Using Horn Clauses and Binary Decision Diagrams for Program Analysis

A Dissertation Presented

by

Wenxin Song

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

August 2010

Stony Brook University

The Graduate School

Wenxin Song

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation.

Eugene Stark, Dissertation Advisor
Professor, Department of Computer Science

Radu Grosu, Chairperson of Defense
Professor, Department of Computer Science

Scott Stoller
Professor, Department of Computer Science

Aarti Gupta
Department Head, NEC Laboratories America

This dissertation is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Using Horn Clauses and Binary Decision Diagrams for Program Analysis

by

Wenxin Song

Doctor of Philosophy

in

Computer Science

Stony Brook University

2010

Logic programming languages have been widely used to express program analyses. In this dissertation, we present a framework of program analysis using *Horn clauses* and *Binary Decision Diagrams* (BDDs). In contrast to previous work of Whaley, *et al.* that used Datalog and BDDs for program analysis, we use Horn clauses (which can be viewed as an extension of Datalog) to express program analysis problems. Horn clauses are more expressive than Datalog by allowing functions as arguments of predicates. Many type-based program analysis problems involve type information in which the type is usually a recursively defined object consisting of subtypes. The use of function symbols makes it possible to specify a type in a natural way.

BDDs are very popular tool in hardware verification and model checking. Recently, BDDs have also been used in program analysis to efficiently analyze large programs. Unlike Datalog, Horn clauses cannot be implemented by using ordinary BDDs due to the presence of functions. In this dissertation, we propose an automata-based approach that treats terms as strings reading from left to right and represents terms by automata. We devise various operations on automata to manipulate terms. Moreover, we show that such automata lend themselves readily to a representation using *Multi-Terminal Binary Decision Diagrams* (MTBDDs).

We present a top-down set-at-a-time depth-first evaluation algorithm for Horn clauses in terms of sets of ground atoms. Such evaluation algorithm computes relevant results starting from the query in a top-down fashion, operates over a set of atoms at a time, and gains efficiency by taking the advantage of symbolic representation of sets of ground atoms in Horn clauses.

The combination of the above ideas yields a framework for program analysis such that analysis queries are naturally expressed as Horn clauses and the evaluation of Horn clauses is efficiently implemented by MTBDDs. Such a framework is suitable for type-based program analysis, such as strictness analysis, binding-time analysis, secure information flow analysis, and so on.

This dissertation is dedicated to my wife, Xuefang, and my daughter, Sabrina

Contents

List of Figures

Acknowledgements

- First of all, I would like to express my deepest gratitude to my advisor, Professor Eugene Stark, for teaching me so much and inspiring me to solve challenging problems. I would never have been able to finish my dissertation without his guidance. The most important thing I have learned from him is to be open-minded when finding solutions and to be extremely precise when writing technical reports.
- I also want to thank my dissertation committee members, Professor Radu Grosu, Professor Scott Stoller and Dr. Aarti Gupta, for their feedbacks and comments.
- I gratefully acknowledge the help from many colleagues of Stony Brook computer science department. Diptikalyan Saha helped me to understand the tabled evaluation of logic programs and Rahul Agarwal introduced me some basic concepts of type-based program analysis. I also want to thank useful discussions from Xiaowan Huang and Wenkai Tan.
- Finally, I cannot finish without saying “thanks” to my wife and parents, who were always supporting me and encouraging me during these years.

Chapter 1

Introduction

1.1 Motivation

Program analysis is a process of analyzing computer programs and gathering information to optimize executable codes via compilers or to ensure that certain properties are satisfied by those programs. This process can also be used to find program bugs and software vulnerabilities. The National Institute of Standards and Technology conducted a study in 2002 to find out that software bugs cost the US economy about $59.5 billion annually [1]. The same study also pointed out that more than a third of that cost, about $22.2 billion, could be eliminated by an improved infrastructure that includes static program analysis tools [1].

Although static analysis is very useful in detecting program bugs, it is not easy to design and implement a static analysis on a real program. Static analysis algorithms are often complicated. In order to achieve good results, the user has to be an expert of both those algorithms and the target languages that are analyzed. An implementation of a static analysis in some traditional programming language often takes hundreds or even thousands of lines of code. Making changes to such implementation is time-consuming and error-prone.

In order to make it easier to design and develop advanced program analyses, some researchers have recently used logic programming languages to specify static analysis problems [17, 44, 58]. In such approach, the source code of a program is stored as a set of tuples in a relational database and the analysis is expressed as a set of deductive rules. Then the analysis problem is solved by evaluating a query along with the deductive rules and the database. For example, consider the following fragment of Java code:

$String\ a = "fido";$

$String\ b;$

$b = a;$

The tuples generated from the code are $vP_0(a, h_1)$ and $assign(b, a)$. The former corresponds to the first statement. It says that variable a points to heap object h_1 when it is declared and initialized. There is no corresponding tuple for the second statement since it is just a declaration. The latter is corresponding to the third statement and it says that there is an assignment from a to b. Suppose we want to find out all the points-to relations in this piece of code. We specify "might point to" relation vP by the following rules:

$vP(v, h) \ \leftarrow\ vP_0(v, h).$

$vP(v_1, h) \ \leftarrow\ assign(v_1, v_2), vP(v_2, h).$

The rule $A \ \leftarrow\ B_1, \ldots, B_n$ is interpreted as: if B_1 and … and B_n all hold then A holds. The first rule stores the initial points-to relations into vP. The transitive closure of vP is computed by the second rule such that if v_2 is assigned to v_1 and v_2 may point to heap object h, then v_1 may also point to h. Now we run the query $vP(X, Y)$ along with the rules and the tuples. Since $vP_0(v, h)$ holds, $vP(a, h_1)$ also holds by following the first rule. Since $assign(b, a)$ and $vP(a, h_1)$ hold, $vP(b, h_1)$ holds by following the second rule. Thus, the results are tuples $vP(a, h_1)$ and $vP(b, h_1)$.

Using logic programs to specify program analyses has many advantages. First of all, logic programming languages, such as Prolog and Datalog, are simple and easy to understand. Program analyses expressed as logic programs take only a few lines of code. Second of all, for logic programs, the implementation is handled by the evaluation system of the language. Therefore, the analysis specifiers do not have to worry about the implementation of the analysis in this case. Finally, there are many implementations of logic programing languages, which can be used directly as the implementations of program analyses.

1.2 Challenges

Although logic programming simplifies the work required to design and implement program analyses, it is still difficult to build general and practical static analysis tools that use logic programming languages.

Many researchers use Datalog to express program analysis. Since Datalog does not allow functions, it can have difficulty in expressing analysis problems that involve recursively defined objects. Here is an example of how functions can be used to define recursive objects. Suppose we have the following type inference rule (for any typed language): if expression y has type t_1 and there is a function application $x\,y$ such that function x is applied to argument y and x has function type $t_1 \rightarrow t_2$ (given an input of type t_1, produce an output with type t_2), then application $x\,y$ has the type t_2. This rule can be expressed naturally by a Horn clause with the help of functions:

$$type(A,T_2) \leftarrow application(A,X,Y), type(X,arrow(T_1,T_2)), type(Y,T_1).$$

In this rule, atom $type(X,T)$ says that expression X has type T and the atom $application(A,X,Y)$ says that expression A is an application $X\ Y$. Here, we use function symbol *arrow* to indicate the function type $T_1 \rightarrow T_2$.

The implementations of program analysis that use logic programming systems are often slower than the ones using traditional languages and are difficult to scale to large programs. Reps found that, for on-demand inter-procedural reaching definitions analysis, a native C implementation is six times faster than the implementation using Corel (which is a general-purpose logic programming system) [44]. Some researchers have used XSB, which is a state-of-the-art implementation of logic programs, to perform program analysis [17]. However, those analyses are only performed on programs with several hundred lines. Moreover, it has been pointed out that XSB uses "tuple-at-a-time" evaluation strategy and is not efficient to access large set of tuples [20]. An alternative of tuple-at-a-time is "set-at-a-time" strategy. Recently, a BDD-based implementation of Datalog has been successfully used to do analyses on large programs that have over 10^{14} contexts [58]. This implementation employs *set-at-a-time* strategy, which computes a set of tuples at a time. However, it has two issues. One is the limited expressiveness of Datalog, which has been addressed before. Another is its bottom-up evaluation strategy, which starts from the initial tuples and repeatedly applies old tuples to rules to obtain new tuples until a fixed point is reached. This strategy can avoid infinite loops by recognizing cycles of queries. However, bottom-up strategy cannot handle functions (a set of ground tuples that contain functions is an infinite set) and is inefficient for answering queries because queries are solved only after all the tuples have been computed and many possibly irrelevant tuples are generated. Comparing to bottom-up, top-down is a strategy that starts from the query and translates it to sub-query (by following the rules) until the initial tuples are reached. The solution of the query is obtained by composing the solutions from the sub-queries. A top-down strategy is efficient to answer queries because it only computes relevant tuples.

One might ask whether we can have an efficient implementation of Horn clauses that employs both top-down and set-at-a-time strategies. If that is the case, then we would have a framework for program analysis that possesses the expressiveness of Horn clauses and is able to scale to large programs. Unfortunately, there is no easy solution to this question. Unlike Datalog, Horn clauses cannot be implemented by using traditional BDDs because of the presence of function symbols. The set of tuples in a Datalog program is a finite set. Thus, each tuple can be encoded by a fixed length of bits. Suppose a Datalog program has only four constants $\{a,b,c,d\}$. Then we can use two bits to encode each constant. Any tuple that contains k elements is then encoded by $2k$ bits. We cannot do the same to Horn clauses. Suppose a Horn clause program has only four symbols $\{f,a,b,c\}$ such that a,b and c are constants and f is a function symbol whose arity is 1. We can also use two bits to encode each symbol, but we cannot use fixed length of bits to encode the tuples that have the same arity. For example, suppose the arity of predicate p is 2, then in this case we would have tuples like $p(a,b), p(a,f(b)), p(a,f(f(b))), \ldots$. Clearly, it is impossible to encode those tuples with fixed length of bits.

1.3 Our Solution

This dissertation provides a solution that leads to an efficient implementation of Horn clauses that employs both top-down and set-at-a-time strategies. We propose an automata-based approach to represent a set of tuples as an automaton. Various operations are devised to implement the top-down set-at-a-time evaluation algorithm on automata. More importantly, such automata lend themselves readily to a representation using MTBDDs. Thus, the solution proposed by this dissertation answers the question being asked before, we can have an efficient BDD-based implementation of Horn clauses that employs both top-down and set-at-a-time strategies. The results of this dissertation lead to a framework for program analysis such that analysis problems are naturally expressed as Horn clauses and the evaluation of Horn clauses is efficiently implemented by MTBDDs. This framework is especially suitable for type-based program analysis since types are naturally expressed as functions.

1.4 Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents a literature review of the materials that are related to our work. Chapter 3 gives an example that uses Horn clauses to express a typical type-based program analysis, strictness analysis, by taking the advantage of functions. A top-down set-at-a-time evaluation algorithm of Horn clauses is presented in Chapter 4. In order to find a way that implements Horn clauses with a BDD-like data structure, an automata representation for Horn clause terms is proposed in Chapter 5. Chapter 6 describes conjunction and disjunction operations of automata. Chapter 7 describes the normalization procedure of automata. Chapter 8 describes grouping and ungrouping operations of automata. Chapter 9 describes expansion and projection operations of automata. Chapter 10 discusses other automata operations and the implementation of operations used in top-down evaluation algorithm. Chapter 11 presents a MTBDD implementation of automata. Chapter 12 briefly describes the modules in the implementation of our framework. Finally, Chapter 13 concludes.

Chapter 2

Related Work

2.1 Using Logic Program Languages for Program Analysis

Logic program languages, such as Prolog and Datalog, have been used successfully to express program analysis problems. Dawson, *et al.* [17] performed groundness analysis on logic programs and strictness analysis on functional programs by using Prolog. Thomas Reps [44] suggested that logic programming language is useful for many context-free language reachability problems. However, the implementations of logic program systems did not show impressive performances when they are used in program analysis. Dawson only performed analyses on programs with several hundred lines of code in the XSB system. The performance of doing those analyses on large programs in the XSB system is unknown. It has been pointed out that the XSB system is not efficient for manipulating large set of tuples [20]. Reps indicated that a native C implementation is six times faster than his logic programming approach with the Coral system. The poor performance may come from the bottom-up evaluation strategy of the Coral system. As I pointed out before, in general, bottom-up evaluation is inefficient for answering queries. Recently, Whaley, *et al.* [31, 58, 59] have used Datalog to express content-sensitive pointer alias analysis and efficiently perform the analysis on large programs that have over 10^{14} contexts with a BDD-based deductive database system BDDBDDB. The success of this framework indicates that a logic programming language with a BDD-based implementation can efficiently conduct program analyses on large programs. CodeQuest is another scalable program analysis tool using Datalog as a querying language with a traditional database system as its back-end [25]. In the sequel, we briefly explain the framework of using Datalog and BDDs for program analysis. Readers are referred to [31, 58, 59] for more details.

2.1.1 Expressing Program Analysis in Datalog

Datalog is a query language for relations. It became popular around 1978 [22] and was first formalized by Ullman [51]. The atoms in Datalog have the form $R(t_1,\ldots,t_n)$ where R is a predicate and $t_1,\ldots,t_n$ are constants or variables. A rule of Datalog has the form $p \leftarrow q_1,\ldots,q_n$, where p, q_1, …, q_n are atoms. It can be read as "p is true if q_1 and … and q_n are true". The atom on the left hand side of a rule is called *head*, the conjunction of atoms on the right hand side is called *body*, a rule without body is called a *fact* (a fact is always true) and a rule without head is called a *query* or a *goal*. All the rules in Datalog are *range-restricted* in the sense that any variable in the head must appear somewhere in the body. A *relation* is viewed as a two-dimensional table associated with a predicate R. The columns of the table are called *attributes* and each attribute is associated with a finite domain that ranges over all possible attribute values. The rows of the table are called *tuples* each of which contains a value for each attribute. If tuple $(a_1,\ldots,a_n)$ is in table R then we have a fact $R(a_1,\ldots,a_n)$. Since the domains of attributes are finite, relations can be represented by BDDs such that the values in a domain of size 2^n are encoded by n bits.

A Datalog program generally consists of a set of rules and a set of relations, where a relation is a set of facts with the same predicate. In order to use Datalog and BDDs for program analysis, computer programs are first transformed (or abstracted) to relations and stored as BDDs. Program analysis problems are expressed as Datalog rules. In BDDBDDB, a specification of program analysis consists of three parts:

1. Domain Declarations: Each declared domain has a name and may have a file that maps the numerical values in this domain to their meanings. The size of each domain (the total number of elements) is also declared.

2. Relation Declarations: Each relation is declared with its attributes. Each attribute has a name and is assigned to a domain.

