom-up evaluation to facts that are needed for a given task. The new predicate $\cdot a$, can be interpreted as "$a$ is needed", and $a^*$ as "$a$ is established". Using this interpretation, the first magic clause means "if $h$ is needed then $b_1$ is needed". The second clause of $M^P$ states that "if $h$ is needed, and if we have established $b_1$, then $b_2$ is needed as well". This statement is a nice illustration of the basic idea of the magic transformation. According to the magic program, $b_2$ is not to be regarded unless we have inferred $b_1$.

To the magic program $M^P$, we add a clause on the form $\cdot g \leftarrow true$, simply stating that we need $g$ to be computed.

**Example 2.6.1** Let $P$ be the program

$$\begin{align*}
append([], X, X) \\
append([A|As], B, [A|Cs]) & \leftarrow \\
append(As, B, Cs)
\end{align*}$$

Using bottom-up evaluation to construct the least Herbrand model of this program, is a bad idea. Assume that we are really only interested in consequences that are instances of the atom $append([1], [2], X)$. Then we first construct $M^P$: 13
\[\text{append}^\bullet([], X, X) \leftarrow \]
\[\bullet\text{append}([], X, X).\]
\[\bullet\text{append}(A, B, C, s) \leftarrow \]
\[\bullet\text{append}([A \mid A], B, [A \mid C], s).\]
\[\text{append}^\bullet([A \mid A], B, [A \mid C], s) \leftarrow \]
\[\bullet\text{append}([A \mid A], B, [A \mid C], s).\]
\[\text{append}^\bullet(As, B, C, s).\]

The bottom-up evaluation of \( M^P \cup \{\bullet\text{append}([1], [2], X)\} \), using \( T^P \) as the operator, would look like this:

\[
d_0 = \{\} \\
d_1 = \{\bullet\text{append}([1], [2], X)\} \\
d_2 = \{\bullet\text{append}([1], [2], X), \bullet\text{append}([], [2], X)\} \\
d_3 = \{\bullet\text{append}([1], [2], X), \bullet\text{append}([], [2], X), \text{append}^\bullet([], [2], [2])\} \\
d_4 = \{\bullet\text{append}([1], [2], X), \bullet\text{append}([], [2], X), \text{append}^\bullet([], [2], [2]), \text{append}^\bullet([1], [2], [1, 2])\} \\
d_5 = d_4
\]

The intuition behind the magic transformation can also be explained in another way. We can see \( M^P \) as a description of what a top-down evaluation of \( P \) would look like, assuming a left-to-right computation rule as in SLD-resolution. We let the predicate \( \bullet p \) represent the "call of \( p \)" while \( p^\bullet \) represents the "success of \( p \)". Then the first magic clause can be translated into "if \( h \) is called, then \( b_1 \) will be called", while the second clause states that "if \( h \) is called and the call of \( b_1 \) has succeeded, then \( b_2 \) will be called".

From this point of view, the result of bottom-up evaluating the program \( M^P \cup \{\bullet g\} \) will describe intermediate results of a goal directed top-down evaluation of \( P \) with respect to the goal \( g \). In Example 2.6.1, the new fact inferred in each iteration step correspond roughly to the steps used in SLD-resolution to construct a refutation tree for the goal \( \leftarrow \text{append}([1], [2], X) \) and the original program. SLD-resolution is described for example in [NM90].

For a more thorough discussion on the relation between magic transformation and top-down evaluation see for example [DR94].

### 2.7 Induced Magic

When the bottom-up evaluation is applied to the result of a magic transformation, it is possible to use the knowledge about the structure of magic programs to avoid some recomputation. One important feature of magic clauses derived from the same original clause is the fact that their bodies overlap, and since bottom-up evaluation is based on solving the clauses of the program using the facts derived so far, identical parts of bodies are solved over and over again.

The idea presented by Michael Codish in [Cod99] is to process at the same time all the magic clauses originating from the same clause. Looking at the
magic clauses generated by a single clause in the original program (Definition 2.6.1) we notice that when we solve the body of the last magic clause from left to right, prefixes corresponding to the bodies of all other clauses will be solved along the way.

Codish also points out that this method makes it possible to skip the magic transformation altogether, since traversing the original clause provides the information needed. Figure 2.6 shows the algorithm for computing $O^{M_i}(d)$, where $M_1, \ldots, M_{n+1}$ are the magic clauses generated by the clause $h \leftarrow b_1, \ldots, b_n$ in the original program. A description of the execution of this algorithm can

$$
e := \Delta (\bullet h, d)$$

$$\textbf{for } i = 1 \textbf{ to } n \textbf{ begin}$$

$$O^{M_i}(d) := \forall (\bullet b_i, e)$$

$$e := \oplus (e, \Delta (b_i^*, d))$$

$$\textbf{end}$$

$$O^{M_{n+1}}(d) := \forall (h^*, e)$$

Figure 2.6: Induced magic

be found in Figure 2.7.

Induced magic is an instance of a well-known technique of improving efficiency of algorithms, called dynamic programming [AHU74].
Chapter 3

Combining Induced Magic and SCCs

The previous chapter presented three suggestions of how to implement bottom-up evaluation efficiently (the semi-naive algorithm, SCCs and induced magic), and an obvious question is whether it is possible to combine them. Incorporating the semi-naive approach into the other two seems quite straightforward, but what about combining SCCs with induced magic?

3.1 The Problem

In a complete combination of the two methods, we would start by constructing the magic program and computing its SCCs. When iterating over a component we would use induced magic, considering only the longest of the magic clauses generated by each clause in the original program. The problem with this suggested solution is that the magic clauses generated by a single clause in the original program often define predicates in different components. For such magic clauses, the SCC approach states that they should be regarded in different iterations while induced magic states that they should be computed together.

Example 3.1.1 Let $P$ be the program

\[
\begin{align*}
    &a([],) \\
    &a([X|Xs]) \leftarrow b(X), a(Xs), c(X) \\
    &b(1) \\
    &c(1)
\end{align*}
\]

then $M^P$ is
\[\alpha^*(\mathbb{I}) \leftarrow \alpha(\mathbb{I})\]
\[\beta(X) \leftarrow \alpha([X]Xs)\]
\[\alpha([X]Xs) \leftarrow \alpha([X]Xs), \beta(X)\]
\[\alpha([X]Xs) \leftarrow \alpha([X]Xs), \beta(X), \alpha(Xs)\]
\[\alpha^*([X]Xs) \leftarrow \alpha([X]Xs), \beta(X), \alpha(Xs), \alpha(X)\]
\[\beta(1) \leftarrow b(1)\]
\[\alpha^*(1) \leftarrow \alpha(1)\]

and the (only possible) sequence of SCCs is
\[
\{\alpha, \beta, \alpha^*\} \rightarrow \{\alpha, \alpha^*, \alpha^*\}
\]

Using induced magic, we would like to compute the clauses (2)-(5) together, but the predicates defined by those clauses (\(\beta, \alpha, \alpha^*, \alpha^*\)) belong to different components and should thus be considered during different iterations.

This example shows that we can not directly combine the SCC method with induced magic. However, we still want to combine the underlying ideas of the two methods. We want the computation to be divided into a sequence of smaller fixpoint iterations while still avoiding the recomputation caused by overlapping bodies. The rest of this chapter will discuss how this may be accomplished.

### 3.2 Solutions

One possible way of solving the problem would be to find another way of dividing the program, where the overlapping magic clauses are always computed in the same iteration. The result will be correct as long as we still make sure that if \(\alpha\) depends on \(\beta\) then \(\alpha\) is not processed before \(\beta\).

This may sound like a nice solution, but it turns out that it will very often result in very big components, as in the following example.