3. Rules: Program analysis is expressed as Datalog rules with the declared relations.

Figure 2.1 is a specification for context insensitive Java points-to analysis. In this specification, domain V is defined for local variables and method parameters, domain H is defined for heap objects and domain F is defined for field descriptors. There are four input relations. Relation vP_0 is the initial points-to relation such that $vP_0(v,h)$ holds if and only if variable v is initialized to refer to a heap object h. Relation $store$ represents store operations such that $store(x,f,y)$ holds if and only if there is a statement $x.f = y$ in the program. Similarly, relation $load$ represents load operations such that $load(x,f,y)$ holds if and only if there is a statement $y = x.f$ in the program. Relation $assign$ represents assignments

DOMAINS

V	262144	*variable.map*
H	65536	*heap.map*
F	16384	*field.map*

RELATIONS

$input$	vP_0	$(variable : V, heap : H)$
$input$	$store$	$(base : V, field : F, source : V)$
$input$	$load$	$(base : V, field : F, dest : V)$
$input$	$assign$	$(dest : V, source : V)$
$output$	vP	$(variable : V, heap : H)$
$output$	hP	$(base : V, field : F, target : V)$

RULES

$$\begin{aligned}
&(1)\ vP(v,h) &&\leftarrow vP_0(v,h).\\
&(2)\ vP(v_1,h) &&\leftarrow assign(v_1,v_2), vP(v_2,h).\\
&(3)\ hP(h_1,f,h_2) &&\leftarrow store(v_1,f,v_2), vP(v_1,h_1), vP(v_2,h_2).\\
&(4)\ vP(v_2,h_2) &&\leftarrow load(v_1,f,v_2), vP(v_1,h_1), hP(h_1,f,h_2).
\end{aligned}$$

Figure 2.1: Java context-insensitive points-to analysis (taken from [58])

such that $assign(x,y)$ holds if and only if there is an assignment $x = y$ in the program. Assume that a call graph of the program is built in advance. Then relation *assign* includes the assignments for method invocations that assign arguments to formal parameters, and the assignments for function returns that assign the values of return statements to return value destinations. The output relations specify the possible points-to relation between heap objects, and points-to relation from variables to heap objects. That is, $hP(h_1, f, h_2)$ holds if heap object field $h_1.f$ may point to heap object h_2, and $vP(v,h)$ holds if variable v may point to heap object h.

$$\begin{array}{ll}
String\ a = \text{“}fido\text{”}; & //\, vP_0(v_a, h_1) \\
String\ b; & \\
Objd = newObj(); & //\, vP_0(v_d, h_3) \\
b = a; & //\, assign(v_b, v_a) \\
d.name = b; & //\, store(v_d, name, v_b)
\end{array}$$

Figure 2.2: A simple Java program fragment

Now we explain the rules in Figure 2.1. Rule (1) stores the initial points-to relation into *vP*. The transitive closure of relation *vP* is computed by rule (2) such that if v_1 includes v_2 and v_2 may point to heap object *h*, then v_1 may point to *h* as well. Rule (3) and (4) deal with store and load operations respectively. In rule (3), if there is a store operation such that $v_1.f = v_2$ and variable v_1 may point to heap object h_1 and variable v_2 may point to heap object h_2, then field $h_1.f$ may point to h_2 as well. Similarly, in rule (4), if there is a load operation such that $v_2 = v_1.f$ and variable v_1 may point to heap object h_1 and field $h_1.f$ may point to heap object h_2, then v_2 may point to h_2 as well.

Context insensitive points-to analysis is illustrated by a simple Java program (shown in Figure 2.2). In that figure, the initial relations that are transformed from the source code are listed as comments following the corresponding statements. Domain *V* contains three variables v_a, v_b and v_d, domain *H* contains two heap objects h_1 and h_3, and there is only one field *name* in domain *F*. Initial points-to relation contains $vP_0(v_a, h_1)$ and $vP_0(v_d, h_3)$. There is one assignment in the program, its corresponding relation is $\{assign(v_b, v_a)\}$, and there is one store operation that corresponds to relation $\{store(v_d, name, v_b)\}$. The analysis starts from rule (1), which finds that $vP(v_a, h_1)$ and $vP(v_d, h_3)$ hold from the initial points-to relation. Moreover, rule (2) says that $vP(v_b, h_1)$ holds since both $assign(v_b, v_a)$ and $vP(v_a, h_1)$ hold. Similarly, rule (3) finds that $hP(h_3, name, h_1)$ holds since $store(v_d, name, v_b)$ and $vP(v_d, h_3)$ and $vP(v_b, h_1)$ all hold.

2.1.2 From Datalog to Relational Algebra

A Datalog query is said to be *domain independent* if its answers only depend on the databases and the constants in the query, and not on the domain of interpretation. When a Datalog program is domain independent and only has finite answers, it is said to be a *safe program* and there is an equivalent relational algebra expression for every safe Datalog program [61].

The Datalog used in BDDBDDB has the following differences from standard Datalog:

- Totally ordered finite domain. In BDDBDDB, each variable in a relation is given a finite domain.
- Comparison operators. BDDBDDB includes built-in comparison operators for comparing domain elements with respect to the associated ordering.
- Unbound variables in the head predicate. BDDBDDB allows unbound variables appearing in the head. Since each variable in BDDBDDB is associated with a finite domain, the evaluation of a query with unbound variables also has finite number of answers.

From the above, we conclude that the Datalog used in BDDBDDB is a safe program. Thus, its rules can be translated into relational algebra expressions via the following operations:

1. Natural join $\bowtie$: Given two relations R and S, $R \bowtie S$ returns a set of all combinations of the tuples from R and S that agree on the common attributes.

2. Union $\cup$: Given two relations R and S, $R \cup S$ returns the union of R and S.

3. Difference $\setminus$: Given two relations R and S, $R \setminus S$ returns the difference of R and S.

4. Projection π: Given a relation R, $\pi_{a_1,\ldots,a_k}(R)$ returns a new relation obtained by removing attributes $a_1,\ldots,a_k$ from R.

5. Renaming ρ: Given a relation R, $\rho_{a \to a'}(R)$ returns a new relation obtained by renaming attribute a to a' in R.

6. Selection σ: Given a relation R, $\sigma_{a=c}(R)$ returns the tuples in R whose value on attribute a is equal to c.

Figure 2.3 shows the corresponding relational algebra operations for the rule $vP(v_1,h) \leftarrow assign(v_1,v_2), vP(v_2,h)$. Relation vP contains attributes *variable* and *heap*, and relation *assign* has attributes *dest* and *source*. We first rename attribute *variable* to *source* in relation vP and get a new relation t_1 that contains two attributes *source* and *heap*. Since relation *assign* contains attributes *dest* and *source*, the natural join $assign \bowtie t_1$ returns a relation t_2 that contains three attributes *dest*, *rule:*

$$vP(v_1,h) \leftarrow assign(v_1,v_2), vP(v_2,h)$$

relational operations:

$$\begin{aligned} t_1 &= \rho_{variable \to source}(vP); \\ t_2 &= assign \bowtie t_1; \\ t_3 &= \pi_{source}(t_2); \\ t_4 &= \rho_{dest \to variable}(t_3); \\ vP &= vP \cup t_4; \end{aligned}$$

Figure 2.3: Relational operations (taken from [58])

source and *heap*. Then we project away attribute *source* from relation t_2 and rename attribute *dest* to *variable*, the resulting relation is t_4, which contains attributes *variable* and *heap*. Finally, the union of t_4 and relation vP is the resulting relation after applying the above rule. The difference operation is used in the semi-naive bottom-up strategy [7] to drop off answers computed in the previous iteration and keep the new answers for the next iteration.

2.1.3 From Relations to Boolean Functions

The elements of a domain in a relation can be represented by numbers starting from 0. A domain with size 2^m is encoded with m bits. An n-ary relation R can be represented as a boolean function $f : D_1 \times \ldots \times D_n \rightarrow \{0,1\}$, where D_i is the numerical domain for the ith attribute and corresponds to a sequence of boolean variables. A tuple $(d_1,...,d_n)$ is in R if and only if $f(d_1,...,d_n) = 1$. There is a logical operation, which produces the same result when applied to boolean function representations of relations, for each relational algebra operation. Suppose relation R_1 is represented by $f_1 : D_1 \times D_2 \rightarrow \{0,1\}$ and R_2 by $f_2 : D_2 \times D_3 \rightarrow \{0,1\}$. Then natural join $R_1 \bowtie R_2$ is represented by $f_3 : D_1 \times D_2 \times D_3 \rightarrow \{0,1\}$, where $f_3(d_1,d_2,d_3) = f_1(d_1,d_2) \wedge f_2(d_2,d_3)$. Relation $\sigma_{a=c}(R)$ is equivalent to the natural join $R \bowtie R'$ such that R' has only one attribute a and contains only one tuple $\{c\}$. The $\cup$ operation is implemented by the logical operation $\vee$ and $R_1 \setminus R_2$ is equivalent to $R_1 \bowtie (\neg R_2)$ in which $\neg R_2$ is represented by function f_2' such that $f_2' = \neg f_2$. Expression $\pi_{a_2}(R_1)$ (a_2 is associated with domain D_2) is represented by $f : D_1 \rightarrow \{0,1\}$ such that $f(d_1) = \exists d_2 . f_1(d_1,d_2)$. A renaming operation $\rho_{d_1 \rightarrow d_3}(R_1)$ is translated into a replace operation on $f_1 : D_1 \times D_2$ in which the boolean variables corresponding to attribute d_1 are replaced with the boolean variables corresponding to attribute d_3, and the result is a function $f_3 : D_2 \times D_3$ such that $f_1(d_1,d_2) = 1$ if and only if $f_3(d_2,d_1) = 1$.

2.1.4 From Boolean Functions to BDDs

Boolean functions are efficiently represented by BDDs [11, 12]. A BDD represents a function $f : \{0,1\}^n \rightarrow B$ that maps a finite set of boolean variables to 1 or 0. Such representation is a rooted directed acyclic graph with two terminal nodes 0 and 1, and a set of non-terminal nodes. Each non-terminal node is labeled by a boolean variable and has two outgoing edges that are called "high" (or "then") edge and "low" (or "else") edge. Given a set of input bits, we evaluate a BDD by following the path from the root to a terminal node such that at each node if the corresponding bit is "1" then the "high" edge is taken, otherwise the "low" edge is chosen.

We say a BDD is *ordered* if variables appear in the same order on all the paths from the root. A BDD is called *reduced* if identical structures are collapsed into a single graph. Reduced and ordered BDDs are maximally sharing structures that compactly represent boolean functions.

The classical BDD operation *apply* implements the basic logic operations $\wedge$, $\vee$ and $-$. Moreover, other BDD operations that are needed to implement relational operations are listed as follows:

1. *exists*: Given a BDD B_f for boolean function f and a boolean variable x, $exists(B_f, x)$ replaces any non-terminal node u that is labeled by x with the "OR" of the children of u. The resulting BDD $exists(B_f, x)$ represents boolean function $\exists x.f$.

2. *replace*: Given a BDD B and a function φ that maps a boolean variable to another boolean variable, $replace(B, \varphi)$ is obtained from B by replacing the label x of each non-terminal node with $\varphi(x)$ and reordering the nodes to enforce the ordering on the new labels. The complexity of this operation is linear if the new variables keep the relative ordering of the old ones. If the relative ordering is changed then the cost of reordering could be exponential.

Note that, in the translation of a logic program, a natural join operation is always followed by a projection operation and these operations can be efficiently combined into one operation named *relprod* (stands for relational product). Similarly, a selection and a projection can be combined into one operation named *restrict*. Since the complexity of BDD operation only depends on the size and the shape of the BDD and not on the size of the relation, large relation can be manipulated very efficiently as long as its BDD representation is compact. Moreover, BDD implementations always use caches to maintain computed results and to guarantee that identical sub-problems are computed only once.

2.1.5 Using Datalog and BDDs for Program Analysis

A specification used in BDDBDDB for context-insensitive Java points-to analysis is shown in Figure 2.1. To conduct a context-sensitive points-to analysis, Whaley, *et al.* [59] presented a cloning-based approach such that a clone of a method is created for each distinct call context, a complete call graph is built from those clones and then a context-insensitive points-to analysis is performed on the pre-computed call graph to get the context-sensitive results. Although exponential explosion may occur due to the large number of clones, the underlying BDDs are created with reasonable sizes since the clones of the same method usually share some commonalities. With this cloning-based approach, context-sensitive points-to analysis can be completed on a program with over 10^{14} contexts. Although BDDs implementation of Datalog has been proved very efficient for some program analysis problems, the fact that function symbols are prohibited in Datalog makes it not suitable for some type-based program analyses in which types are formed by recursively defined objects. For example, with function symbols, it is easy to express the function type in the following type inference rule that we mentioned before:

$$type(A,T_2) \leftarrow application(A,X,Y), type(X,arrow(T_1,T_2)), type(Y,T_1).$$

More importantly, in this case, the domain of variables T_1 and T_2 may contain arbitrarily large types because of the presence of function symbol *arrow*. BDDB-DDB can only handle this type object under certain assumptions (such as fixed nesting depth of functions).

2.2 Evaluation of Horn Clauses

In general, there are two ways to evaluate a logic program, top-down and bottom-up.

Top-down evaluation is efficient for answering queries because it is goal-directed, i.e., only the results relevant to the goal are computed. However, this approach may go into an infinite loop because of recursive clauses. Moreover, most of implementations of top-down evaluation compute a tuple at a time. This strategy may lose some efficiency when accessing large set of tuples. To overcome the termination problem, researchers introduced the memoing (also called tabling) technique into top-down evaluation to memorize the answers of a subgoal so that they can be reused in the future when the same subgoal is called [57].

To remove the potential bottleneck of tuple-at-a-time strategy, researchers convert the depth-first subgoal scheduling scheme to a breadth-first one in order to obtain a set-at-a-time search engine [20]. However, this conversion is not natural since it loses the essential feature of top-down evaluation, namely “depth-first”. If we only search some but not all answers for a query, then depth-first evaluation method is more efficient than the breadth-first evaluation.

Bottom-up evaluation can avoid infinite loops by recognizing cycles generated by recursive clauses. It is usually adopted in deductive databases since it is set-at-a-time and is efficient for disk-resident data. However this approach is inefficient for answering queries because queries are solved only after all the tuples have been computed and many possibly irrelevant tuples are generated. A goal-directed bottom-up evaluation can be achieved by transforming the logic program to a form in which the evaluation only focuses on relevant tuples. This transformation is called the magic-set transformation [8]. For Datalog program, bottom-up evaluation with magic-set transformation is at least as efficient as top-down evaluation [52]. However for more general logic programs, bottom-up evaluation can do much worse than top-down evaluation due to the non-ground terms. In the presence of large non-ground terms, the “answer-return” unifications performed by bottom-up search are very costly. Bottom-up evaluation with non-ground terms was optimized by Sudarshan and Ramakrishnan [50] with a somewhat refined version of the magic-set transformation. They proved that given a program P and a query Q, if the cost of Prolog evaluation of Q is time unit t, then their optimization evaluates Q on P in time $O(t \cdot \log\log t)$. However, Ramakrishnan and Ullman also pointed out that Prolog-style (top-down depth-first tuple-at-a-time) evaluation is likely to run faster for many programs in practice [43].

It is desirable to have an evaluation, which is top-down depth-first and set-at-at-time, to efficiently answer queries with large programs. However, this kind of strategy did not receive much attention in the last decade. Early in 1987, Vieille proposed the query-subquery recursive (QSQR) evaluation algorithm for Datalog, which is top-down and set-at-a-time [53] but incomplete [54]. The QSQR algorithm is generalized by Bugaj and Nguyen to a top-down set-at-a-time depth-first evaluation for Horn knowledge bases [33]. However, Bugaj and Nguyen's original algorithm is incomplete in the sense that it fails to find all the answers in certain cases. Recently, they released a revised and extended report to correct the incompleteness [34]. The revised evaluation algorithm is complete for the goals with bounded nesting depth of functions.

2.3 Type-based Program Analysis

Type-based program analysis is a collection of techniques that uses types to infer the properties of computer program and takes advantage of the assumption that the program type checks [42]. This approach is simple because type rules are easy to understand and type derivations of programs are more convenient for designing static analysis. Moreover, it is easy to prove the correctness of an analysis that uses types since there is a well understood method to prove the soundness of a type system [40, 60] and the soundness proof of a type system often automatically verifies the correctness of the analysis.

Type-based program analyses are usually used in the following applications:

1. *Bind-Time Analysis*: This analysis is to identify static variables and expressions of a program that can be evaluated at compile-time, and dynamic variables and expressions that can only be evaluated at run-time [23,27,28,41]. The target programs are optimized by compilers via partial evaluations of static variables and expressions.

2. *Strictness Analysis*: If a functional programming language does not evaluate function arguments unless their values are required to evaluate the function call itself, then such language is called a *lazy* functional programming language. We say a function is *strict* in a parameter if this parameter is undefined implies that the result of the function is also undefined. The purpose of strictness analysis is to show that a function in a lazy functional programming language is strict in one or more parameters. The compiler can then use this information to decide whether it is safe to evaluate a parameter before passing it to the function. The strictness analysis was first introduced by Mycroft who showed how to use abstract interpretation for strictness analysis for first order functions [36]. Burn, *et al.* [14] extended Mycroft's work to higher order functions. Type inference was first used in strictness analysis by Kuo and Mishra [30]. After that, a number of researchers proposed more accurate strictness analysis based on types and generalized the approaches to type-based program analysis frameworks [10,13,26,29].

3. *Totality Analysis*: This analysis is as useful as strictness analysis, but it has not received so much attention until recently. This analysis is to show that an argument of a function is terminating so that it is safe to evaluate that argument before performing the function call [16, 47–49].

4. *Race Detection*: In a multi-threaded program, threads accessing shared data structures without synchronization may cause inconsistencies. Race detection is to ensure that the lock of a shared data structure is held by at most one thread at a time. Type-based race detection for Java programs has been proposed by several groups such as Flanagan, *et al.* [2, 18, 19], Agarwal, *et al.* [3, 4, 45].

5. *Secure Information Flow Analysis*: This analysis is to ensure that secure information of a program is not leaking, that is, there is no information flow from "high" level variables to "low" level ones if we classify variables into different security levels. A type-based system for secure information flow analysis was proposed by Volpano, *et al.* [55, 56]. This system is extended by Banerjee and Naumann to a Java-like object-oriented programming language [9].

Chapter 3

Expressing Strictness Analysis in Horn Clauses

3.1 Horn Clauses

In this chapter, we show an example that uses Horn clauses to express a typical type-based analysis, strictness analysis. In the sequel, we introduce some basic notations and then give the definition of Horn clauses.

A set of variable symbols is denoted by V. A set of function symbols is denoted by Σ. We use x, y, z, v for variable symbols and a, b, c, d, f, g for function symbols. The *arity* of a function symbol f is denoted by $\#_f$. We use the term *constant* to refer to a symbol c in Σ such that $\#_c = 0$ and we use the term *function symbol* to refer to a symbol f in Σ such that $\#_f > 0$. We define a *term* to be a constant, or a variable, or a function $f(t_1, \ldots, t_{\#_f})$ in which f is a function symbol and $t_1, \ldots, t_{\#_f}$ are terms themselves. Sometimes, we omit parentheses and commas from a term $f(t_1, \ldots, t_{\#_f})$ and denote it by $f t_1 \ldots t_{\#_f}$. A term t is *linear* if each variable of t occurs only once, otherwise t is *non-linear*. In this dissertation, when we use the word "term", we always entertain the possibility of non-linearity. An *atom* has the form $p(t_1, \ldots, t_n)$, where p is called a *predicate* and $t_1, \ldots, t_n$ are terms. A *Horn clause* is a clause of the form $L \leftarrow R$, where L is an atom called *head*, R is a conjunction of atoms called *body*, and $\leftarrow$ means "is implied by". The Horn clause of the form $A \leftarrow B_1, \ldots, B_n$ can be read as "if B_1 and $\ldots$ and B_n then A". If A is an atom of the form $p(t_1, \ldots, t_m)$ then we say this clause *defines predicate* p. We also call a clause with the form $A \leftarrow B_1, \ldots, B_n$ a *definite* Horn clause. In this dissertation, we restrict our attention to definite Horn clauses. A Horn clause without body is called a *fact*, and a Horn clause without head is called a *query* or a *goal*. If a Horn clause has a head and a non-empty body, then we call it a *rule*. A *Horn clause program* is a set of rules and facts.

3.2 Type-based Strictness Analysis

In this section, we use Horn clauses to express strictness analysis for a simple typed lambda calculus. The grammar of this typed lambda calculus is shown as the following:

$$\sigma = Int \mid Bool \mid \sigma \rightarrow \sigma$$

$$e = x \mid c \mid e_1 \ op \ e_2 \mid \ \lambda x^{\sigma}.e \mid e_1 e_2 \mid if \ e_1 \ then \ e_2 \ else \ e_3$$

As we mentioned before, a function is *strict* in a parameter if the undefinedness of this parameter implies that the result of the function is also undefined. Compilers of lazy functional programming languages use the information provided by strictness analysis to decide whether it is safe to evaluate a parameter before passing it to the function (i.e., to use a pass-by-value strategy). In type-based strictness analysis, two type constants are defined. The symbol $\perp$ denotes undefinedness and the symbol $\top$ denotes all values of the type. In strictness analysis, a function with strictness type $\perp \rightarrow \perp$ means this function is strict. The strictness logic used in this dissertation follows Jensen's work [29]. For simplicity, we do not consider conjunctive types.

In order to perform strictness analysis on the simple typed lambda calculus, we index lambda calculus expressions by integers (e.g., by numbering the nodes of

$\lambda x^{\sigma}.x:$	$lambda\_expr(e_1, x, \sigma, e_2).$
	$var\_expr(e_2, x).$
$xy:$	$app\_expr(e, e_1, e_2).$
	$var\_expr(e_1, x).$
	$var\_expr(e_2, y).$
$x\ op\ c:$	$binop\_expr(e, e_1, e_2).$
	$var\_expr(e_1, x).$
	$const\_expr(e_2, c).$
$if\ c\ then\ x\ else\ y:$	$if\_expr(e, e_1, e_2, e_3)$
	$const\_expr(e_1, c).$
	$var\_expr(e_2, x).$
	$var\_expr(e_3, y).$

Figure 3.1: Transform syntax of lambda calculus to facts

the parse tree) and transform the syntax of a lambda calculus into facts as shown in Figure 3.1 (note that, the facts are examples and not general rules). In such transformation, each expression of lambda calculus corresponds to a predicate whose first argument is the index of the expression and the rest of arguments correspond to the components in that expression. For example, application xy is transformed to a predicate $app\_expr(e, e_1, e_2)$ in which e is the index of this expression and e_1 and e_2 are indices of the sub-expressions x and y respectively. We use V to denote the finite domain of all the variables appearing in a lambda calculus program, C to denote the domain of all the constants, T to denote the domain of the types, and we use I to denote the domain of the indices of the expressions. The domain of each predicate is listed as follows:

- $lambda\_expr : I \times V \times T \times I.$

- $app\_expr : I \times I \times I$.
- $binop\_expr : I \times I \times I$.
- $if\_expr : I \times I \times I \times I$.
- $var\_expr : I \times V$.
- $const\_expr : I \times C$.

The type inference rules for strictness analysis are shown in figure 3.2. A judgment has the form $E \vdash e : t$, where E is the environment that maps program variables to their strictness types, e stands for an expression and t is the strictness type of e under the environment E. Rule **Var** deduces the trivial strictness types for variables. Rule **Abs** states that if expression e has the type t_2 under the environment E combined with a mapping that binds variable x to strictness type t_1, then the lambda expression $\lambda x^t.e$ has strictness type $t_1 \rightarrow t_2$ under the environment E. Rule **App** expresses that if e_1 maps arguments with type t_1 to results with type t_2, then applying this expression to an argument with type t_1 will get the result with strictness type t_2. Rule **Op-Left** and rule **Op-Right** handle arithmetic operations and logical operations, they say that a binary operation is strict in each of its arguments. Rule **If-1** and rule **If-2** for the conditional imply that the conditional has strictness type $\perp$ if the boolean condition is undefined or both of the branches are undefined.

The Horn clauses for strictness analysis are shown in Figure 3.3. Note that, by following the convention of Prolog, we use lower-case characters for constants, function symbols and predicates, use upper-case characters for variables, and use wildcard (_) in a predicate to define an argument that can match anything. The

Var $E[x:t] \vdash x:t$

Abs $$\frac{E[x:t_1] \vdash e:t_2}{E \vdash \lambda x^t.e : t_1 \rightarrow t_2}$$

App $$\frac{E \vdash e_1 : t_1 \rightarrow t_2 \quad E \vdash e_2 : t_1}{E \vdash e_1 e_2 : t_2}$$

Op-Left $$\frac{E \vdash e_1 : \bot \quad E \vdash e_2 : t_2}{E \vdash e_1 \quad op \quad e_2 : \bot}$$

Op-Right $$\frac{E \vdash e_1 : t_1 \quad E \vdash e_2 : \bot}{E \vdash e_1 \quad op \quad e_2 : \bot}$$

If-1 $$\frac{E \vdash e_1 : \bot}{E \vdash if \quad e_1 \quad then \quad e_2 \quad else \quad e_3 : \bot}$$

If-2 $$\frac{E \vdash e_2 : t \quad E \vdash e_3 : t}{E \vdash if \quad e_1 \quad then \quad e_2 \quad else \quad e_3 : t}$$

Figure 3.2: The type inference rules for strictness analysis

predicate *type* represents the judgment $E \vdash e : t$ with three arguments, the first one is for environment E, the second one is for expression e and the third one is for strictness type t. The environment is coded as a list of pairs of the form (x,t) such that (x,t) binds strictness type t to variable x. Atom $env((X,T),Env)$ represents an environment (list) such that the binding (X,T) is the head and environment (list) Env is the tail. The empty environment is represented by constant [ ]. Clauses (1) and (2) define the predicate *is_member*. Clause (1) says that if there is an environment that contains the head (X,T) then (X,T) is a member of that environment. Clause (2) says that if X and X' are not literally

(1) $is\_member((X,T), env((X,T), Env)).$

(2) $is\_member((X,T), env((X',T'), Env)) \leftarrow not\_eq(X,X'),$
$is\_member((X,T), Env).$

(3) $type(Env, X, T) \leftarrow var\_expr(X,\_), is\_member((X,T), Env).$

(4) $type(Env, L, arrow(T_1,T_2)) \leftarrow lambda\_expr(L, X, T_1, E),$
$type(env((X,T_1), Env), E, T_2).$

(5) $type(Env, A, T_2) \leftarrow app\_expr(A, E_1, E_2),$
$type(Env, E_1, arrow(T_1,T_2)), type(Env, E_2, T_2).$

(6) $type(Env, B, bottom) \leftarrow binop\_expr(B, E_1, \_), type(Env, E_1, bottom).$

(7) $type(Env, B, bottom) \leftarrow binop\_expr(B, \_, E_2), type(Env, E_2, bottom).$

(8) $type(Env, I, bottom) \leftarrow if\_expr(I, E_1, \_, \_), type(Env, E_1, bottom).$

(9) $type(Env, I, T) \leftarrow if\_expr(I, \_, E_2, E_3), type(Env, E_2, T),$
$type(Env, E_3, T).$

Figure 3.3: Horn clauses for strictness analysis

identical then we have that $is\_member((X,T), env((X',T'), Env))$ holds whenever $is\_member((X,T), Env)$ holds. Note that, *not_eq* is a primitive predicate and $not\_eq(X,Y)$ holds if and only if X and Y are not literally identical. In this dissertation, inequality comparisons are only allowed on finite domains. Clause (3) is for rule **Var**, it says that if X is the index of a variable expression and the binding (X,T) is a member of environment *Env* then X has strictness type T under the environment *Env*. Rule **Abs** is expressed in clause (4) such that if there is a lambda expression $L = \lambda X^t.E$ and term E is associated with strictness type T_2 under environment $env((X,T_1),Env)$ (in which variable X has type T_1) then expression L is associated with type $arrow(T_1,T_2)$ under environment Env. Here, $arrow(T_1,T_2)$ stands for strictness type $T_1 \rightarrow T_2$, which is a recursively defined type that consists of subtypes T_1 and T_2. The rest of the clauses express the corresponding rules straightforwardly. Clause (5) is for rule **App**, clauses (6) and (7) are corresponding to rules **Op-Left** and **Op-Right**, and clause (8) and (9) are corresponding to rules **If-1** and **If-2**. This example shows that Horn clauses are expressive enough to define type-based program analysis by utilizing the functions for recursively defined objects.

Chapter 4

Top-down Set-oriented Algorithm

4.1 Set-At-A-Time Evaluation

Top-down depth-first evaluation is efficient for answering queries of Horn clauses due to its goal-directed strategy. Most of implementations of top-down evaluation compute a single tuple at a time. This strategy may lose some efficiency when accessing large set of tuples. To remove the potential bottleneck of tuple-at-a-tome strategy, Bugaj and Nguyen proposed a top-down set-at-a-time depth-first evaluation algorithm that computes a set of tuples at a time. Their original algorithm is incomplete in the sense that it fails to find all the answers in certain cases [33]. Recently, they released a revised and extended report to correct the incompleteness [34]. In this Chapter, we present the corrected version of top-down set-at-a-time evaluation algorithm in terms of sets of ground n-tuples.

4.2 Notations

A *substitution* is defined to be a function that maps a variable to a term, and is denoted by $[\theta(x_1)/x_1,\ldots,\theta(x_n)/x_n]$ in which $\{x_1,\ldots,x_n\}$ is the domain of the substitution. If θ is a substitution and E is a term, a tuple of terms, an atom, a substitution or a Horn clause, then $E\theta$ is obtained from E by simultaneously replacing each variable x of E with $\theta(x)$. If θ and δ are substitutions such that $\theta = [\theta(x_1)/x_1,\ldots,\theta(x_n)/x_n]$, then $\theta\delta = [\theta(x_1)\delta/x_1,\ldots,\theta(x_n)\delta/x_n]$. If $E' = E\theta$ then we say E' is a *substitution instance* of E. Moreover, if $\theta(x_i)$ is a ground term for all $i \in [1,n]$, then we say θ is a *ground substitution* and E' is a *ground substitution instance* of E. For example, atom $p(X,Y,a)$ is substitution instance of atom $p(X,Y,Z)$ with substitution $[a/Z]$ and atom $p(a,b,c)$ is a ground substitution instance of $p(X,Y,Z)$ with ground substitution $[a/X,b/Y,c/Z]$.