**Example 3.2.1** If \(P\) is the program

\[
\alpha(X) \leftarrow b(X)
\]
\[
b(\mathbb{I})
\]
\[
b([X]Xs) \leftarrow c(X), b(Xs)
\]
\[
c(1)
\]

then \(M^P\) is

\[
\beta(X) \leftarrow \alpha(X)
\]
\[
\alpha^*(X) \leftarrow \alpha(X), \beta(X)
\]
\[
\beta(\mathbb{I}) \leftarrow \beta(\mathbb{I})
\]
\[
\alpha^*(X) \leftarrow \beta([X]Xs)
\]
\[
\alpha^*(Xs) \leftarrow \beta([X]Xs), \alpha^*(X)
\]
\[
\alpha^*([X]Xs) \leftarrow \beta([X]Xs), \alpha^*(X), \beta(Xs)
\]
\[
\alpha^*(1) \leftarrow \alpha(1)
\]

18
In the original SCC method, the iteration would be divided according to the sequence
\[
\{a\} \rightarrow \{b, c, e, a\} \rightarrow \{b\} \rightarrow \{a\}
\]
If we want all overlapping clauses to be computed at the same time, we get the following sequence
\[
\{a\} \rightarrow \{b, c, e, a\}
\]
Suppose that we add the clause \(d(X) \leftarrow a(X)\) to the program. In the original SCC method this would only result in two non-recursive SCCs being added to the sequence. \(\{d\}\) is added at the beginning, and \(\{a\}\) at the end of the sequence.

In the new method, we would end up with the following dependency graph since \(d\) depend on \(a\), and \(a\) and \(d\) are to be computed together
\[
\{d\} \rightarrow \{a, c, d\}
\]

Another solution is to do it the other way around. The SCC method is kept intact and we use a more general method than induced magic to avoid some of the recomputation caused by the overlapping bodies in magic programs. We compute the magic transformation and the SCC sequence as usual, but at the beginning of each iteration we look at the clauses defining the predicates of that component in order to identify those that have overlapping bodies. During the iteration, these clauses are processed together to avoid recomputation.

Example 3.2.2 Consider the program in Example 3.1.1. The predicates in the second SCC (\(\{c, e, a\}\)) are defined by the following subset of \(M^P\)
\[
\begin{align*}
a^e([],[]) & \leftarrow a([],[]) \quad (1) \\
c^e(X) & \leftarrow a([X|Xs]), a^e(X), a^e(Xs) \quad (2) \\
a^e([X|Xs]) & \leftarrow a([X|Xs]), b^e(X), a^e(Xs), c^e(X) \quad (3) \\
c^e(1) & \leftarrow c(1) \quad (4)
\end{align*}
\]
so when the component is processed, we start by noticing that clause (2) and clause (3) overlap and should be treated together during the iteration.

This solution avoids the problem of big components, but introduces the risk of recompiling some overlapping clauses. It is an easy task to construct programs for which the first solution is reduced to ordinary induced magic, and programs for which the second solution is reduced to the original SCC method. However, it seems like the second solution will be rather efficient for most programs while the first one will almost always behave really bad. Some clues to why this is the case are presented in Section 6.1.

The rest of this report will only discuss the second solution. The main reason is that it seems to have a higher possibility of practical impact on the efficiency of bottom-up evaluation. Another reason is that it is easier to incorporate this solution into the existing system.
Chapter 4

Compressed Magic

The previous chapter discussed why combining induced magic with the SCC method is not trivial. We sketched a method based on the idea that we can identify the overlapping clauses that happen to define predicates in the same SCC and compute those together. In this chapter we will give a full description of this method, which we will refer to as "compressed magic".

4.1 Overlapping Clauses and Compressed Sets

In the previous chapters we have occasionally referred, quite informally, to overlapping clauses. We now give a precise definition of this concept.

**Definition 4.1.1** Let $C_1$ and $C_2$ be the following clauses

\[
C_1 : \quad h_1 \leftarrow b_1, \ldots, b_n
\]
\[
C_2 : \quad h_2 \leftarrow c_1, \ldots, c_m
\]

Then $C_1$ overlaps $C_2$ iff $n \geq m$ and

\[
\exists \theta : \quad b_1, \ldots, b_m = (c_1, \ldots, c_m)\theta
\]

with $\theta$ being a variable renaming.

**Example 4.1.1** Considering the clauses

\[
C_1 : \quad a(X) \leftarrow b(X, Y), \ c(Y), \ d(Y)
\]
\[
C_2 : \quad e(Y, X) \leftarrow b(Y, X), \ e(X)
\]
\[
C_3 : \quad f(X) \leftarrow b(Y, X), \ e(Y)
\]

we find that $C_1$ overlaps $C_2$ (with $\theta = \{X/Y, Y/X\}$), but not $C_3$.

By this definition, the longest magic clause overlaps all the other magic clauses originating from the same clause. Thanks to the variable renaming we
also recognize overlapping magic clauses originating from different clauses in the original program. The potential use of this is discussed in Section 6.2.

Next, we need a way of representing overlapping clauses once they have been identified, so that they can be easily handled during the iteration.

**Definition 4.1.2** The compressed set

$$\{ \langle l_1, h_1 \rangle, \ldots, \langle l_n, h_n \rangle \} \leftarrow b_1, \ldots, b_m$$

represents the set containing the following \( n \) overlapping clauses:

\[
\begin{align*}
h_1 & \leftarrow b_1, \ldots, b_{i_1} \\
h_2 & \leftarrow b_1, \ldots, b_{i_2} \\
& \vdots \\
h_n & \leftarrow b_1, \ldots, b_{i_n}
\end{align*}
\]

**Example 4.1.2** The compressed set

$$\{ \langle 1, a(X) \rangle, \langle 1, b(X) \rangle, \langle 2, a(X, Y) \rangle \} \leftarrow c(X), d(X, Y)$$

represents the clauses

\[
\begin{align*}
a(X) & \leftarrow c(X) \\
b(X) & \leftarrow c(X) \\
a(X, Y) & \leftarrow c(X), d(X, Y)
\end{align*}
\]

The concept of overlapping extends naturally to compressed sets.

**Definition 4.1.3** Let \( R_1 \) and \( R_2 \) be the following compressed sets

\[
\begin{align*}
R_1 & : H_1 \leftarrow b_1, \ldots, b_n \\
R_2 & : H_2 \leftarrow c_1, \ldots, c_m
\end{align*}
\]

Then \( R_1 \) overlaps \( R_2 \) iff \( n \geq m \) and

\[
\exists \theta : b_1, \ldots, b_m = (c_1, \ldots, c_m)\theta
\]

with \( \theta \) being a variable renaming.

We notice that two overlapping sets can always be represented as a single one. Assume that \( R_1 \) and \( R_2 \) from the definition above represent the sets \( P_1 \) and \( P_2 \), respectively. Then the compressed set \( H_1 \cup \{ \langle l, h\theta \rangle \mid \langle l, h \rangle \in H_2 \} \leftarrow b_1, \ldots, b_n \) represents \( P_1 \cup P_2 \).

**Definition 4.1.4** If \( R_1, \ldots, R_n \) are compressed sets where each \( R_i \) represents the set \( P_i \), we define the set \( \{ R_1, \ldots, R_n \} \) to represent the set \( \bigcup_{i=1}^n P_i \).

**Definition 4.1.5** A set of compressed sets is **fully compressed** if it contains no overlapping compressed sets.
We notice that any program can be represented as a set of compressed sets, for example just by replacing each clause $h \leftarrow b_1, \ldots, b_n$ with the compressed set $\{ (n, h) \} \leftarrow b_1, \ldots, b_n$. More important, we can construct a fully compressed representation of any given program, by constructing one compressed set from any overlapping compressed sets as described above, until the program is fully compressed.

This method of constructing fully compressed representations of programs can be implemented in a straightforward way, but Chapter 5 presents a more efficient implementation.