If θ is a substitution such that $E\theta = E'\theta$ then we say θ is a *unifier* of E and E'. A substitution θ is called a *most general unifier* (*mgu*) of E and E' if for any other substitution δ of E and E', we have $\delta = \theta\gamma$, where γ is also a substitution. The most general unifier is unique up to renaming. For example, $[a/X,b/Y,b/Z]$ is a unifier of atoms $p(X,Y,c)$ and $p(a,Z,c)$ and $[a/X,Y/Z]$ is a mgu of those atoms.

We define an *n-tuple* to be a tuple of n terms and a *ground n-tuple* to be an n-tuple that does not contain any variable. An n-tuple t is *linear* if each variable of t occurs only once, otherwise t is *non-linear*. We say a set of ground n-tuples S is *an instance of atom* $p(\bar{t})$ if and only if the arity of p is n and every ground n-tuple of S is a ground substitution instance of $\bar{t}$. In some cases, we also say S is *an instance of tuple* $\bar{t}$. If S contains all the ground substitution instances of n-tuple $\bar{t}$ then we say S is *the complete instance of atom* $p(\bar{t})$. Similarly, in some cases, we also say S is *the complete instance of tuple* $\bar{t}$. Note that, a set of ground n-tuples that is the complete instance of an atom may be infinitely large because of the presence of function symbols. The *term depth* of a ground n-tuple is the maximal nesting depth of function symbols that appear in that n-tuple, and we denote the term depth of n-tuple t by $|t|_\Sigma$. For example, $|(a, f(a,g(b)))|_\Sigma = 2$. We denote an n-tuple that contains all the distinct variables appearing in a set of atoms S by $Var(S)$ (in which variables appear in an arbitrary ordering). For example, $Var(\{p(X,g(Z)),q(X,Y)\})$ could be (X,Y,Z).

We classify each predicate as either *intensional predicate* or *extensional predicate*. Intensional predicates are defined by rules, while extensional predicates are defined by facts. For example, if we have a rule

$$path(X,Y) \leftarrow path(X,Z), edge(Z,Y)$$

and a fact $edge(a,b)$, then *path* is an intensional predicate while *edge* is an extensional one.

In this chapter, a *Horn knowledge base* is defined to be a Horn clause program P that defines intensional predicates, and an *extensional instance* I, which is a function that maps each n-ary extensional predicate to a set of ground n-tuples. For example, a Horn knowledge base that computes the transitive closure of a graph contains a program

$$P = \{path(X,Y) \leftarrow edge(X,Y)., path(X,Y) \leftarrow path(X,Z), edge(Z,Y).\}$$

and an extensional instance I that maps predicate *edge* to a set of ground 2-tuples $\{(a,b),(b,c),(b,d)\}$.

Note that, each ground n-tuple $\bar{t}$ in $I(p)$ is treated as an atom $p(\bar{t})$ and $I(p)$ is treated as a set of ground atoms of p. We use E_I to denote an extensional database associated with extensional instance I such that $E_I = I(p_1) \cup \ldots \cup I(p_n)$, where $p_1, \ldots, p_n$ are all the extensional predicates. Suppose a set S of ground n-tuples has the same arity as predicate p. If $\bar{t} \in S$, then we also say that atom $p(\bar{t})$ is in S.

Suppose (P,I) is a Horn knowledge base. Then a goal (or query) has the form

$\leftarrow A_1, \ldots, A_k$ in which $A_1, \ldots, A_k$ are all atoms. Note that, an arbitrary goal can be transformed into an equivalent one of the form $\leftarrow q(\bar{x})$ in which q is a predicate and $\bar{x}$ is a tuple of variables. Such transformation is done in polynomial time by introducing new intensional predicates and new Horn clauses. For example, goal $\leftarrow p(X), q(X,Y)$ can be transformed into $\leftarrow r(X,Y)$ by introducing new intensional predicate r and new clause $r(X,Y) \leftarrow p(X), q(X,Y)$. Without loss of generality, we assume that a top goal always has the form $q(\bar{x})$. A *correct answer* of a top goal $q(\bar{x})$ on Horn knowledge base (P,I) is a ground substitution θ such that $q(\bar{x})\theta$ is a logical consequence of $P \cup E_I$, and we write $P \cup E_I \models q(\bar{x})\theta$.

4.3 The Algorithm

The following operations are used in Bugaj and Nguyen's algorithm:

1. *instance*$(\bar{t})$:
   - where $\bar{t}$ is an n-tuple,
   - returns a set of ground n-tuples S, which is the complete instance of $\bar{t}$.
2. *union*(S_1, S_2) and *diff*(S_1, S_2):
   - where S_1 and S_2 are sets of ground n-tuples,
   - return $S_1 \cup S_2$ and $S_1 \setminus S_2$ respectively.
3. *join_with_head_atom*$(J, A, \overline{X})$:
   - where:
     (a) J is a set of ground n-tuples.

(b) A is an atom of an n-ary predicate,

(c) $\overline{X}$ is a k-tuple of variables that contains all the variables appearing in A,

- returns a set of ground k-tuples

$$\{\overline{X}\theta \mid \quad A\theta \in J \textit{ and } \theta \textit{ is a ground substitution} \\ \textit{whose domain is } \overline{X}\}.$$

4. *join_with_body_atom*$(\overline{X}, S, B, M)$:

- where

(a) $\overline{X}$ is a k-tuple of variables that contains all the variables appearing in B,

(b) S, which is a set of ground k-tuples, is an instance of k-tuple $\overline{X}$,

(c) B is an atom,

(d) M, which is a set of ground n-tuples, is an instance of atom B,

- returns the set of ground k-tuples

$$\{\overline{X}\theta \mid \overline{X}\theta \in S \textit{ and } B\theta \in M\}.$$

5. *map*$(\overline{X}, S, B)$:

- where

(a) $\overline{X}$ is a k-tuple of variables that contains all the variables appearing in B,

(b) S, which is a set of ground k-tuples, is an instance of k-tuple $\overline{X}$,

(c) B is an atom,

- returns the set of ground n-tuples $\{B\theta \mid \overline{X}\theta \in S\}$.

6. $filter(S,L)$:

- where S is a set of ground k-tuples and L is an integer,
- returns the set S' of ground n-tuples such that $S' = \{t \mid t \in S \text{ and } |t|_\Sigma \leq L\}$.

We show Bugaj and Nguyen's algorithm in Figure 4.1. Given a Horn knowledge base (P,I) and a top goal $g = q(\bar{x})$, their algorithm starts from the top goal, iteratively searches all the rules that define q until no new answers are found. In each iteration, a goal is recursively reduced to subgoals by following those rules that define the predicate of that goal. Suppose g' is a goal for predicate p. If p is extensional, then the answers of g' are $I(p)$. If p is intensional, then the algorithm recursively evaluates g' by searching all the rules that define p, reducing g' to subgoals, and obtaining the answers of g' by composing the answers from its subgoals. We will discuss the answer returning process in detail when we present the searching process of a single rule. Note that a goal (or subgoal) is always a set of ground n-tuples. Obviously, this algorithm is top-down, depth-first and set-at-a-time.

This top-down evaluation may go into a cycle if a subgoal is directly or indirectly reduced to itself. To avoid infinite computations caused by cycles, two kinds of variables are introduced:

```
TopDown Algorithm
(* Evaluate a top goal $q(\bar{x})$ on a Horn knowledge base $(P,I)$ *)
1. Initialize variables $ans_p$ to be empty
   sets for all intensional predicate $p$ of $P$
2. Initialize variable $L$ to be the maximal term depth
3. Repeat
      3.1 Initialize variables $input_p$ to be empty
          sets for all intensional predicates $p$ of $P$
      3.2 Call $eval$(the complete instance of $\{\bar{x}\}, q$)
   Until no new tuples are added to any variable $ans_p$
4. Return $ans_q$

Function $eval(J, p)$
1. $J = diff(J, input_p)$
2. if $J$ is empty then return
3. $input_p = union(input_p, J)$
4. For each program clause $A \leftarrow B_1, \ldots, B_n$ of $P$ that defines $p$, do:
   1. $i = 0$
   2. $\overline{X} = Var(\{A, B_1, \ldots, B_n\})$
   3. $S_0 = join\_with\_head\_atom(J, A, \overline{X})$
   4. While $i < n$ and $S_i$ is not empty do:
      4.1 $i = i + 1$
      4.2 if the predicate $p_i$ of $B_i$ is extensional
          then
             $S_i = join\_with\_body\_atom(\overline{X}, S_{i-1}, B_i, I(p_i))$
          else
             Call $eval(filter(map(\overline{X}, S_{i-1}, B_i), L), p_i)$
             $S_i = join\_with\_body\_atom(\overline{X}, S_{i-1}, B_i, ans_{p_i})$
   5. $ans_p = union(ans_p, map(\overline{X}, S_n, A))$
```

Figure 4.1: Bugaj and Nguyen's algorithm

- A variable input_p is created for each intensional predicate p to maintain the subgoals that are generated so far for p in each iteration.
- A variable ans_p is created for each intensional predicate p to maintain the answers that have been computed for p through all the iterations.

This caching scheme (storing evaluated subgoals in input_p and computed answers in ans_p) is also called *tabling or memoing* and it is a well-known technique that is applied in the implementations of logic programming languages to avoid infinite computations.

Suppose now we evaluate a subgoal J for predicate p. First, we would have $J' = \mathit{diff}(J, \mathit{input}_p)$. If J' is empty then the subgoal J has been evaluated before, the evaluation terminates and the answers in ans_p for this subgoal are returned directly. If J' is not empty then the new subgoal J' is merged into input_p and is evaluated by searching all the rules that define p. If after searching a rule that defines p, some answers of J' are found, then those answers are merged into ans_p.

Suppose now we search a rule $A \leftarrow B_1, \ldots, B_n$ for a goal J whose predicate is p. The searching process is to iteratively compute a set S_{i-1} of ground k-tuples that is an instance of $\overline{X}$, where $\overline{X} = \mathit{Var}(\{A, B_1, \ldots, B_n\})$. Each ground k-tuple $\overline{X}\delta$ in S_{i-1} represents a potential answer $A\delta$ of J. The subscription $i-1$ means that the unevaluated goals of this rule are $\{\leftarrow B_i\delta, \ldots, B_n\delta \mid \overline{X}\delta \in S_{i-1}\}$. After the search of this rule is finished, $\{A\delta \mid \overline{X}\delta \in S_n\}$ is the answers of J and it is merged into ans_p.

Initially, S_0 is equal to $\mathit{join\_with\_head\_atom}(J, A, \overline{X})$. That is, $\overline{X}\theta$ is in S_0 if there is a ground n-tuple $A\theta \in J$ with some ground substitution θ. Suppose S_{i-1}

is computed and the predicate of atom B_i is p_i. Now we compute set S_i. If p_i is extensional, then

$$S_i = join\_with\_body\_atom(\overline{X}, S_{i-1}, B_i, I(p_i)).$$

If p_i is intensional, then function

$$eval(filter(map(\overline{X}, S_{i-1}, B_i), L), p_i)$$

is called to recursively evaluate the subgoal generated from atom B_i and S_{i-1} (Note that, $filter(map(\overline{X}, S_{i-1}, B_i), L)$ only contains ground n-tuples whose term depths are less than or equal to L). After the call terminates,

$$S_i = join\_with\_body\_atom(\overline{X}, S_{i-1}, B_i, ans_{p_i}).$$

That is, S_i is equal to $join\_with\_body\_atom(\overline{X}, S_{i-1}, B_i, R)$ in which R is either $I(p_i)$ or ans_{p_i}. Suppose $\overline{X}\theta$ is in S_{i-1} with some ground substitution θ. If there exists a ground m-tuple $B_i\theta$ in R, then $\overline{X}\theta$ is also in S_i. That is, ground n-tuple $A\theta$ of J is still a potential answer of J after the ith atom B_i is processed.

4.4 Correctness

In this section, we first show that TopDown algorithm always terminates, and it is sound and is complete for the goals with bounded nesting depth of functions.

Then we present an example to illustrate the iterative execution of TOPDOWN algorithm.

Theorem 4.4.1 (Termination) *Suppose (P,I) is a Horn knowledge base and we run a query on* TOPDOWN *algorithm with a fixed term depth bound L. Then the algorithm runs in polynomial time in the size of the extensional instance I.*

Proof. Directly from Theorem 3.7 of [34]. □

To show that TOPDOWN algorithm is sound and complete (for the goals with bounded nesting depth of functions), we first introduce the following definitions related to SLD resolution.

Let $g = (A_1, \ldots, A_k)$ be a goal. We say *goal g' is derived from g by using an answer F in some global variable ans_p or a fact F from extensional instance I* if there exists a unifier θ such that $A_1\theta = F$ and $g' = (A_2, \ldots, A_k)\theta$. We say *goal g' is derived from g by using a clause $A \leftarrow B_1, \ldots, B_m$* if $A_1\theta = A\theta$ and $g' = (B_1, \ldots, B_m, A_2, \ldots, A_k)\theta$. In this case, A_1 is said to be *the parent* of $B_1, \ldots$, and B_m. The relation *ancestor* is defined recursively from the parent relation. If g' is derived from g then we write $g \Rightarrow g'$. Let g_0 be a given goal. Suppose (P,I) is a Horn knowledge base and g_0 is a goal. Then *an SLD resolution* from $P \cup I \cup \{g_0\}$ is a derivation $g_0 \Rightarrow g_1 \Rightarrow \ldots \Rightarrow g_n$. An *SLD-refutation* of from $P \cup I \cup \{g_0\}$ is a finite SLD-derivation in which the last goal has an empty sequence of subgoals. A *computed answer θ for $P \cup I \cup \{g_0\}$* is the substitution obtained by restricting the composition $\theta_1 \ldots \theta_n$ to the variables of g_0, where $\theta_1, \ldots, \theta_n$ is the sequence of mgu's used in an SLD-refutation of $P \cup I \cup \{g_0\}$. The SLD-resolution is sound and complete for definite programs and goals [15, 32].

Theorem 4.4.2 (Soundness) *Suppose we execute* TOPDOWN *algorithm to evaluate a query $q(\bar{x})$ on Horn knowledge base (P,I). Then, for every answer A' in some variable ans_p, we have $P \cup E_I \models A'$.*

Proof. Directly from Theorem 3.6 of [34]. □

Theorem 4.4.3 (Completeness) *Suppose θ is the computed answer of an SLD-refutation $P \cup I \cup \{q(x)\}$ that uses the leftmost selection function and does not contain any goal with term depth greater than L. Then the execution of* TOPDOWN *algorithm (with term depth L) for the query $q(x)$ on Horn knowledge base (P,I) returns a set ans_q that contains the set $\{t \mid t \in S \text{ and } |t|_\Sigma \leq L\}$, where S is the complete instance of $q(x)\theta$.*

Proof. Directly from Theorem 3.6 of [34]. □

Now we show an example in Figure 4.2 to illustrate the recursive execution of TOPDOWN algorithm.