**Example 4.1.3** In Example 3.2.2 we considered the component $\{ \bullet c, \bullet c, \bullet a \}$ defined by the clauses

\[
\begin{align*}
\bullet a([]) & \leftarrow \bullet a(()) \\
\bullet c(X) & \leftarrow \bullet a([X|Xs]), \bullet b(X), \bullet a(Xs) \\
\bullet a([X|Xs]) & \leftarrow \bullet a([X|Xs]), \bullet b(X), \bullet a(Xs), \bullet c(X) \\
\bullet c(1) & \leftarrow \bullet c(1)
\end{align*}
\]

The compressing algorithm would first construct the following compressed sets

\[
\begin{align*}
\{ (1, \bullet a([]) ) \} & \leftarrow \bullet a(()) \\
\{ (3, \bullet c(X) ) \} & \leftarrow \bullet a([X|Xs]), \bullet b(X), \bullet a(Xs) \\
\{ (4, \bullet a([X|Xs]) ) \} & \leftarrow \bullet a([X|Xs]), \bullet b(X), \bullet a(Xs), \bullet c(X) \\
\{ (1, \bullet c(1) ) \} & \leftarrow \bullet c(1)
\end{align*}
\]

Then, since the third compressed set overlaps the second one, it would merge them into a single one, resulting in

\[
\begin{align*}
\{ (1, \bullet a([]) ) \} & \leftarrow \bullet a(()) \\
\{ (3, \bullet c(X) ) \} & \leftarrow \bullet a([X|Xs]), \bullet b(X), \bullet a(Xs) \\
\{ (4, \bullet a([X|Xs]) ) \} & \leftarrow \bullet a([X|Xs]), \bullet b(X), \bullet a(Xs), \bullet c(X) \\
\{ (1, \bullet c(1) ) \} & \leftarrow \bullet c(1)
\end{align*}
\]

This representation does not contain any overlapping compressed sets, and thus it is a fully compressed representation of the program.

### 4.2 Evaluation of Compressed Programs

The previous section introduced a way of representing a program, where overlapping clauses are merged together as compressed sets. The reason for doing this was that it would make it possible to evaluate them together. This section presents the new evaluation algorithm.

Assume that the program $P$ is to be evaluated. We start by computing a fully compressed representation $\{ R_1, \ldots, R_n \}$ of the program, and during the iteration we use this representation to compute $O^P$. We know that $O^P(d) = O^{P_1} \cup O^{P_2} \cup \ldots O^{P_n}$, where $P_i$ is the set represented by $R_i$. All we need is an efficient method to compute $O^P$ for the case when $P$ is represented by a single compressed set $R$. If $R$ is
\[ H \leftarrow b_1, \ldots, b_n \]

we can compute \( O^P \) in the following way:

\[
e_i(d) = \Delta(b_1, d) \\
e_i(d) = e_{i-1}(d) \oplus \Delta(b_1, d) \quad 1 \leq i \leq n
\]

\[
O^P(d) = \bigcup_{(m,h) \in H} \triangledown(h, e_m(d))
\]

Figure 4.1 shows a part of the computation of a compressed set.

Trivially, this way of computing \( O^P(d) \) will give the same result as the original algorithm, but with the improvement that overlapping clauses are processed together.

![Diagram](image)

Figure 4.1: A part of the computation of \( O^P(d) \), where \( P \) is represented by the compressed set \( H \leftarrow b_1, \ldots, b_n \) and \( (m, h) \in H \).
Chapter 5

Implementation Issues

The compressing algorithm as it was described previously is quite far from a straightforward implementation. The idea of starting with a big set of compressed sets, and finding overlapping clauses to merge until the set contains no more overlapping clauses, seems difficult to implement. Instead we use another approach, sorting the clauses of the original program so that all overlapping clauses end up next to each other. Then only one pass of the sorted clauses is needed to create the fully compressed representation of the program.

5.1 Sorting Clauses

First, we need to define what we mean by a sorted sequence of clauses. Clauses are compared with respect to their body atoms from left to right. The atoms are compared primarily by predicates, and secondarily by their arguments from left to right. When comparing two arguments, we use the comparison “constants < variables < compound terms”. Within these groups, we use the following comparisons:

1. Variables are compared based on the position of their leftmost occurrence in the body (see Example 5.1.2).

2. Constants are compared in any fixed way, for example by the Prolog standard order.

3. Compound terms are compared first by arity, then by name of principal functor, and then by recursively comparing the arguments from left to right, using the same comparison as for atoms.

The reason for comparing clauses in this way is that we want a clause $C_1$ to overlap a clause $C_2$ iff the body of $C_2$ is considered equal to a prefix of the body of $C_1$ by our comparison. Since we defined overlapping modulo variable renaming, we need to compare variables as described above.
Example 5.1.1 The following sequence of clauses, where \( h_1, \ldots, h_8 \) are arbitrary heads, is correctly sorted:

\[
\begin{align*}
    h_1 & \leftarrow b(1) < \\
    h_2 & \leftarrow b(X) = \\
    h_3 & \leftarrow b(Y) < \\
    h_4 & \leftarrow b(X), c(X) < \\
    h_5 & \leftarrow c(X) < \\
    h_6 & \leftarrow c(Y), b(Y) < \\
    h_7 & \leftarrow c(X), b(Y) = \\
    h_8 & \leftarrow c(Y), b(X)
\end{align*}
\]

Example 5.1.2 The comparison of variables is illustrated by the following sequence:

\[
\begin{align*}
    h_1 & \leftarrow b(A), c(B, A), d(A) < \\
    h_2 & \leftarrow b(A), c(B, A), d(B) < \\
    h_3 & \leftarrow b(A), c(B, A), d(A)
\end{align*}
\]

The first two atoms of the three bodies are considered equal. The third atom of each body are all compound terms with the principal functor \( d \), and a variable argument. The variables are compared with respect to their leftmost occurrence in the three bodies respectively (underlined in the clauses above).

The following properties of the comparison operation will be needed when proving that the compression method is correct.

Lemma 5.1.1 The clause comparison operation defined above has the following properties:

1. \( < \) is transitive.
2. \( C_1 < C_2 \Rightarrow C_1 \) does not overlap \( C_2 \).
3. \( C_1 = C_2 \Rightarrow C_1 \) and \( C_2 \) overlap each other.
4. Let \( \leq \) denote the union of \(<\) and \(=\). If \( C_1 \leq C_2 \leq C_3 \) and \( C_3 \) overlaps \( C_1 \), then \( C_2 \) overlaps \( C_1 \).

Proof: The first three properties follow trivially from the description of the comparison, and from the definition of overlapping clauses. For the fourth property, we let \( x \) be the number of body atoms in \( C_1 \). Since \( C_3 \) overlaps \( C_1 \) we know that the \( x \) first body atoms of \( C_1 \) and \( C_3 \) are equal modulo variable renaming. This means that during the comparison of \( C_1 \) and \( C_3 \), the \( x \) first body atoms are considered equal. Then \( C_1 \leq C_2 \leq C_3 \) tells us that the \( x \) first body atoms of \( C_2 \) need also to be considered equal to those of \( C_1 \) and \( C_3 \). Finally, this implies that \( C_2 \) overlaps \( C_1 \). \qed

26
5.2 Compressing a Program

Before presenting the compression algorithm, we need to define three primitive operations. The implementation of these operations is quite straightforward. For a clause $C$ and a compressed set $R$ representing $P$, we define

$$
\text{comp}(C) := \text{a compressed set representing } \{C\}.
$$

$$(C, R) \in \text{overlaps} \quad \text{iff} \quad C \text{ overlaps all clauses in } P.
$$

$$
\text{merge}(C, R) := \begin{cases} 
R' \text{ representing } P \cup \{C\} & \text{if } (C, R) \in \text{overlaps} \\
\text{undefined} & \text{otherwise}
\end{cases}
$$

Note that we can check $\text{overlaps}(C, R)$ by comparing the bodies of $C$ and $R$, instead of comparing $C$ with all clauses in $P$.

Figure 5.1 presents the compression algorithm. Let $C_1, \ldots, C_n$ be a sorted sequence of clauses. After executing the algorithm, $P$ will be a fully compressed representation of $\{C_1, \ldots, C_n\}$.

\begin{verbatim}
P := {} 
R := \text{comp}(C_1) 
i := 1
for i := 1 to n begin 
    if $\text{overlaps}(C_{i+1}, R)$ then 
        R := $\text{merge}(C_{i+1}, R)$ 
    else 
        P := P $\cup \{R\}$ 
        R := $\text{comp}(C_{i+1})$
end
P := P $\cup \{R\}$
\end{verbatim}

Figure 5.1: Compressed magic

In order to show that this algorithm is correct, we will use the following loop invariant

\begin{align*}
\exists x \ (0 \leq x < i \leq n \land \\
P \text{ represents } \{C_1, \ldots, C_x\} \land \\
R \text{ represents } \{C_{x+1}, \ldots, C_i\} \land \\
P \cup \{R\} \text{ is fully compressed }
\end{align*}

The invariant is described graphically in Figure 5.2.

Trivially, the invariant holds when we first enter the loop (for $x = 0$). If the invariant holds when we leave the loop, then we can be sure that a correct answer has been computed. Since $i$ is incremented in each step of the loop, the matter of termination is equally trivial. Left to prove is that if the invariant

\begin{align*}
\exists x \ (0 \leq x < i \leq n \land \\
P \text{ represents } \{C_1, \ldots, C_x\} \land \\
R \text{ represents } \{C_{x+1}, \ldots, C_i\} \land \\
P \cup \{R\} \text{ is fully compressed }
\end{align*}
holds at the beginning of a pass through the loop, then it will still hold at the end of that pass (after \( i \) is increased by one).

Assume that the invariant holds and that \( i < n \). Let \( P, R \) and \( i \) denote the values of the corresponding variables at the start of a pass through the loop. Similarly, let \( P', R' \) and \( i' = i + 1 \) denote the values of those variables at the end of the pass. Also, let \( x \) be the value making the invariant true at the start. Our goal is to show that there exist a value \( x' \) such that the invariant hold at the end as well. To prove this, we will consider two possible cases, corresponding to the two branches of the if statement. These cases are described graphically in Figure 5.3 and 5.4.

![Figure 5.2: The loop invariant](image)

![Figure 5.3: The first case](image)

![Figure 5.4: The second case](image)

First consider the case where \((C_{j+1}, R) \in \text{overlaps}\). For this case \( x' = x \) will make the invariant true. It is quite easy to see that the conditions (1), (2) and (3) still holds. To prove (4), we assume that (4) is false, showing that this leads to a contradiction.

If \( P \cup \{ R' \} \) contains two overlapping compressed sets, one of them has to be \( R' \) since the invariant states that \( P \) is fully compressed. Let \( Q \in P \) denote the
other one. If \( Q \) overlaps \( R' \) than \( Q \) overlaps \( R \) as well, contradicting \( P \cup \{ R \} \) being fully compressed. The only way we can have \( R' \) overlaps \( Q \) but not \( R \) overlaps \( Q \), is when \( (C_{i+1}, Q) \in \text{overlaps} \). According to the last property in Lemma 5.1.1, this means that also the clauses in \( R \) overlap the clauses in \( Q \). This contradicts the fact that \( P \cup \{ R \} \) is fully compressed.

Now, let us consider the case where \( (C_{i+1}, R) \notin \text{overlaps} \). For this case we will use \( x' = i \). Like for the first case, confirming (1)-(3) is trivial, and we focus on showing that (4) is true.

Assume that (4) does not hold. We know that one of the overlapping sets must be \( R' \), and we let \( Q \in P' \) denote the other. \( P' = P \cup \{ R \} \) and \( (C_{i+1}, R) \notin \text{overlaps} \) implies that \( Q \in P \). Using the three first properties from Lemma 5.1.1, we can show that \( (C_y, R') \notin \text{overlaps} \) for \( 1 \leq y \leq i \). The only possibility left is \( (C_{i+1}, Q) \in \text{overlaps} \), which implies a contradiction as shown above.

So, to summarize the proof, we know that

1. The invariant holds when we enter the loop for the first time.
2. If the invariant holds at the beginning of a pass, it will hold at the end of this pass.
3. The loop will terminate.
4. If the invariant holds when the loop terminates, the algorithm will terminate with a correct answer.

Together, these facts prove that the algorithm always terminates with a correct answer.

### 5.3 Data Structures

We need a data structure to represent the compressed set \( H \leftarrow b_1, \ldots, b_n \). The most obvious way would be to represent \( H \) as a list of length-head pairs but, considering that these pairs will be used in an ascending order with respect to their length-value, a sorted list would be more efficient. To further minimize the work during the iteration we can represent the compressed set by the data structure \([L_1, L_2, \ldots, L_n] \leftarrow b_1, b_2, \ldots, b_n\) where \( L_i \) is a list of all heads \( h \) such that \( (i, h) \in H \).

**Example 5.3.1** The compressed set

\[
\{(1, a(X)), (1, b(X)), (2, c(Y)), (4, d(X, Z))\} \leftarrow e(X), f(X, Y), g(Y, Z), h(Z)
\]

would be represented by

\[
[[a(X), b(X)], [c(Y)], [], [d(X, Z)]] \leftarrow e(X), f(X, Y), g(Y, Z), h(Z)
\]
During the computation of $O^P$ the list and the body can be traversed at the same time, so that after solving the $i$th atom of the body and combining the result with the previous result, the terms in the $i$th list should be used to infer new facts.

Figure 5.5 shows how the algorithm for evaluating compressed sets given in Section 4.2 can be implemented for this data structure. Let $P$ be represented by the structure $[L_1, \ldots, L_n] \leftarrow b_1, \ldots, b_n$. After executing the algorithm, we have $d' = O^P(d)$.

\[
\begin{align*}
  d' &:= \emptyset \\
  e &:= \epsilon \quad \text{(the empty variable binding)} \\
  \text{for } i &:= 1 \text{ to } n \\
  \text{begin} \\
  e &:= e \odot (\Delta (b_i, d)) \\
  d' &:= d' \cup \bigcup_{h \in L_i} \forall (h, e) \\
  \text{end}
\end{align*}
\]

Figure 5.5: Evaluation of a compressed set
Chapter 6

Properties of SCCs in Magic Programs

In Section 3.2 we decided that the best way to combine induced magic with the SCC approach was to build compressed sets of overlapping clauses that define predicates in the same component. Now we will try to justify this decision by showing that this method will avoid much of the recomputation caused by overlapping clauses. We will also look at the effects of the fact that the chosen method avoids recomputation of overlapping clauses that are generated from different clauses in the original program.

6.1 Dependency Loops in Magic Programs

We want to show that, for ordinary programs, the predicates defined by magic clauses generated from a single clause in the original program often end up in the same SCC. This will be done by identifying common structures in the original program that result in recursive SCCs in the magic program. The reason why we focus on recursive SCCs are that non-recursive components are computed without fixed point iteration, and their impact on the overall computation time is significantly smaller than that of recursive SCCs.

Throughout the chapter we will use \( a() \) to denote an atom built of the predicate \( a \) and a correct number of arbitrary terms.

Definition 6.1.1 Given a program \( P \), denote by \( \text{def}(P) \) the set of predicates defined in \( P \). If all predicates occurring in \( P \) are in \( \text{def}(P) \), we say that \( P \) is fully defined.

Lemma 6.1.1 For any predicate \( a \in \text{def}(P) \) the predicate dependency graph of \( \text{magic}(P) \) contains the edge \( \bullet a \rightarrow a^\bullet \).

Proof: The edge is caused by the last magic clause generated by a clause defining \( a \). \qed
Lemma 6.1.2 If $a$ depends on $b$ in $P$, then $\bullet b$ depends on $\bullet a$ and $a^\bullet$ depends on $b^\bullet$ in $M^P$.

Proof: The dependencies in $M^P$ are caused by the magic clauses that are generated by the clauses causing the dependency between $a$ and $b$ in $P$. □

The recursive components in the predicate-dependency graph of the magic program are always caused by one or several loops. The most common cause of such loops is probably recursive clauses in the original program, or clauses that are mutually recursive. Such clauses are the subject of the following theorem.

Theorem 6.1.1 Let $P$ be a program, and let $C \in P$ be a clause

\[ a() \leftarrow b_1(), \ldots, b_n(), c(), d_1(), \ldots, d_m() \]

such that $c$ depends on $a$ in $P$. Then the predicate dependency graph of $M^P$ contains the loops

\[ \bullet a \rightarrow \bullet b_x \rightarrow b_x^\bullet \rightarrow \bullet c \rightarrow \ldots \rightarrow \bullet a \]

for each $b_x \in \text{def}(P)$, $1 \leq x \leq n$ and

\[ a^\bullet \leftarrow d_y^\bullet \leftarrow \bullet d_y \leftarrow c^\bullet \leftarrow \ldots \leftarrow a^\bullet \]

for each $d_y \in \text{def}(P)$, $1 \leq y \leq m$.