To trace the complete evaluation of this example, we build a graph, which is called the subgoal dependency graph (**SDG**), to indicate the dependencies between subgoals. In such a graph, each node contains a subgoal and the answers for that subgoal. There is only one root node in a SDG, which contains the top goal. If there is a directed edge from a node n_1 to another node n_2, then subgoal in n_1 is dependent on the subgoal in n_2. We also say *node n_1 is dependent on node n_2*. Suppose we give an ordering on the rules in Horn clause program P as we do

- Rules:
  1. $n(X,Y) \leftarrow r(X,Y)$.
  2. $n(X,Y) \leftarrow p(X,Z),\ n(Z,W),\ q(W,Y)$.
- Facts:
  $p(c,d)$. $p(b,c)$. $p(c,b)$.
  $q(e,a)$. $q(a,i)$. $q(i,o)$.
  $r(d,e)$.
- Query: $\leftarrow n(c,Y)$.
- Expected answers: $\{(c,a),\ (c,o)\}$.
- Answers returns by TOPDOWN algorithm: $\{(c,a)\}$.

Figure 4.2: A Horn clause program

to this example. Then the label on directed edge is a pair (i,j) and it indicates that the dependency happens on the jth atom in the body of the ith rule.

After running the first iteration of TOPDOWN algorithm for this example, we obtain a SDG that is shown in Figure 4.3. Each node has three rows. The first row is the index of the node, the second row is a subgoal and the third row is the answer set for that subgoal. Note that, in each node, a subgoal is specified by a set of atoms, which stands for the set $\{t \mid t \in S \text{ and } |t|_\Sigma \leq L\}$, where S is a set that contains all the complete instances of those atoms. Now we trace the complete evaluation of this example. We start from node 1 with goal $\{n(c,Y)\}$. At this time, variable $input_n$ contains only (c,Y) and ans_n is empty. We go to node 2 by following a directed edge (start searching rule (1)). The subgoal $\{r(c,Y)\}$ has no answers. Thus, this directed edge would not contribute any answers to ans_n. We go to node 3 by following another directed edge (start searching rule (2)). At node 3, we obtain answers $\{p(c,b), p(c,d)\}$ for subgoal $\{p(c,Y)\}$. These answers

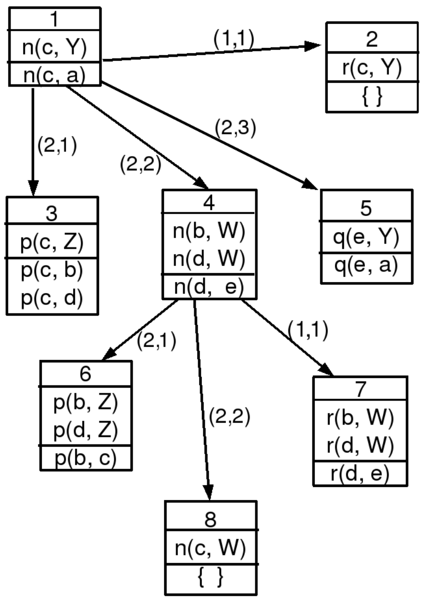

Figure 4.3: A subgoal dependency graph for the first iteration

produce subgoal $\{n(b,W),n(d,W)\}$ at node 4.

Now we are at node 4. At this time, variable *input*$_n$ is the set

$$\{(c,Y),(b,W),(d,W)\}$$

and *ans*$_n$ is still empty. We go to node 7 by following a directed edge (start searching rule (1)). At node 7, the answer for subgoal $\{r(b,W),r(d,W)\}$ is $r(d,e)$. This will produce an answer $n(d,e)$ at node 4. We go to node 6 by following another directed edge (start searching rule (2)). At node 6, we obtain answer $p(b,c)$ for subgoal $\{p(b,Y),p(d,Y)\}$. This answer produces subgoal $\{n(c,W)\}$ at node 8.

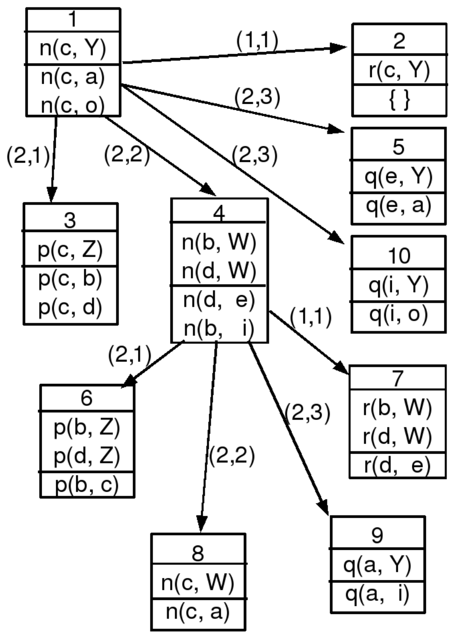

Figure 4.4: A subgoal dependency graph for the second iteration

Since subgoal $\{n(c,W)\}$ is the same as goal $n(c,Y)$ and there is no answer for goal $\{n(c,Y)\}$ yet, node 6 does not contribute any answers to node 4.

After searching all the rules that define predicate n, we obtain an answer $n(d,e)$ at node 4 for subgoal $\{n(b,W),n(d,W)\}$. In turn, the answer $n(d,e)$ will form subgoal $\{q(e,Y)\}$ at node 5. At node 5, the answer for subgoal $\{q(e,Y)\}$ is $q(e,a)$. By joining the answers in nodes 3, 4 and 5, we obtain an answer $n(c,a)$ for goal $\{n(c,Y)\}$ at node 1. Thus, after the first iteration, $input_n$ is $\{(c,Y),(b,W),(d,W)\}$ and ans_n is $\{(d,e),(c,a)\}$.

After running the second iteration of TopDown algorithm for this example, we obtain a SDG that is shown in Figure 4.4. This time, we start the search from node 1. Again, node 2 does not contribute any answers. We go from node 1 to node 3 and then to node 4. Now we search answers for subgoal $\{n(b,W),n(d,W)\}$ at node 4. We go to node 7 by following a directed edge (start searching rule (1)), however, node 7 does not contribute any new answers this time. We go to node 6 by following another directed edge (start searching rule (2)). Then we will reach node 8 with subgoal $\{n(c,W)\}$. Since $n(c,W)$ is already contained in $input_n$ (it is the same as $n(c,Y)$) and there is an answer $n(c,a)$ for goal $\{n(c,Y)\}$, an answer $n(c,a)$ is obtained at node 8. This answer forms a subgoal $\{q(a,Y)\}$ at node 9. At node 9, the answer for subgoal $\{q(a,Y)\}$ is $q(a,i)$. By joining the answers in nodes 6, 8 and 9, we obtain an answer $n(b,i)$ for subgoal $\{n(b,W),n(d,W)\}$ at node 4. In turn, the answer $n(b,i)$ will form a subgoal $\{q(i,Y)\}$ at node 10. At node 10, the answer for subgoal $\{q(i,Y)\}$ is $q(i,o)$. By joining the answers in nodes 3, 4 and 10, we obtain a new answer $n(c,o)$ for goal $\{n(c,Y)\}$ at node 1. Thus, after the second iteration, ans_n is $\{(b,i),(d,e),(c,a),(c,o)\}$. If we repeat this process, we will not find any new answers for predicate n. Thus, TOPDOWN algorithm finds all the answers for this example via iteratively searching.

Chapter 5

Definition of Automata for Sets of Ground Tuples

5.1 Representing Sets of Ground N-Tuples

In previous chapter, we gave a presentation of TopDown algorithm by using sets of ground n-tuples to represent subgoals and sets of answers. Generally, a set of ground n-tuples that is the complete instance of some atom is an infinite set because of the presence of function symbols. Therefore, a finite representation of sets of ground n-tuples is needed to implement TopDown algorithm. Here we present an automata-based representation for sets of ground n-tuples, which is called $step/skip$-automaton. This representation readily leads itself to a symbolic representation of sets of ground n-tuples. In this chapter, we first give the background of automata representations of terms. Then we present some basic notations and the definition of $step/skip$-automata. Finally we define various operations on automata to implement those operations used in TopDown algorithm.

5.2 Previous Work

Generally, an automata representation of terms is to build a deterministic finite automaton $\mathcal{M}$ that represents a set of terms T such that:

- All and only ground substitution instances of T are accepted by $\mathcal{M}$.
- A ground substitution instance t of T is accepted by $\mathcal{M}$ with a single scan.

The automata representations of terms have been studied for decades and various automata have been proposed [24, 37–39, 46].

Albert Gräf [24] proposed a tree automata representation for terms in which an automaton is a tuple (S, s_0, F, δ), where S is a finite set of states, s_0 is the start state, F is a set of final states and $F \subset S$, δ is the state transition function. A state is essentially *a matching set*, which is defined as follows:

1. Each term in the original term set T has the form $r : \alpha\beta$ in which $\alpha\beta$ is a term and r is the index of that term.

2. A *matching item* has the form $r : \alpha \bullet \beta$ in which $\alpha\beta$ is a term and r is the index of that term and the meta-symbol $\bullet$ is called the *matching dot*. For matching item $r : \alpha \bullet \beta$, α and β are called the *prefix* and *suffix* respectively. The first symbol of β is called the *matching symbol*.

3. A *matching set* is a set of matching items that have the same prefix.

Suppose automaton $\mathcal{A} = (S, s_0, F, \delta)$ represents all the ground substitution instances of a set of terms T. The terms in T are all linear and any variable occurrence in the terms is replaced by ω. Suppose f is a function symbol. Then $\omega^{\#_f}$ denotes a string of $\#_f$ variable symbols.

The initial state s_0 of $\mathcal{A}$ is the set $\{r : \bullet\alpha\beta \mid r : \alpha\beta \in T\}$. Suppose M is a state (matching set) and b is a symbol in $\Sigma \cup \{\omega\}$. Then the next state $\delta(M, b)$ is defined as follows:

1. $accept(M, b) = \{r : \alpha b \bullet \beta \mid r : \alpha \bullet b\beta \in M\}$.

2. $close(M) = M \cup \{r : \alpha \bullet f\omega^{\#_f}\beta \mid r : \alpha \bullet \omega\beta \in M$ and there is an item $q : \alpha \bullet f\mu$ in M with some suffix μ and some f in $\Sigma\}$.

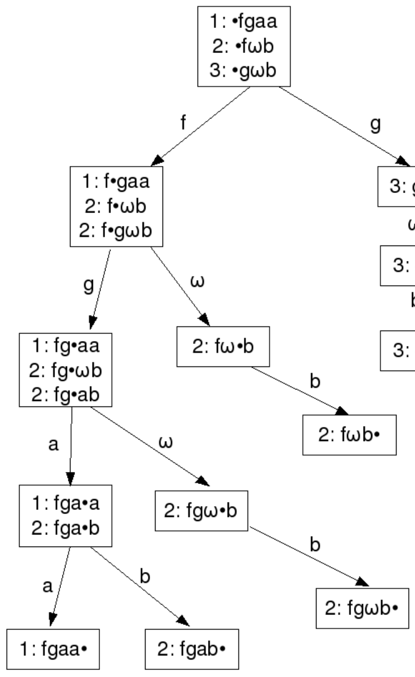

Figure 5.1: An automaton that represents all the ground substitution instances of a set of terms

3. $\delta(M,b) = close(accept(M,b))$.

Clearly, the set $accept(M,b)$ is formed by those items in M whose matching symbol is b. If $accept(M,b)$ contains suffixes of the form $\omega\beta$ and $f\mu$, then a matching item with suffix $f\omega^{\#_f}\beta$ is added into $\delta(M,b)$ by the *close* function to postpone the decision between those two suffixes by one more symbol. Without such extra matching items, backtracking will be required to match suffix $\omega\beta$ when the input

f fails to match suffix $f\mu$. Note that, transition $\delta(M, \omega)$ is taken only when the current input is b (with $b \neq \omega$) and $\delta(M, b)$ is not defined. Moreover, when transition $\delta(M, \omega)$ is taken, a complete term in the input is read. Finally, in a final state s, any matching item in s has the form $\alpha\beta\bullet$. Let $T = \{fgaa, f\omega b, q\omega b\}$ with $\#_a = \#_b = 0$ and $\#_g = 1$ and $\#_f = 2$ and $\#_q = 2$. The automaton $\mathcal{A}$ that represents all the ground substitution instances of T is shown in Figure 5.1. Note that, in this dissertation, when we show a figure of an automaton, we always omit the error states.

The automaton $\mathcal{A}$ mentioned above is time-efficient, because it does not reexamine any symbol. However $\mathcal{A}$ may not be space-efficient, because multiple states may share the same set of suffixes. Nadia Nedjah, *et al.* [39] optimized $\mathcal{A}$ by representing states with suffixes and collapsing states that contain the same set of suffixes to a single state. An optimal automaton $\mathcal{A}'$ that represents all the ground substitution instances of T is shown in Figure 5.2.

The above automata are based on left-to-right traversal of terms. Another class of automata are based on adaptive traversals, which refer to traversals that are adapted for different terms and are opposed to fixed order traversal [38, 46]. Adaptive automata for terms usually have fewer states than fixed order automata. Consequently, matching an instant of a term with adaptive traversal usually uses less time. However, adaptive automata are not suitable for applications in which terms are changed frequently. This is because the traversal order itself must be changed when a term is changed.

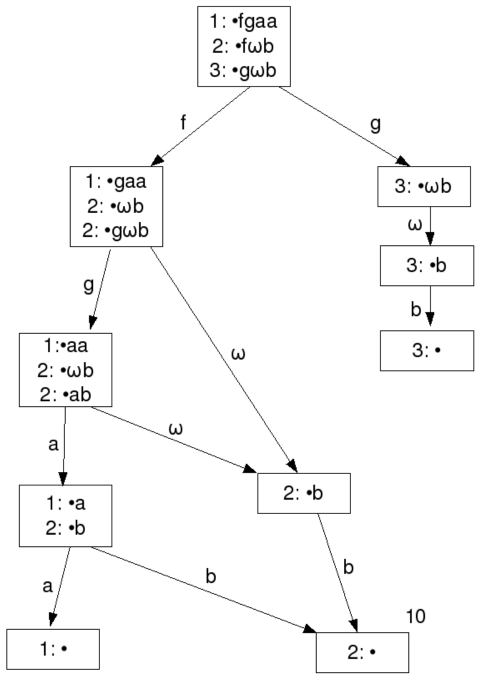

Figure 5.2: An optimal automaton that represents all the ground substitution instances of a set of terms

5.3 Informal Definition

The automata representation for terms presented in previous section (proposed by Gräf and optimized by Nedjah, *et al.*) is to compile a set T of terms into a tree automaton $\mathscr{A}$ such that the states are essentially rewriting rules. In this chapter, we propose a special kind of automata as finite representations of infinite sets of ground n-tuples, not as efficient pattern matchers. Moreover, in order to implement the TopDown algorithm, we devise various operations on our automata representation to implement those operations on sets of ground n-tuples.