Proof: Most of the edges in the loops are directly caused by the magic clauses generated by $C$. The rest of them are explained by Lemma 6.1.1 and 6.1.2. □

Example 6.1.1 Let $P$ be a program containing the clause

\[ \text{length}([X][X], A) \leftarrow \text{length}(X, B), \]
\[ \text{sum}(B, 1, A) \]

and let $\text{sum} \in \text{def}(P)$. Then $M^P$ contains the clauses

\[ \bullet \text{length}(X, B) \leftarrow \]
\[ \bullet \text{length}([X][X], A) \]
\[ \bullet \text{sum}(B, 1, A) \leftarrow \]
\[ \bullet \text{length}([X][X], A), \]
\[ \text{length}^\bullet(X, B) \]
\[ \text{length}^\bullet([X][X], A) \leftarrow \]
\[ \bullet \text{length}([X][X], A), \]
\[ \text{length}^\bullet(X, B), \]
\[ \text{sum}^\bullet(B, 1, A) \]

These magic clauses result in the following edges in the predicate-dependency graph of $M^P$. 32
Together with the edge \( \bullet \text{sum} \rightarrow \text{sum}^\bullet \) from Lemma 6.1.1, we get the loops described in Theorem 6.1.1.

The following corollary shows why Theorem 6.1.1 is of interest to us.

**Corollary 6.1.1** Let \( P \) be a fully defined program, and let \( C \in P \) be a clause

\[
a() \leftarrow \ldots c(), \ldots
\]

such that \( c \) depend on \( a \) in \( P \). Then the magic clauses generated by \( C \) define predicates that are distributed over at most two components in the dependency graph of \( M^P \).

**Proof:** The loops described in Theorem 6.1.1 will result in two SCCs (or possibly a single one). All predicates defined by the magic clauses generated by \( C \) are in one of these SCCs.

This means that recursive clauses will be computed fairly efficiently by our algorithm. If we allow the program to contain undefined predicates, the result is somewhat weaker since undefined predicates are not included in the two SCCs described above. Still we can show that the predicates defined by the magic clauses are distributed over at most two recursive SCCs, since an undefined predicate \( p \) in \( P \) will result in the non-recursive SCCs \( \{p\} \) and \( \{p^\bullet\} \) in the dependency graph of \( M^P \).

We have shown one possible cause of recursive SCCs in the dependency graph, and probably the most common one. An interesting property of the components generated in this way, is that they often contain dependencies on the form \( b^\bullet \rightarrow b^\bullet \). The following theorem explains why this property is interesting.

**Theorem 6.1.2** Let \( b \) be a predicate in a fully defined program \( P \), such that \( b^\bullet \) depends on \( b^\bullet \) in \( M^P \). Also, let \( C \in P \) be a clause

\[
a() \leftarrow a_1(), \ldots, a_n().
\]

such that \( b = a \) or \( b \) depends on \( a \) in \( P \). Then the magic clauses generated by \( C \) define predicates that are all in the same SCC.

**Proof:** The magic clauses generated by \( C \) give us the edges \( \bullet a \rightarrow \bullet a_1 \) and \( a_1^\bullet \rightarrow a^\bullet \), and from Lemma 6.1.1 we get \( \bullet a_1 \rightarrow a_1^\bullet \). If \( b \neq a \), we use Lemma 6.1.2 to show that \( \bullet b \rightarrow \ldots \rightarrow \bullet a \) and \( a^\bullet \rightarrow \ldots \rightarrow b^\bullet \). Now, since \( \bullet b \) depends on \( b^\bullet \), we have a loop containing both \( \bullet a_1 \) and \( a^\bullet \). By doing the same for the rest of
the body atoms in $C$, we have shown that they will end up in the same SCC.

As in the previous theorem, allowing undefined predicates in the program gives us a weaker result, but we can show that the predicates are still distributed over no more than one recursive SCC.

**Example 6.1.2** Traversing a list is a common way of manipulating data in logic programs. This example considers a program that performs a complicated operation on all elements of a list, building a list of the results. Such a program might contain the clauses

\[
\begin{align*}
\text{traverse}([], []) & \quad \text{(1)} \\
\text{traverse}([X\{Xs\}, [Y\{Ys\}]) & \leftarrow \text{compute}(X, Y), \text{traverse}(Xs, Ys) \quad \text{(2)} \\
\text{compute}(X, Y) & \leftarrow \ldots \quad \text{(3)}
\end{align*}
\]

Corollary 6.1.1 tells us that the three magic clauses generated by (2), will be nicely compressed into at most two compressed sets. Using Theorem 6.1.2 we also know that (3) and any clause defining a predicate on which \text{compute} depends, will generate magic clauses that will be compressed into a single compressed set.

We have shown that recursive structures in the original program are treated well by our method, and that they tend to result in efficient treatment of other clauses as well. There are several other ways of introducing dependency loops in the magic program, but they do not seem to be as common in ordinary programs.

**Example 6.1.3** Clauses on the form

\[
\begin{align*}
a() & \leftarrow \ldots, p(), \ldots, q(), \ldots \\
b() & \leftarrow \ldots, q(), \ldots, p(), \ldots
\end{align*}
\]

result in the loop

\[
\ast p \rightarrow p^* \rightarrow \ast q \rightarrow q^* \rightarrow p
\]

provided that $p$ and $q$ are defined in the program. This example shows that we might have loops in the dependency graph of $M^P$ even when there is no loops in the dependency graph of $P$.

### 6.2 Overlapping Modulo Variable Renaming

Considering clauses that overlap modulo variable renaming, makes it possible to find overlapping magic clauses that are derived from different clauses in the original program. This might seem like a rather important improvement, but we will show that this is not the case. Looking at a clause $C$ with head $h$, we notice that the first atom of the body of all magic clauses derived from $C$ is $\ast h$. So the only way two magic clauses derived from the clauses $C_1$ and $C_2$ can overlap is if the heads of $C_1$ and $C_2$ are identical modulo variable renaming.
**Example 6.2.1** The program

\[
\begin{align*}
a(X) & \leftarrow b(X) \\
a(Y) & \leftarrow c(Y) \\
b([[]]) & \leftarrow c(X), b(X) \\
c(1) & \leftarrow \end{align*}
\]

is transformed into

\[
\begin{align*}
* b(X) & \leftarrow * a(X) \\
* c(X) & \leftarrow * a(X), b(X) \\
* c(Y) & \leftarrow * a(Y) \\
* * ([]) & \leftarrow * b([[]]) \\
* c(1) & \leftarrow * b([X|Xs]) \\
* b([X|Xs]) & \leftarrow * b([X|Xs]), c(X) \\
* c([X|Xs]) & \leftarrow * b([X|Xs]), c(X), b(X) \\
* c(1) & \leftarrow * c(1)
\end{align*}
\]

with the SCC sequence

\[
\{* a\} \rightarrow \{* b, * c, * b\} \rightarrow \{* b\} \rightarrow \{* a\}
\]

Looking closer at the clauses defining the predicates in the second SCC (1, 3, 6 and 9), we find that (1) and (3) have overlapping bodies even though they originate from different clauses.

In ordinary programs, identical heads are not very common. With the additional requirement that the generated magic clauses should define predicates in the same SCC, the occurrences are ever fewer. If we for example change the fourth clause in the example into \(b([X|Xs]) \leftarrow d(X), b(Xs)\) (maybe adding the clause \(d(1)\) as well), the magic clauses (1) and (3) no longer define predicates in the same SCC, and will not be computed together.
Chapter 7

Regular Approximations, an Application

Computing regular approximations is one application area for bottom-up evaluation, and this chapter presents a brief introduction to the subject as well as a description of how it can be viewed as an instance of our general bottom-up framework described in Section 2.3. The uses of regular approximation include compile-time optimization, program specialization and static analysis for debugging.

In Chapter 2.3 we showed an example of how bottom-up evaluation can be used to compute types for the predicates of a logic program. We used specifications like \( \{ q(\text{int}) \} \) to describe that the argument of the predicate \( q \) is restricted to values in the set \( \text{int} \). This idea will now be extended, and instead of using a predefined set of types, we will construct types during the bottom-up evaluation.

We restrict the types to sets that can be described by regular term grammars. The reason for this is that set operations such as intersection and membership test can be implemented very efficiently for these sets. Another reason is the possibility of representing infinite sets using a finite set of grammar rules.

A specification like the one described above, can be viewed as an approximation, or superset, of the least Herbrand model of the program. This suggests that we might use bottom-up evaluation to compute specifications, and in each step approximate the operations. Before explaining this further, we need to define some basic concepts.

7.1 Regular Term Grammars

Our grammars are built over an alphabet \( T \cup F \), where \( T \) is a set of type symbols, and \( F \) contains the functors and predicates from the object language (i.e. the logic programming language used). Constants are regarded as functors of arity 0.
Definition 7.1.1 We consider grammar rules on the form
\[ t_0 \rightarrow f(t_1, \ldots, t_k) \]
where \( t_i \in T \) and \( f \in F \) is of arity \( k \geq 0 \).

Definition 7.1.2 A regular term grammar is a finite set of grammar rules satisfying the condition that if it contains the distinct rules
\[
\begin{align*}
t & \rightarrow f(\ldots) \\
t & \rightarrow f'(\ldots)
\end{align*}
\]
then \( f \neq f' \).

Example 7.1.1 The following is an example of a regular term grammar
\[
\begin{align*}
\text{bin} & \rightarrow 0 \\
\text{bin} & \rightarrow 1 \\
\text{binlist} & \rightarrow [] \\
\text{binlist} & \rightarrow [\text{bin} \mid \text{binlist}]
\end{align*}
\]
The reason for introducing regular term grammars is that they can be used to represent regular sets, and the following definition shows how this is done.

Definition 7.1.3 For a regular term grammar \( G \) and a type symbol \( t \), we define
\[
\mathbb{E}^G[t] := \{ u \mid t \xrightarrow{G} u \land u \text{ contain no type symbols} \}
\]
where \( \xrightarrow{G} \) is the transitive closure of the rewriting relation introduced by \( G \).

Example 7.1.2 For the grammar \( G \) from the previous example we have
\[
\begin{align*}
\mathbb{E}[\text{bin}] &= \{0, 1\} \\
\mathbb{E}[\text{binlist}] &= \{[], [0], [1], [0, 0], [0, 1], [1, 0], [1, 1], [0, 0, 0], \ldots\}
\end{align*}
\]
The advantage of using regular term grammars to represent sets is the possibility of computing set operations in an efficient way. We will define grammar intersection and upper bound, and show how they are used to implement set operations. For definitions of set inclusion, emptiness and membership, see [Pie98].

Definition 7.1.4 Let \( G_1 \) and \( G_2 \) be two regular term grammars. The intersection of them, denoted \( G_1 \cap G_2 \), is defined like this:
\[
G_1 \cap G_2 = \big\{ t_1 \cap t_2 \rightarrow f(r_1 \cap s_1, \ldots, r_n \cap s_n) \big\} \\
\quad t_1 \rightarrow f(r_1, \ldots, r_n) \in G_1 \land \\
\quad t_2 \rightarrow f(s_1, \ldots, s_n) \in G_2
\]

Example 7.1.3 Let \( G \) be the regular term grammar from the previous example, and \( G_2 \) be
\[
\begin{align*}
dig & \rightarrow 1 \\
dig & \rightarrow 2 \\
dlist & \rightarrow [] \\
dlist & \rightarrow [\text{dig} \mid \text{dlist}]
\end{align*}
\]
Then $G \cap G_2$ is

- $bin \cap dig \rightarrow 1$
- $binlist \cap dlist \rightarrow []$
- $binlist \cap dlist \rightarrow [bin \cap dig \mid binlist \cap dlist]$

**Definition 7.1.5** Let $G_1$ and $G_2$ be two regular term grammars. Their upper bound, denoted $G_1 \cup G_2$, is a regular term grammar defined like this:

1. For each pair of rules
   
   - $t_1 \rightarrow f(r_1, \ldots, r_n) \in G_1$
   - $t_2 \rightarrow f(s_1, \ldots, s_n) \in G_2$

   $G_1 \cup G_2$ contains the rule
   
   - $t_1 \cup t_2 \rightarrow f(r_1 \cup s_1, \ldots, r_n \cup s_n)$

2. For each pair of type symbols $t_i, t_j$ in $G_i$ and $G_j$ respectively (where $\{i, j\} = \{1, 2\}$), such that
   
   - $t_i \rightarrow f(r_1, \ldots, r_n) \in G_1$

   but $G_j$ does not contain a rule on the form
   
   - $t_j \rightarrow f(s_1, \ldots, s_n)$

   $G_1 \cup G_2$ contains the rule
   
   - $t_1 \cup t_2 \rightarrow f(r_1, \ldots, r_n)$

   and all rules in $G_i$ that can be used in a rewriting starting from one of $r_1, \ldots, r_n$.

**Example 7.1.4** Take $G$ and $G_2$ from the previous example. The upper bound $G \cup G_2$ is

- $bin \cup dig \rightarrow 0$
- $binlist \cup dlist \rightarrow []$
- $binlist \cup dlist \rightarrow [bin \cup dig \mid binlist \cup dlist]$
- $bin \cup dig \rightarrow 2$

**Proposition 7.1.1** For the regular term grammars $G_1$ and $G_2$, we have

- $[t_1]^{G_1} \cap [t_2]^{G_2} = [t_1 \cap t_2]^{G_1 \cap G_2}$
- $[t_1]^{G_1} \cup [t_2]^{G_2} \subseteq [t_1 \cup t_2]^{G_1 \cup G_2}$

This means that if we use grammars to represent regular sets, we can compute set intersection using grammar intersection. Similarly, union of sets can be approximated by the upper bound relation. The reason why set union can not be computed exactly is that the domain of regular sets is not closed under union.

**Example 7.1.5** Consider the regular sets

- $S_1 = \{a(1,1)\}$
- $S_2 = \{a(2,2)\}$

The set $S_1 \cup S_2$ is not regular, but can be approximated by the regular set

- $S_1 \cup S_2 \subseteq \{a(1,1), a(1,2), a(2,1), a(2,2)\}$

39
7.2 Approximation as Bottom-up Evaluation

Our task is to define an operator \( A^C \) that is an approximation of the \( T^C \) operator, because then using this operator we would compute an approximation of the least Herbrand model of the program.

Since we want to be able to use the compressed magic method when computing the operation, we will follow Section 2.3 and define \( A^C \) as a composition of three operations. We start by defining the two domains \( D \) and \( E \). As when computing least Herbrand models, \( D \) is the powerset of all ground atomic formulas over the alphabet used in the program, but here we also require that each set in \( D \) can be described by a regular term grammar. \( E \) contains all possible mappings from variables to regular sets. Now we can define the three operations \( \Delta, \oplus \) and \( \forall \).

**Definition 7.2.1** With \( D \) and \( E \) defined as described above, we define \( \Delta(t, d) \) as follows. For each variable \( X \) in \( t \), let \( Y_1, \ldots, Y_n \) be the occurrences of \( X \) in \( t \).

In the resulting variable binding, let \( X \) be bound to the set \( \bigcap_{i=1}^n \{ a \mid t_{Y_i} \in d \} \), where \( t_{Y_i} \) is the result of replacing \( Y_i \) by \( a \) in \( t \).

**Example 7.2.1** With \( t = a(X, X) \) and \( d = \{ a(1, 2), a(1, 3), a(2, 2), a(2, 3) \} \) we would have \( \Delta(t, d) = \{ X/\{1, 2, 3 \} \} \).

**Definition 7.2.2** We define \( \oplus(e_1, e_2) = e_3 \) where \( e_3 \) contains the bindings

\[
X/(s_1 \cap s_2) \in e_3 \quad \text{iff} \quad X/s_1 \in e_1 \land X/s_2 \in e_2 \\
X/s_1 \in e_3 \quad \text{iff} \quad X/s_1 \in e_1 \land X/s_2 \notin e_2 \\
X/s_2 \in e_3 \quad \text{iff} \quad X/s_1 \notin e_1 \land X/s_2 \in e_2
\]

**Example 7.2.2** Let \( e_1 = \{ W/\{a, b\}, Y/\{b\} \} \) and \( e_2 = \{ W/\{b, c\}, Z/\{d\} \} \), and we have \( \oplus(e_1, e_2) = \{ W/\{b\}, Y/\{b\}, Z/\{d\} \} \).