Our automata representation represents ground substitution instances of non-linear n-tuples based on left-to-right traversal. We first consider an automaton that represents the complete instance of 2-tuple $(f(x,x), g(y,y))$, where $f, g \in \Sigma$ with $\#_f = \#_g = 2$ and $x, y \in V$. Our automaton representation for this n-tuple is shown in Figure 5.3. In that figure, each state is denoted by a circle and is distinguished from others by a label outside the circle. In order to read a substitution instances of an n-tuple $\bar{t}$, we need two kinds of states:

- A state that takes a constant or a function symbol in the input and goes to the next state. This state corresponds to a constant or a function symbol in $\bar{t}$, which is called "step state". For example, in Figure 5.3, states s_1 and s_4 are step states. An out-going edge of a step state is labeled by a symbol b in Σ to indicate that symbol b is read by this transition.
- A state that skips a complete term in the input and goes to the next state. This state corresponds to a variable in $\bar{t}$, which is called "skip state". For example, in Figure 5.3, states s_2, s_3, s_5 and s_6 are skip states. A skip state has only one out-going edge that is dotted. The dotted edge indicates that a complete term is skipped by this transition.

Note that, a non-terminal state is either a step state or a skip state, but cannot be both. Since the proposed automata consists of step states and skip states, they are

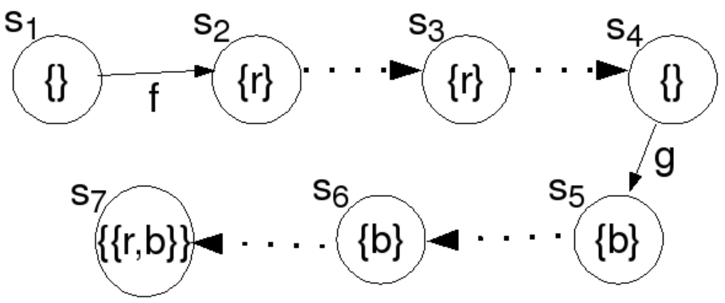

Figure 5.3: An automaton that represents the complete instance of n-tuple $(f(x,x),g(y,y))$

called $step/skip$-automata. In the sequel, without special notation, term "automaton" is used to mean $step/skip$-automaton and term "automata" is used to mean $step/skip$-automata.

Suppose after reading a ground n-tuple t, we reach a terminal state s_7 in Figure 5.3. At this time, we can only say that t is a ground substitution instance of the "linear version" of $(f(x,x),g(y,y))$ since we have not checked the equality constraints of the variables appearing in non-linear 2-tuple $(f(x,x),g(y,y))$. In order to address the equality constraints among the variables of $(f(x,x),g(y,y))$, we do the following:

1. We define $\mathcal{C}$ to be a finite set of colors.

2. We label each non-terminal state in our automaton with a set of colors. For example, in Figure 5.3, states s_2 and s_3 are labeled with set $\{r\}$, and states

s_5 and s_6 are labeled with set $\{b\}$.

3. We associate each terminal state with a set of sets of colors. For example, in Figure 5.3, terminal state s_7 is associated with a set of sets of colors $\{\{r,b\}\}$.

Basically, a color is treated as an atomic proposition:

- if ground n-tuple t satisfies c, then whenever two states that are reached in the same run (after scanning t) share a common color c, the ground terms that are read starting at those states or skipped at those states must be identical.

For example, in Figure 5.3, color r is satisfied by a ground n-tuple t whenever the ground terms of t that are skipped at state s_2 and s_3 are identical. Similarly, color b is satisfied by t whenever the ground terms of t that are skipped at state s_5 and s_6 are identical.

A set of colors is treated as a conjunction of atomic propositions and a set of sets of colors is treated as a disjunction of conjunctions. For example, in Figure 5.3, if ground n-tuple t satisfies colors r and b, then we say that t is a ground substitution instance of $(f(x,x),g(y,y))$ since the formula associated with terminal state s_7 is a set $\{\{r,b\}\}$.

In this chapter, we devise various operations on automata to implement the operations used in TOPDOWN algorithm. One operation is called disjunction, which can construct an automaton that represents the complete instances of a set of n-tuples. For example, suppose we have two automata $\mathcal{M}_1$ and $\mathcal{M}_2$ that represents the complete instances of 2-tuples $(f(x,x),g(y,z))$ and $(f(x,y),g(z,z))$

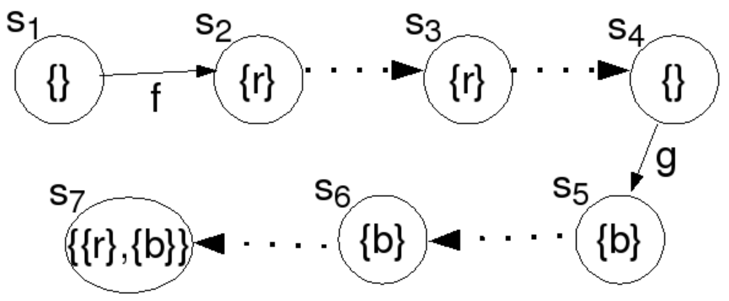

Figure 5.4: An automaton that represents the complete instances of n-tuples $\{(f(x,x),g(y,z)),(f(x,y),g(z,z))\}$

respectively, a disjunction of $\mathcal{M}_1$ and $\mathcal{M}_2$ that represents the complete instances of $\{(f(x,x),g(y,z)),(f(x,y),g(z,z))\}$ is shown in Figure 5.4. Note that, here we only show the resulting automaton of the disjunction operation, the details of the disjunction operation and all other operations will be discussed afterwards. The only difference between Figure 5.3 and Figure 5.4 is that the formula associated with terminal state s_7 in Figure 5.4 is $\{\{r\},\{b\}\}$ instead of $\{\{r,b\}\}$. That is, ground n-tuple t is a ground substitution instance of $\{(f(x,x),g(y,z)),(f(x,y),g(z,z))\}$ if we reach terminal state s_7 after reading t and t satisfies either color r or color b.

5.4 Notations

In this section, we present the notations used in this chapter. We omit parentheses and commas of any term $f(t_{\#_f},\dots,t_1)$ and denote it by $ft_{\#_f}\dots t_1$. Similarly, we omit parentheses and commas of an n-tuple and denote an n-tuple $(t_n, \ldots, t_1)$ by a sequence $t_n \ldots t_1$ of terms. For our purposes, it turns out to be more convenient as we traverse an n-tuple from left to right if we number the arguments of an n-tuple or a term in descending order. The ordering of these numbers corresponds to the way that we compute the positions of prefixes of an n-tuple, which will be introduced below.

In order to identify the terms and sub-terms in an n-tuple, we introduce the notation of positions. A *position* is a sequence of positive integers, and the empty position (sequence) is denoted by ε. We denote the set of all positions by $\textsc{Pos}$ and the set of nonempty positions by $\textsc{Pos}^+$. An ordering $\succ$ on $\textsc{Pos}^+$ is defined as follows: $m_1 m_2 \ldots m_k \succ n_1 n_2 \ldots n_l$ if and only if one of the following two conditions holds:

1. $m_1 m_2 \ldots m_k$ is a proper prefix of $n_1 n_2 \ldots n_l$.

2. $\exists i \geq 1$ such that $m_1 m_2 \ldots m_{i-1} = n_1 n_2 \ldots n_{i-1}$ and $m_i > n_i$.

By convention, we define $p \succ \varepsilon$ for all positions p in $\textsc{Pos}^+$.

Lemma 5.4.1 *The ordering $\succ$ on* $\textsc{Pos}$ *is a total ordering.*

Proof. Each position $n_1 \ldots n_k$ other than ε can be viewed as an infinite sequence $n_1 \ldots n_k \infty \ldots$, where ∞ is greater than all of the positive integers. Then the ordering defined above is exactly the restriction to sequences of this form of the lexicographic ordering on all infinite sequences formed from $\{1, 2, \ldots, \infty\}$. $\square$

We denote the concatenation of two positions p_1 and p_2 by $p_1\ p_2$. If $p_1 = m_1 \ldots m_k$ and $p_2 = n_1 \ldots n_l$, then $p_1\ p_2 = m_1 \ldots m_k\ n_1 \ldots n_l$. Note that, if $p_1 = \varepsilon$

then $p_1\ p_2 = p_2$. If position p is $p_1\ p_2$, then we say p_1 *is a prefix of* p and p_2 *is a suffix of* p. Moreover, we denote prefix p_1 of p by $p \setminus p_2$. We define function $next : \textsc{Pos}^+ \to \textsc{Pos}$ as follows:

1. $next(p) = q\ (k-1)$, provided $p = q\ k\ \underbrace{11\dots1}_{m}$ in which q is a position and $k > 1$ and $m \geq 0$.

2. $next(p) = \varepsilon$, provided $p = \underbrace{11\dots1}_{m}$ and $m > 0$.

Note that for all $p \in \textsc{Pos}^+$, we have $p \succ next(p)$ and $p \succ p\ \#_f$ ($\#_f > 0$). We claim the following facts about function *next* and notation $p \setminus p_2$.

Lemma 5.4.2 *If $p_1 \succeq p_2 \succ next(p_1)$ then $next(p_2) \succeq next(p_1)$.*

Proof. If $p_1 = p_2$ then we have $next(p_2) = next(p_1)$. Now suppose $p_1 \succ p_2$. In general, p_1 has the form $q\ k\underbrace{11\dots1}_{u}$, in which q is a position and $k > 1$ and $u \geq 0$. Then $next(p_1)$ is $q\ (k-1)$. Since $p_1 \succ p_2 \succ next(p_1)$, it follows that p_2 is $p_1\ q'\ h\underbrace{11\dots1}_{v}$, in which q' is a position and $h > 1$ and $v \geq 0$. Consequently, $next(p_2)$ is $p_1\ q'\ (h-1)$. Since $p_1\ q'\ (h-1) = q\ k\underbrace{11\dots1}_{u}\ q'\ (h-1) \succ q\ (k-1)$, $next(p_2) \succ next(p_1)$. □

Lemma 5.4.3 *Suppose p is a position and p_2 is a suffix of p. Then the following conditions all hold:*

1. *If p_3 is a position, then $(p\ p_3) \setminus (p_2\ p_3) = p \setminus p_2$.*

2. *If $next(p_2) \succ \varepsilon$, then $next(p) \setminus next(p_2) = p \setminus p_2$.*

3. *If $p_2 = \varepsilon$ or $next(p_2) = \varepsilon$, then $next(p \setminus p_2) = next(p)$.*

Proof. Suppose $p = p_1\, p_2$. Then $p \setminus p_2 = p_1$. Moreover, $p\, p_3 = p_1\, p_2\, p_3$. Thus,

$$(p\, p_3) \setminus (p_2\, p_3) = (p_1\, p_2\, p_3) \setminus (p_2\, p_3).$$

That is, $(p\, p_3) \setminus (p_2\, p_3) = p_1$. It follows that $(p\, p_3) \setminus (p_2\, p_3) = p \setminus p_2$. Thus, condition (1) holds.

Suppose $next(p_2) \succ \varepsilon$. Then p_2 has the form $k\, \alpha$ with some $k > 1$ and some position α. Since $p = p_1\, p_2$, $next(p) = p_1\, next(p_2)$. Thus, $next(p) \setminus next(p_2) = p_1$. If follows that if $next(p_2) \succ \varepsilon$ then $next(p) \setminus next(p_2) = p \setminus p_2$. Thus, condition (2) also holds.

Suppose $p_2 = \varepsilon$. Then $next(p) = next(p_1)$. Suppose $next(p_2) = \varepsilon$. Then p_2 has the form $11\ldots1$. Since $p = p_1\, p_2$, $next(p) = next(p_1)$. In either case, we have $next(p_1) = next(p \setminus p_2) = next(p)$. Thus, condition (3) holds as well. $\square$

Lemma 5.4.4 *Suppose p is a position with $p \succ \varepsilon$ and p_2 is a suffix of p. Then $next(p) \succeq next(p \setminus p_2)$.*

Proof. Suppose $p = p_1\, p_2$. If $p_2 = \varepsilon$ then $next(p) = next(p \setminus p_2)$. If $p_2 \succ \varepsilon$ then we have two cases:

1. $next(p_2) \succ \varepsilon$. Then $next(p) = p_1\, next(p_2)$. Since $p_1 \succ next(p_1)$, $next(p) \succ next(p_1) = next(p \setminus p_2)$.

2. $next(p_2) = \varepsilon$. Then $next(p) = next(p_1) = next(p \setminus p_2)$.

In summary, we always have $\mathit{next}(p) \succeq \mathit{next}(p \setminus p_2)$. $\square$

We now associate positions with prefixes of n-tuples. We use e to denote an empty prefix. Given PF the set of all prefixes of n-tuple $t_n \dots t_1$, we define function $pos_n : PF \rightarrow \textsc{Pos}$ as follows:

1. $pos_n(e) = n$.
2. $pos_n(\alpha f) = pos_n(\alpha)\,\#_f$, provided f is a function symbol.
3. $pos_n(\alpha c) = \mathit{next}(pos_n(\alpha))$, provided c is a constant.
4. $pos_n(\alpha x) = \mathit{next}(pos_n(\alpha))$, provided x is a variable.

We say *p is a position of n-tuple t* if there exists a prefix α of t such that $pos_n(\alpha) = p$. We say b is *the symbol of n-tuple t at position p*, and we write $b = t\ @\ p$, if αb is a prefix of n-tuple t with $pos_n(\alpha) = p$ and b is a symbol in Σ or V. We denote the set of all the positions of n-tuple t by $\textsc{Pos}(t)$.

Given a position p and a symbol $b \in \Sigma$, we define position $follow(p,b)$ to be:

$$follow(p,b) = \begin{cases} \varepsilon & \text{if } \#_b = 0 \text{ and } p = \varepsilon \\ \mathit{next}(p) & \text{if } \#_b = 0 \text{ and } p \succ \varepsilon \\ p\,\#_b & \text{if } \#_b > 0 \end{cases}$$

Note that for all $p \in \textsc{Pos}^+$ and all symbol $b \in \Sigma$, we have $p \succ follow(p,b)$.

The following facts are related to notation $follow(p,b)$.

Lemma 5.4.5 *Suppose p is a position with $p \succ \varepsilon$ and b is a symbol in Σ and p' is a suffix of p such that $follow(p',b) \succ \varepsilon$. Then $follow(p,b) \setminus follow(p',b) = p \setminus p'$.*

Proof. We consider the following cases:

1. $\#_b = 0$. Since $follow(p',b) \succ \varepsilon$, $p' \succ \varepsilon$. Moreover, $follow(p,b) = next(p)$ and $follow(p',b) = next(p')$. Since $follow(p',b) = next(p') \succ \varepsilon$, then by condition (2) of Lemma 5.4.3, $next(p) \setminus next(p') = p \setminus p'$.

2. $\#_b > 0$. Then $follow(p,b) = p\,b$ and $follow(p',b) = p'\,b$. By condition (1) of Lemma 5.4.3, $(p\,b) \setminus (p'\,b) = p \setminus p'$.

In either case, we have $follow(p,b) \setminus follow(p',b) = p \setminus p'$. □

Lemma 5.4.6 *Suppose p is a position with $p \succ \varepsilon$ and p_2 is a suffix of p and b is a symbol in Σ. Then the following conditions all hold:*

1. *If $follow(p_2,b) \succ \varepsilon$, then $follow(p,b) \succ next(p \setminus p_2)$.*

2. *If $follow(p_2,b) = \varepsilon$, then $follow(p,b) = next(p \setminus p_2)$.*

Proof. Suppose $p = p_1\,p_2$. We consider the following cases:

1. $\#_b = 0$. Then $follow(p,b) = next(p)$ and $follow(p_2,b) = next(p_2)$. There are two cases:

   (a) $follow(p_2,b) \succ \varepsilon$. Then $p_2 \succ next(p_2) \succ \varepsilon$. Moreover, $next(p) = p_1 \setminus next(p_2)$. Clearly, $follow(p,b) = next(p) \succ next(p_1) = next(p \setminus p_2)$. It follows that condition (1) holds in this case.

   (b) $follow(p_2,b) = \varepsilon$. This leaves two possibilities:

i. $p_2 = \varepsilon$. Then $p = p_1$ and $follow(p,b) = next(p) = next(p_1) = next(p \setminus p_2)$.

ii. $p_2 \succ \varepsilon$. Then $follow(p_2,b) = next(p_2) = \varepsilon$. Moreover, we have $follow(p,b) = next(p) = next(p_1) \setminus next(p_2) = next(p_1)$. That is, $follow(p,b) = next(p \setminus p_2)$.

In either case, condition (2) holds.

2. $\#_b > 0$. Then $follow(p_2,b) = p_2\ \#_b \succ \varepsilon$. Thus, Condition (2) holds vacuously in this case. Moreover, $follow(p,b) = p\ \#_b = p_1\ p_2\ \#_b \succ next(p_1) = next(p \setminus p_2)$. It follows that condition (1) holds in this case.

□

We define the *sub-term of a term* $t = ft_n \ldots t_1$ $(n > 0)$ *starting at nonempty position p* (denoted by $t \diamond p$) recursively as follows:

1. If p is the singleton sequence k, then $t \diamond p$ is defined if and only if $n \geq k$. In that case, $t \diamond p = t_k$.

2. If p has the form $k\ q$ ($q \in \textsc{Pos}^+$ and $k > 0$) then $t \diamond p$ is defined if and only if $n \geq k$ and $t_k \diamond q$ is defined. In that case, $t \diamond p = t_k \diamond q$.

Now we define the *sub-term of n-tuple t starting at nonempty position p* (denoted by $t \diamond p$) recursively as follows:

1. If p is the singleton sequence k, then $t \diamond p$ is defined if and only if t is $t_n t_{n-1} \ldots t_1$ with $n \geq k$. In that case, $t \diamond p = t_k$.

2. If p has the form $k\ q$ ($q \in \textsc{Pos}^+$ and $k > 0$) then $t \diamond p$ is defined if and only if t is $t_n t_{n-1} \ldots t_1$ with $n \geq k$ and $t_k \diamond q$ is defined. In that case, $t \diamond p = t_k \diamond q$.

For example, if $t = f\ x_1\ a\ g\ x_2$ (abbreviation of $(f(x_1,a),g(x_2)))$ then $t \diamond 2 = f\ x_1\ a$ and $t \diamond 21 = f\ x_1\ a \diamond 11 = a$.

5.5 Formal Definition

Formally, we define an $step/skip$-automaton $\mathcal{M}$ to be a tuple

$$(Q, s^0, \delta, \sigma, color, \tau)$$

as follows:

1. Q is a finite set of *states*.
2. $s^0 \in Q$ is the *initial state*.
3. $color : Q \rightarrow 2^C$ is a function that maps each state to a finite set of colors.
4. $\delta : Q \times \Sigma \nrightarrow Q$ is a partial function called the *step function* that takes a state and a symbol b in Σ to the next state by reading b. The states on which function δ is defined with at least one symbol in Σ are called *step states*. We denote the set of all the step states in Q by Q_δ. Note that, for a step state s, $\delta(s,b)$ may not be defined for all the symbols b in Σ.
5. $\sigma : Q \nrightarrow Q$ is a partial function called the *skip function* that takes a state to the next state by skipping a complete term in the input. The states on which

function σ is defined are called *skip states*. We denote the set of all the skip states in Q by Q_σ.

6. The states on which no transition function is defined are called *terminal states*. We denote the set of all the terminal states in Q by Q_T.

7. Q_δ and Q_σ and Q_T are pairwise disjoint sets.

8. τ is a mapping that assigns to each terminal state in Q a set of sets of colors. We also call $\tau(t)$ *the acceptance condition of t*. The acceptance condition $\emptyset$ corresponds to **F**, i.e., the equality constraint is not satisfiable. The acceptance condition $\{\emptyset\}$ corresponds to **T**, i.e., the equality constraint is trivially satisfiable.

9. The *underlying directed graph* of automaton $\mathcal{M}$ is defined to be the directed graph $G = (Q,E)$ such that $E = \{(s_1,s_2) \mid s_1 \in Q_\sigma \text{ and } \sigma(s_1) = s_2\} \cup \{(s_1,s_2) \mid s_1 \in Q_\delta \text{ and } \delta(s_1,b) = s_2 \text{ with some symbol b in } \Sigma\}$. We require that the underlying directed graph of $\mathcal{M}$ must be acyclic.

The requirement that an automaton $\mathcal{M}$ must be acyclic is very important. In the following sections, we will discuss various operations on automata, which are performed by traversing the underlying directed graphs from the initial states and following the edges. By enforcing this restriction, we can avoid going into cycles while we perform those operations on automata.

We define a *path* in automaton $\mathcal{M}$ to be a sequence of states $s_1,\ldots,s_n$ such that $s_1 = s^0$ and for any integer i in $[1,n-1]$, there is an edge from s_i to s_{i+1} in the underlying directed graph of $\mathcal{M}$. Since all automata are acyclic, any path in an automaton is a finite sequence. We say a state s' in automaton $\mathcal{M}$ *is reachable from* another state s if s' is reachable from s in the underlying directed graph of $\mathcal{M}$. In that case, we say s' is a *successor* of s and s is a *predecessor* of s'.

Suppose t is a ground n-tuple and automaton $\mathcal{M}$ is defined to be a tuple $(Q, s^0, \delta, \sigma, color, \tau)$. We define the *partial computation* of $\mathcal{M}$ on t to be a mapping $C_{\mathcal{M},t}$ that maps the positions of some prefix of t to states in Q such that the following conditions all hold:

1. $C_{\mathcal{M},t}(n) = s^0$.

2. Suppose $C_{\mathcal{M},t}(p)$ is a skip state s with some position p of t and $t\ @\ p = b$. Suppose further, q is the greatest position of t with $C_{\mathcal{M},t}(q) = s$. Then we have:

   - $C_{\mathcal{M},t}(follow(p,b)) = s$ if $follow(p,b) \succ next(q)$.
   - $C_{\mathcal{M},t}(follow(p,b)) = \sigma(s)$ if $follow(p,b) = next(q)$.

3. Suppose $C_{\mathcal{M},t}(p)$ is a step state s with some position p of t and $t\ @\ p = b$. Then we have:

   - $C_{\mathcal{M},t}(follow(p,b)) = \delta(s,b)$ if $\delta(s,b)$ is defined.
   - $C_{\mathcal{M},t}(follow(p,b))$ is not defined if $\delta(s,b)$ is not defined.

We say $C_{\mathcal{M},t}$ is *a computation of $\mathcal{M}$ on t* if $C_{\mathcal{M},t}$ maps all the positions in $\mathrm{Pos}(t)$ to Q. If $C_{\mathcal{M},t}$ is a computation of $\mathcal{M}$ on t and $C_{\mathcal{M},t}(\varepsilon) \in Q_T$, then we say $C_{\mathcal{M},t}$ is *terminating*. In the sequel, we denote the greatest position p of t with $C_{\mathcal{M},t}(p) = s$ by $p^s_{\mathcal{M},t}$.

Lemma 5.5.1 *Suppose $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is an automaton. Then the following condition holds for all positions p of a ground n-tuple t:*

- *If $C_{\mathcal{M},t}(p) = s$ is a step state, then $p = p^s_{\mathcal{M},t}$.*

Proof. We verify this by induction on positions p of t that the stated condition holds for p. Suppose $p = n$. Then $C_{\mathcal{M},t}(p) = s^0$ and $p = p^{s^0}_{\mathcal{M},t}$. Clearly, the stated condition holds for position n of t.

Suppose p is a position of t such that $n \succeq p \succ \varepsilon$ and the stated condition holds for p and $t \ @ \ p = b \in \Sigma$. We show that the stated condition also holds for position $follow(p,b)$ of t. Suppose $C_{\mathcal{M},t}(p) = s$. We consider the following cases:

1. s is a step state. By induction, $p = p^s_{\mathcal{M},t}$. By the definition of partial computation, if $\delta(s,b)$ is not defined, then $C_{\mathcal{M},t}(follow(p,b))$ is not defined, and if $\delta(s,b)$ is defined, then $C_{\mathcal{M},t}(follow(p,b)) = \delta(s,b) = s'$. In the former case, the stated condition holds vacuously. In the latter case, $follow(p,b) = p^{s'}_{\mathcal{M},t}$. Thus, the stated condition holds for position $follow(p,b)$ of t.

2. s is a skip state. Then there are two cases:

   - $follow(p,b) \succ next(p^s_{\mathcal{M},t})$. Then by the definition of partial computation, $C_{\mathcal{M},t}(follow(p,b)) = s$. Since s is still a skip state, the stated condition holds vacuously in this case.
   - $follow(p,b) = next(p^s_{\mathcal{M},t})$. Then by the definition of partial computation, $C_{\mathcal{M},t}(follow(p,b)) = \sigma(s) = s'$. In this case, $follow(p,b) = p^{s'}_{\mathcal{M},t}$. Thus, the stated condition holds for position $follow(p,b)$ of t.

□

Given an automaton $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$, we say a color *c occurs at a non-terminal state s* if $c \in color(s)$, and we say color *c occurs at a terminal state s* if c is contained in some set of $\tau(s)$. We denote the set of all the colors that occur at the states of automaton $\mathcal{M}$ by $C^{\mathcal{M}}$. Suppose t is a ground n-tuple such that $C_{\mathcal{M},t}$ is a computation of $\mathcal{M}$ on t. We say ground n-tuple *t satisfies a color c with respect to* $\mathcal{M}$ if and only if for all positions p and q of t such that $p = p^{s_1}_{\mathcal{M},t} \succ \varepsilon$ and $q = p^{s_2}_{\mathcal{M},t} \succ \varepsilon$ and c occurs at states s_1 and s_2, we have $t \diamond p = t \diamond q$. If t satisfies color c with respect to $\mathcal{M}$, then we write $t \models_{\mathcal{M}} c$. We say ground n-tuple *t satisfies a set C of colors with respect to* $\mathcal{M}$ if and only if t satisfies all the colors in C with respect to $\mathcal{M}$, and we write $t \models_{\mathcal{M}} C$. We say ground n-tuple *t satisfies an acceptance condition* $S = \{C_1, ..., C_m\}$ *with respect to* $\mathcal{M}$ if and only if there exists at least one integer $i \in [1, m]$ with $t \models_{\mathcal{M}} C_i$, and we write $t \models_{\mathcal{M}} S$.

We say $\mathcal{M}$ *accepts* a ground n-tuple t if $C_{\mathcal{M},t}$ is terminating and in addition $t \models_{\mathcal{M}} \tau(C_{\mathcal{M},t}(\varepsilon))$. An automaton $\mathcal{M}_1$ is *equivalent to* another automaton $\mathcal{M}_2$ if and only if $\mathcal{M}_1$ and $\mathcal{M}_2$ accept the same set of ground n-tuples.

5.6 Automaton for An N-Tuple

We say automaton $\mathcal{M}$ *is for n-tuple t* if $\mathcal{M}$ accepts and only accepts all the ground substitution instances of t.

To obtain an automaton for an n-tuple t, we shall construct an automaton

$\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ such that each state in $\mathcal{M}$ is a position of t. The formal definition of $\mathcal{M}$ is given as follows:

1. $s^0 = n$.

2. Q is the least subset of Pos that contains s^0 and is closed under skip and step functions, as described in the sequel.

3. p is a step state if $t \ @ \ p = b \in \Sigma$. In that case, $color(p) = \emptyset$ and $\delta(p,b) = follow(p,b)$ and $\delta(p,b')$ is not defined for all symbol $b' \in \Sigma$ such that $b' \neq b$.

4. p is a skip state if $t \ @ \ p = x \in V$. In that case, $color(p) = \{c_x\}$ and $\sigma(p) = next(p)$.

5. ε is the terminal state of $\mathcal{M}$ and $\tau(\varepsilon) = \{\{c_{x_1}, \ldots, c_{x_k}\}\}$ in which $x_1, \ldots, x_k$ are all the variables appearing in t.

For example, an automaton $\mathcal{M}$ that is for 2-tuple $fxxgyy$ with $\#_f = \#_g = 2$ is shown in Figure 5.5.

Formally, we say an n-tuple t' is *a linear version* of an n-tuple t, and we write $t' = \underline{t}$, if t' is obtained from t as follows:

1. For all position p of t, if $t \ @ \ p = b \in \Sigma$, then $t' \ @ \ p = b$.

2. For all position p of t, if $t \ @ \ p = x \in V$, then $t' \ @ \ p = x_p$, where x_p is a fresh variable.

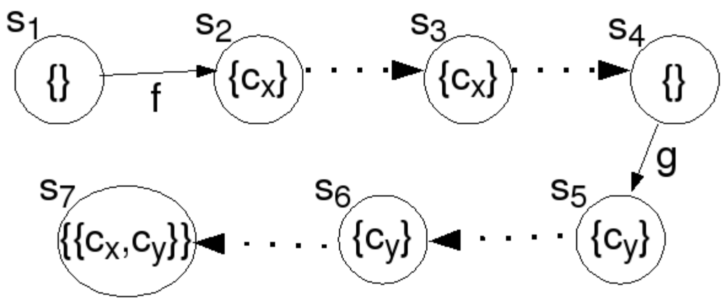

Figure 5.5: An automaton for n-tuple *fxxgyy*

For example, $fxgyz$ is an 2-tuple with $\#_f = 1$ and $\#_g = 2$. Then a linear version of $fxgyz$ is $fx_{21}gx_{12}x_{11}$. Clearly, each variable in a linear version of an n-tuple occurs only once. Moreover, a ground substitution instance of t is also a ground substitution instance of $\underline{t}$, but the reverse may not be true.