**Definition 7.2.3** Let \( X_1 \ldots X_n \) be the variables occuring in a term \( t \), and let \( S_i \) be a set such that \( X_i/S_i \in e \) (if \( e \) does not contain a mapping from \( X_i \) then \( S_i \) is the empty set). We define \( \forall(t, e) \) as

\[
\forall(t, e) = \{ t' \mid t' \text{ is the result of replacing each occurrence of } X_i \text{ in } t \\
\text{with an element from } S_i, \text{ for } 1 \leq n \leq n. \}
\]

**Example 7.2.3** Let \( t = a(X, X, Y) \) and \( e = \{ X/\{1, 2\}, Y/\{3\} \} \), and we have \( \forall(t, e) = \{ a(1, 1, 3), a(1, 2, 3), a(2, 1, 3), a(2, 2, 3) \} \).

Since we require that it should be possible to represent the resulting set using a regular term grammar, we loose the possibility of specifying that the first two arguments of \( t \) in the example above should always be the same.

Having defined the three operations we define \( A^C \) like we defined the general \( O^C \) operation in Section 2.3.

\[
A^C(d) = \forall(h, \bigoplus_{i=1}^n (\Delta (h, d)))
\]
We also define $A^P$ as
\[ A^P(d) = \bigcup_{C \in \mathcal{E}} A^C(d) \]

If we use regular grammars to represent the elements of $D$ and $E$, we will probably use the approximation of set union presented in the previous section when computing $A^P$. The result is that we would in fact not compute $A^P$ as defined above, but rather $A^{IP}$ such that $A^P \subseteq A^{IP}$.

By defining regular approximation as an instance of the general bottom-up evaluation method, we know that we can use all the improvements discussed in this report when approximating programs. Especially, when we want to approximate the call-success semantics of a program (that is, when we apply the magic transformation and approximate the resulting magic program), we might use the compressed magic approach.

### 7.3 Avoiding Non-termination

The requirements for termination of bottom-up evaluation, described in Section 2.1, are not satisfied by general logic programs in the case of above discussed approximation. To ensure termination, we use a normalization\(^1\) operation after each step of the iteration. The actual definition of this operation falls outside the scope of this report, but the following example illustrates the idea behind the normalization operation. The issue of normalization is discussed in [Pic98, Mil99].

**Example 7.3.1** Without a normalization function, the approximation of the program
\[
\text{alist}([], X) \leftarrow \text{alist}(X)
\]
would not terminate since each iteration step just adds another element to the infinite set
\[
\{\text{alist}([], X), \text{alist}([a], X), \text{alist}([a, a], X), \text{alist}([a, a, a], X), \ldots\}
\]
The goal of the normalization operation is to find structures that can be approximated with a recursive construction. Assume that the result of the third iteration step is represented by $[[t]]^G$, with $G$ being the grammar
\[
\begin{align*}
t_1 & \rightarrow [] \\
t_2 & \rightarrow [t_2|t_3] \\
t_3 & \rightarrow t_4 | t_3 \\
t_3 & \rightarrow [t_4|t_5] \\
t_5 & \rightarrow [a|a]
\end{align*}
\]
Applying the normalization operation would result in the grammar $G'$
\[
\begin{align*}
t_1 & \rightarrow [] \\
t_1 & \rightarrow [t_2|t_1] \\
t_2 & \rightarrow a
\end{align*}
\]
\(^1\)This operation is sometimes referred to as widening or shortening.
This approximation is allowed, since \([t_i]^{G'} \subseteq [t_i]^{G''}\). The result of the next iteration step will be normalized into the same \(G''\), and the iteration terminates.

### 7.4 Benchmarks

The compressed magic method has been implemented as an addition to an existing system, making it easy to compare its efficiency against the unchanged version. A description of the system can be found in [Pic98].

The part of the system that we are interested in computes regular approximations of magic programs using the SCC method. It also uses a variant of the semi-naive algorithm that is different from the one presented in this report. We stated that at least one fact used when solving the body should have been inferred in the very previous iteration step. The version used in the system uses the relaxed requirement that at least one predicate of the body should be the predicate of a fact inferred in the very previous step.

An outline of the original and the new version of the system is presented in Figure 7.1. As the figure shows, the changes only affected a small part of the system. Also, many subcomponents within the changed sections were reused in the new version.

![System overview](image)

**Figure 7.1: System overview.**

The input to the system is a program \(P\). A magic transformation and SCC computation result in the magic program \(M^P\) and the SCC sequence \(S\). The dashed box represents a recursion over the SCCs \(K_1, \ldots, K_n\) in \(S\). Each iteration step starts with finding the subprogram \(P' \subseteq M^P\) that defines the predicate in the current SCC. Then \(O^P\) is computed iteratively until the result is a fixed point. In the new system, \(P^d\) is compressed into \(R\), and the new version of the evaluation algorithm is used during the iteration.
Table 7.1 shows the compression rate of the benchmark programs. The second and third column contain the number of clauses in the program and the number of recursive SCCs in the predicate-dependency graph of the magic program. The remaining columns show the number of predicates in the largest SCC, the number of magic clauses and the number of compressed sets. The compression rate is defined as

$$\text{compression rate} = \frac{\text{magic clauses} - \text{compressed sets}}{\text{magic clauses}}$$

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Clauses</th>
<th>Rec. SCCs</th>
<th>Largest SCC</th>
<th>Magic clauses</th>
<th>Compr. sets</th>
<th>Compr. rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>a9</td>
<td>932</td>
<td>78</td>
<td>613</td>
<td>3634</td>
<td>2568</td>
<td>29%</td>
</tr>
<tr>
<td>chat5</td>
<td>378</td>
<td>11</td>
<td>286</td>
<td>794</td>
<td>463</td>
<td>42%</td>
</tr>
<tr>
<td>eliza</td>
<td>173</td>
<td>16</td>
<td>67</td>
<td>456</td>
<td>363</td>
<td>20%</td>
</tr>
<tr>
<td>analyzer</td>
<td>136</td>
<td>12</td>
<td>116</td>
<td>423</td>
<td>287</td>
<td>32%</td>
</tr>
<tr>
<td>exdora</td>
<td>81</td>
<td>32</td>
<td>68</td>
<td>746</td>
<td>674</td>
<td>10%</td>
</tr>
</tbody>
</table>

Table 7.1: Compression rates

These results show that the compression rates vary a lot. This was expected since the compression rate depends highly on the structure of the program. The high compression rates support the discussion in Section 6.1, where we argued that many overlapping clauses will end up in the same SCC.

Table 7.2 presents the computation time\(^2\) for the programs, comparing the original implementation with the new one. The first column shows the compression rate from Table 7.1. The last column contains the improvement rate, calculated similarly to the compression rate.

$$\text{improvement rate} = \frac{\text{original time} - \text{new time}}{\text{original time}}$$

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Compr.</th>
<th>Orig. Time</th>
<th>New Time</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>a9</td>
<td>29%</td>
<td>135 s</td>
<td>51 s</td>
<td>62%</td>
</tr>
<tr>
<td>chat5</td>
<td>42%</td>
<td>53 s</td>
<td>41 s</td>
<td>24%</td>
</tr>
<tr>
<td>eliza</td>
<td>20%</td>
<td>4.9 s</td>
<td>4.5 s</td>
<td>9%</td>
</tr>
<tr>
<td>analyzer</td>
<td>32%</td>
<td>4.0 s</td>
<td>2.6 s</td>
<td>34%</td>
</tr>
<tr>
<td>exdora</td>
<td>16%</td>
<td>9.9 s</td>
<td>9.4 s</td>
<td>6%</td>
</tr>
</tbody>
</table>

Table 7.2: Improvement rates

The computation time does not include the magic transformation and the SCC computation, since they are identical in both versions. The parts that are included correspond to those in the dashed box in Figure 7.1.

\(^2\)The test was performed on a SUNW, Ultra 10.
Of course, testing five programs is not enough to draw any conclusions regarding the average improvement obtained by the new method. The result tells us, however, that we get a very high improvement rate for some programs.
Chapter 8

Conclusions

Using a general framework, we have presented a bottom-up evaluation technique that combines the basic ideas of induced magic and the SCC method. We have also described an implementation of the method, where the magic clauses are sorted before we identify those that overlap. Finally, we described a system that uses an instance of the bottom-up evaluation framework to infer regular types for logic programs. We showed how the new method was incorporated into this system, and that it indeed made the system run faster for our benchmark programs.

This chapter discusses the efficiency of the proposed method, and presents a few suggestions of future work.

8.1 Efficiency Discussion

A worst-case analysis of our method shows no improvement compared to the original SCC approach. In the worst case, no overlapping occur in the components and the only difference between the methods is the overhead cost introduced by the search for overlapping clauses.

In Section 6.1 we showed that recursive structures in the original program, as well as every clause they depend on, are treated well by our method. This result is important since recursive structures are the most common cause of recursive components, and thus have a great influence on the overall computation time.

So, although we get no improvement in the worst case, it seems like the evaluation of an average program would spend much time in components that does contain many overlapping clauses. This conjecture is supported by the benchmark data presented in the previous chapter.

Finally, it should be noted that the time complexity of the compression algorithm is fairly low compared to that of the whole bottom-up evaluation. The algorithm presented in Chapter 5 runs in $O(n \log n)$, where $n$ is the number of clauses considered. It is known that bottom-up evaluation has a bad time complexity behaviour in the average case, mainly because the computation time
depends on the number of iterations needed to find a fixed point.

8.2 Future Research

The new method has been compared against the original SCC approach, but we have not compared it against induced magic. Such a comparison would have required a whole new implementation, which was not possible to accomplish within the time limits of the project. A possible future task is performing a full scale testing, where the new method is compared against other optimizations for a wide range of programs and for different instances of bottom-up evaluation.

Another interesting suggestion is to look into the possibility of further optimization. Incorporating the method into a system brought the idea that it would be better to look at the whole system together, including transformation and SCC computation. It should be possible to optimize these parts to make the compression easier. The rest of this chapter will give a rough description of this idea, leaving the formal description as a possible area of future work.

8.3 Optimized Compressed Magic

The type inference system, as described in Figure 7.1, is divided into several subsystems. Many of them perform well-known tasks such as applying the magic transformation to a program, or computing the SCCs of a graph.

The advantage of this approach is that it allows us to concentrate on solving the subproblems that are specific for this application. However, this division technique is probably not the most efficient way to solve the whole problem, and this section sketches another solution.

Looking closer at the parts of the system, we notice that we actually know a lot about overlapping clauses when the magic program is generated. We know that each magic clause overlaps all shorter clauses generated from the same original clause. Since this knowledge is not included in the magic program, we later have to perform a complicated search for overlapping clauses when compressing the program. The compression would be much easier if we could somehow keep the knowledge of overlapping clauses during the whole process.

The predicate-dependency graph of the magic program could be calculated directly from the original program. At the same time, each edge could be marked with a reference to the magic clauses that caused it. We never generate the magic clauses, but refer to them implicitly using $x, y$ to represent the $y$:th magic clause generated by the $x$:th clause in the original program.

Example 8.3.1 Let the first clause in the original program be

$$a(X, Z) \leftarrow b(X, Y), c(Y, Z)$$

This clause contributes with the following vertices and edges to the predicate-dependency graph of the magic program

46
The labels represent the clauses that would be generated from the clause if we applied the magic transformation. These clauses are
\[
\begin{align*}
\bullet \ b(X, Y) & \leftarrow \bullet a(X, Z) & (1.1) \\
\bullet \ c(Y, Z) & \leftarrow \bullet a(X, Z), b(X, Y) & (1.2) \\
\bullet \ a(X, Z) & \leftarrow \bullet a(X, Z), b(X, Y), c(Y, Z) & (1.3)
\end{align*}
\]

When the SCCs of the dependency graph are computed, each SCC is associated with a set containing the labels of all edges going to a predicate in the component. Such a set corresponds to the magic clauses that define the predicates of that SCC.

There are two reasons for constructing these sets. The compression algorithm can be made much easier if the origin of each magic clause is known. Also, during the iteration over a component we need not gather the clauses defining the predicates in the component. A minor drawback of the method is that we are not able to find overlapping clauses generated by different original clauses but, as discussed in Section 6.2, this does not happen very often.

**Example 8.3.2** This example illustrates the whole procedure. Let \( P \) be the following program

\[
\begin{align*}
a([[]]) & \leftarrow b(X), a(X) & (1) \\
a([X][Xs]) & \leftarrow b(X), a(Xs) & (2) \\
b(1) & \leftarrow & (3)
\end{align*}
\]

The decorated predicate-dependency graph of \( M^P \), computed directly from \( P \), would look like this

We compute the SCCs of the graph, collecting the labels as described above.

\[
\{ \bullet a, \bullet b, \bullet c \} \leftarrow \{ a^* \}
\]

\[
\begin{array}{cccc}
2.1 & 1.1 \\
2.2 & 2.3 \\
3.1
\end{array}
\]
The representation of magic clauses makes compression easy. For each component, we simply merge all labels that have identical left-side values.

\[
\{ \bullet a, \bullet b, \bullet \} \rightarrow \{ a \}
\]

\[
\begin{array}{ll}
2.[1, 2] & 1.[1] \\
3.[1] & 2.[3]
\end{array}
\]

Without explicitly generating \( M^P \), we have constructed a representation of the compressed sets that we need for the iteration. As a comparison, computing \( M^P \) would have resulted in the following magic clauses:

\[
\begin{align*}
a^*([]) & \leftarrow a([]) & (1.1) \\
b^*(X) & \leftarrow a([X[Xs]]) & (2.1) \\
a^*(Xs) & \leftarrow a([X[Xs]], b(X)^*) & (2.2) \\
a^*([X[Xs]]) & \leftarrow a([X[Xs]], b(X)^*, a(Xs)^*) & (2.3) \\
b^*(1) & \leftarrow b(1) & (3.1)
\end{align*}
\]

When iterating over a component, all we need is its set of labels and the original program. Figure 5.5 sketches the algorithm for computing \( O^P(d) \), where \( P' \subseteq M^P \) is represented by the compressed label \( x.Y \), and the \( x \)-th clause of \( P \) is \( h \leftarrow b_1, \ldots, b_n \).

We make the algorithm simpler by letting \( b_{n+1} \) denote \( h^* \), and \( b_n^* \) denote \( h \). The reason for this new notation should be obvious when we look at the magic clauses generated from the \( x \)-th clause.

<table>
<thead>
<tr>
<th>Old notation</th>
<th>New notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bullet b_1 \leftarrow \bullet h )</td>
<td>( \bullet b_1 \leftarrow b_{0}^* )</td>
</tr>
<tr>
<td>( \bullet b_2 \leftarrow \bullet h, b_1^* )</td>
<td>( \bullet b_2 \leftarrow b_{0}^<em>, b_1^</em> )</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>( h^* \leftarrow \bullet h^<em>, b_1^</em>, \ldots, b_n^* )</td>
<td>( b_{n+1} \leftarrow b_{0}^<em>, b_1^</em>, \ldots, b_n^* )</td>
</tr>
</tbody>
</table>

\[
d' := \{ \} \\
e := e \\
\text{for } i := 1 \text{ to } \text{max}(Y) \text{ begin} \\
\quad e := e \ominus (\Delta (b_{i-1}^*, d)) \\
\quad \text{if } i \in Y \text{ then } d' := d' \cup \forall(b_i, e) \\
\text{end}
\]

Figure 8.1: Optimized evaluation of a compressed set
Bibliography


Index

approximation, 32
atom, 4

benchmarks, 37
bottom-up evaluation, 5

clause, 4
comparing clauses, 21
complete lattice, 3
compressed set, 18
compression rate, 38

data structures, 25
definite
  clause, 4
goal, 4
  logic program, 4
dependency loops, 27
dynamic programming, 13

fixed point, 3
fully defined program, 27

goal, 4
grammar
  intersection, 33
  upper bound, 34

improvement rate, 38
induced magic, 12

least Herbrand model, 5, 6
logic program, 4
loops, 27

magic transformation, 11

naive algorithm, 8

normalization, 36

optimized compressed magic, 40
overlapping
  clauses, 17
  compressed sets, 18

predicate-dependency graph, 9

regular approximation, 32
regular term grammar, 33

semi-naïve algorithm, 8
sorting clauses, 21
strongly connected components, 9
system overview, 37
types in logic programs, 7, 32