Lemma 5.6.1 *Suppose μ is an n-tuple and t is a ground substitution instance of $\underline{\mu}$. If for all positions p and q of μ such that $\mu\ @\ p = \mu\ @\ q = x \in V$, we have $t \diamond p = t \diamond q$, then t is a ground substitution instance of μ.*

Proof. Since t is a ground substitution instance of $\underline{\mu}$, by the definition of $\underline{\mu}$, for all positions p of μ such that $\mu\ @\ p = b \in \Sigma$, we have $t\ @\ p = \underline{\mu}\ @\ p = \mu\ @\ p = b$. Suppose for all positions p and q of μ such that $\mu\ @\ p = \mu\ @\ q = x \in V$, we have $t \diamond p = t \diamond q$. Then it is clear that t is a ground substitution instance of μ since t satisfies all the equality constraints in μ. □

Suppose p is a position of t and α is the prefix of t with $pos_n(\alpha) = p$. We denote the prefix α by $t \leftarrow p$ (read as "t up to p"). If $t \leftarrow p = \alpha$ and $t \mathbin{@} p = b \in \Sigma$, then $t \leftarrow follow(p,b) = \alpha b$. For example, if $t = f\, x_1\; a\; g\, x_2$ (that is the abbreviation of $(f(x_1,a),g(x_2)))$, then we have $pos_2(e) = 2$ and $t \leftarrow 2 = e$, $pos_2(f) = 22$ and $t \leftarrow 22 = f$, ..., $pos_2(p\, x_1\; a\; g\, x_2) = \varepsilon$ and $t \leftarrow \varepsilon = f\, x_1\; a\; g\, x_2$. Now we claim the following facts about the above construction.

Lemma 5.6.2 *Suppose $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is an automaton that is obtained by applying the construction mentioned in the beginning of this section with respect to an n-tuple μ. Suppose further, t is a ground n-tuple. Then the following conditions hold for all the positions p of t:*

1. *$C_{\mathcal{M},t}(p)$ is defined and is equal to q if and only if $t \leftarrow q$ is a ground substitution instance of $\underline{\mu} \leftarrow q$.*

2. *If $C_{\mathcal{M},t}(p) = q$ and q is not a skip state then $p = q$.*

Proof. We verify this by induction on positions p of ground n-tuple t that the stated conditions all hold for p. Suppose $p = n$. Then $C_{\mathcal{M},t}(n) = n$. Clearly, the stated conditions all hold in this case.

Suppose $p \succ \varepsilon$ is a position of t such that the stated conditions all hold for p and $t \mathbin{@} p = b \in \Sigma$. We show that the stated conditions all hold for position $follow(p,b)$ of t. Suppose $C_{\mathcal{M},t}(p)$ is defined and is equal to q. We consider the following cases:

1. q is a step state of $\mathcal{M}$. By condition (2) of the induction hypothesis, $p = q$. Then we have that $C_{\mathcal{M},t}(follow(p,b))$ is defined if and only if $\mu \mathbin{@} p = b$

(equivalently, $\delta(p,b)$ is defined). By condition (1) of the induction hypothesis, $t \leftarrow p$ is a ground substitution instance of $\underline{\mu} \leftarrow p$. If $\mu \ @ \ p = b$, then $t \leftarrow follow(p,b)$ is a ground substitution instance of $\underline{\mu} \leftarrow follow(p,b)$. Thus, $C_{\mathcal{M},t}(follow(p,b))$ is defined if and only if $t \leftarrow follow(p,b)$ is a ground substitution instance of $\underline{\mu} \leftarrow follow(p,b)$. Thus, condition (1) holds in this case. Suppose $C_{\mathcal{M},t}(follow(p,b))$ is defined. Then

$$C_{\mathcal{M},t}(follow(p,b)) = \delta(p,b) = follow(p,b).$$

Clearly, condition (2) also holds in this case.

2. q is a skip state of $\mathcal{M}$. Then there are two cases:

   (a) $follow(p,b) \succ next(q)$. Then $C_{\mathcal{M},t}(follow(p,b))$ is defined if and only if $C_{\mathcal{M},t}(p)$ is defined. By condition (1) of the induction hypothesis, condition (1) still holds in this case. Since $C_{\mathcal{M},t}(p) = q$, $C_{\mathcal{M},t}(follow(p,b))$ is defined and is equal to q. Since q is still a skip state, condition (2) holds vacuously in this case.

   (b) $follow(p,b) = next(q)$. Since $C_{\mathcal{M},t}(p) = q$, $C_{\mathcal{M},t}(follow(p,b)) = next(q)$. Since $follow(p,b) = next(q)$, condition (2) also holds in this case. By condition (1) of the induction hypothesis, $t \leftarrow q$ is a ground substitution instance of $\underline{\mu} \leftarrow q$. If $C_{\mathcal{M},t}(follow(p,b))$ is defined and is equal to $next(q)$, then $t \leftarrow follow(p,b)$ is a ground substitution instance of $\underline{\mu} \leftarrow next(q) = \underline{\mu} \leftarrow follow(p,b)$ since the complete ground sub-term $t \diamond q$ is skipped at skip state q and $t \leftarrow follow(p,b) = (t \leftarrow q)(t \diamond q)$. Conversely, if $t \leftarrow follow(p,b)$ is a ground substitution instance of $\underline{\mu} \leftarrow next(q) = \underline{\mu} \leftarrow follow(p,b)$, then b is the last symbol of sub-term $t \diamond q$ that is skipped by skip state q. Thus, $C_{\mathcal{M},t}(follow(p,b))$ is defined and is equal to $next(q) = follow(p,b)$. It follows that condition (1) also holds in this case.

□

Theorem 5.6.3 *Suppose $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is an automaton that is obtained by applying the construction mentioned in the beginning of this section with respect to an n-tuple μ. Then $\mathcal{M}$ is an automaton for n-tuple μ.*

Proof. Suppose t is a ground n-tuple that is a ground substitution instance of μ. Then $t \leftarrow \varepsilon$ is a ground substitution instance of $\underline{\mu} \leftarrow \varepsilon$. By condition (1) of Lemma 5.6.2, $C_{\mathcal{M},t}(\varepsilon)$ is defined and is equal to ε. By the definition of $\mathcal{M}$, ε is the terminal state of $\mathcal{M}$. Thus, $C_{\mathcal{M},t}(\varepsilon)$ is terminating. Since t is a ground substitution instance of μ, for all color c_x in $\{c_x \mid x \text{ is a variable appearing in } \mu\}$, we have $t \models_{\mathcal{M}} c_x$. Moreover, $\tau(\varepsilon) = \{\{c_{x_1}, \ldots, c_{x_k}\}\}$ in which $x_1, \ldots, x_k$ are all the variables appearing in t. It follows that $t \models_{\mathcal{M}} \tau(\varepsilon)$. Thus, t is accepted by $\mathcal{M}$.

Conversely, suppose t is a ground n-tuple that is accepted by $\mathcal{M}$. Then $C_{\mathcal{M},t}(\varepsilon)$ is terminating and is equal to ε. By condition (1) of Lemma 5.6.2, $t \leftarrow \varepsilon = t$ is a ground substitution instance of $\underline{\mu} \leftarrow \varepsilon = \underline{\mu}$. Moreover, $\tau(\varepsilon) = \{\{c_{x_1}, \ldots, c_{x_k}\}\}$ in which $x_1, \ldots, x_k$ are all the variables appearing in t. Since t is accepted by $\mathcal{M}$, $t \models_{\mathcal{M}} \tau(\varepsilon)$. That is, for all color c_x in $\{c_x \mid x \text{ is a variable appearing in } \mu\}$, we have

$t \models_{\mathcal{M}} c_x$. Thus, for all positions p and q of μ such that $\mu \mathbin{@} p = \mu \mathbin{@} q = x \in V$, we have $t \diamond p = t \diamond q$. By Lemma 5.6.1, t is a ground substitution instance of μ. $\square$

Chapter 6

Conjunction and Disjunction of Automata

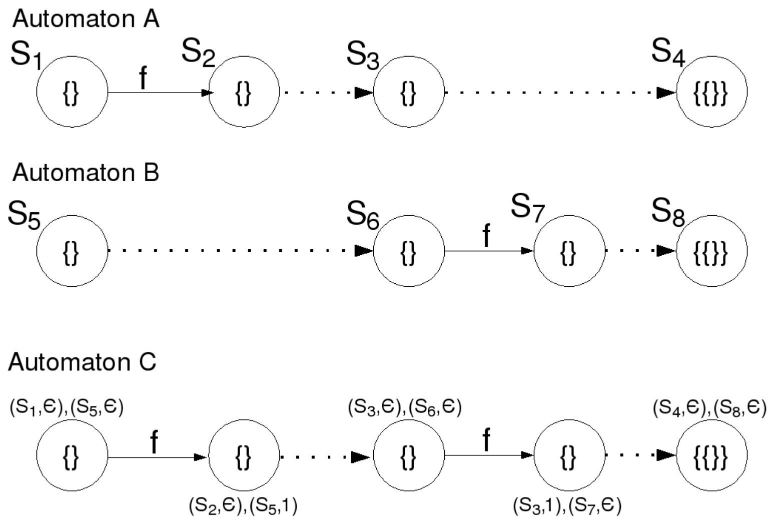

Figure 6.1: Example of conjunction

6.1 Conjunction of Automata

We say an automaton $\mathcal{M}$ is *a conjunction* of a collection of automata $\{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$, if $\mathcal{M}$ accepts a set S of ground n-tuples such that $S = S_1 \cap \ldots \cap S_m$ in which S_1 and $\ldots$ and S_m are sets of ground n-tuples accepted by $\mathcal{M}_1$ and $\ldots$ and $\mathcal{M}_m$ respectively.

Informally, to obtain a conjunction of a collection of automata, we shall construct an automaton $\mathcal{M}$ such that each state in $\mathcal{M}$ is a set $\{(s_i, p_i) \mid i \in [1,m]\}$ such that:

- s_1 and $\ldots$ and s_m are states from $\mathcal{M}_1$ and $\mathcal{M}_m$ respectively.
- For all $i \in [1,m]$, p_i is *the relative position* of s_i. At least one of p_i is ε. Suppose we give all the automata the same input and reach a state

$\{(s_i, p_i) \mid i \in [1,m]\}$ and the next input is a symbol $b \in \Sigma$. Then for all the states s_i with $p_i = \varepsilon$, s_i is a state in $\mathcal{M}_i$ that reads b (if s_i is a step state) or skips a complete sub-term whose first symbol is b (if s_i is a skip state). For all the states s_j with $p_j \succ \varepsilon$, s_j is a skip state in $\mathcal{M}_j$ that skips a complete term t such that $t \ @ \ p_j = b$. That is, the relative positions tells us how to "synchronize" the states from all the components to read the same input.

For example, two automata A and B are shown in Figure 6.1 with $\#_f = 1$. Automaton C is a conjunction of those two automata.

Formally, suppose $\{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$ is a collection of automata such that

$$\mathcal{M}_i = (Q_i, s_i^0, \delta_i, \sigma_i, color_i, \tau_i)$$

for all $i \in [1,m]$. We assume that for all $i, j \in [1,m]$ with $i \neq j$, $C^{\mathcal{M}_i}$ and $C^{\mathcal{M}_j}$ are disjoint sets. This assumption can be implemented by replacing each color u appearing in $\mathcal{M}_i$ with u^i and each color v appearing in $\mathcal{M}_j$ with v^j, where i and j are called *tags*. Now we construct $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ as follows:

1. $s^0 = \{(s_i^0, \varepsilon) \mid i \in [1,m]\}$.

2. Q is the least subset of $(Q_1 \times \textsc{Pos}) \times \ldots \times (Q_m \times \textsc{Pos})$ that contains s^0 and is closed under skip and step functions, as described in the sequel.

3. If $s = \{(s_i, p_i) \mid i \in [1,m]\}$ is non-terminal state in $\mathcal{M}$, then $color(s) = \bigcup\{color_i(s_i) \mid i \in [1,m] \text{ and } p_i = \varepsilon\}$. Note that, by our assumption, for all $i, j \in [1,m]$ with $i \neq j$, $color_i(s_i)$ and $color_j(s_j)$ are disjoint sets.

4. $s = \{(s_i, p_i) \mid i \in [1,m]\}$ is a step state in $\mathcal{M}$ if there is at least one $i \in [1,m]$ such that s_i is a step state. Suppose b is a symbol in Σ. If $\delta_i(s_i, b)$ is not defined for some step state s_i with $i \in [1,m]$, then $\delta(s,b)$ is not defined, otherwise $\delta(s,b)$ is $\{(s'_i, p'_i) \mid i \in [1,m]\})$ such that:

   - If s_i is a step state, then $p'_i = \varepsilon$ and $s'_i = \delta_i(s_i, b)$.
   - If s_i is a skip state and $follow(p_i, b) \succ \varepsilon$, then $p'_i = follow(p_i, b)$ and $s'_i = s_i$.
   - If s_i is a skip state and $follow(p_i, b) = \varepsilon$, then $p'_i = \varepsilon$ and $s'_i = \sigma_i(s_i)$.

5. $s = \{(s_i, p_i) \mid i \in [1,m]\}$ is a skip state in $\mathcal{M}$ if s_i is a skip state for all $i \in [1,m]$. In that case, $\sigma(s)$ is $\{(s'_i, p'_i) \mid i \in [1,m]\}$ such that:

   - If either $p_i = \varepsilon$ or $next(p_i) = \varepsilon$, then $p'_i = \varepsilon$ and $s'_i = \sigma_i(s_i)$.
   - If $next(p_i) \succ \varepsilon$, then $p'_i = next(p_i)$ and $s'_i = s_i$.

6. A terminal state s in $\mathcal{M}$ has the form $\{(s_i, p_i) \mid i \in [1,m]\}$ such that at least one s_i is a terminal state. In that case, if for all $i \in [1,m]$, s_i is a terminal state, then $\tau(s)$ is

$$\{C_1 \cup \ldots \cup C_m \mid C_1 \in \tau_1(s_1) \text{ and } \ldots \text{ and } C_m \in \tau_m(s_m)\},$$

   otherwise, $\tau(s) = \emptyset$.

We claim the following facts about the above construction.

Lemma 6.1.1 *Suppose automaton $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is obtained by applying the above construction on a collection of automata $\{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$, where*

$$\mathcal{M}_i = (Q_i, s_i^0, \delta_i, \sigma_i, color_i, \tau_i)$$

for all $i \in [1,m]$. Then for all the states $\{(s_i, p_i) \mid i \in [1,m]\}$ of $\mathcal{M}$ that are reachable from s^0, if s_i is not a skip state for some $i \in [1,m]$, then $p_i = \varepsilon$.

Proof. We verify this by induction on states s of $\mathcal{M}$ that are reachable from s^0 that if s has the form $\{(s_i, p_i) \mid i \in [1,m]\}$ in which s_i is not a skip state with some $i \in [1,m]$, then $p_i = \varepsilon$. Suppose $s = s^0 = \{(s_i^0, \varepsilon) \mid i \in [1,m]\}$. Then the stated condition holds for s.

Suppose s is a non-terminal state of $\mathcal{M}$ such that s is a successor of s^0 and the stated condition holds for s. We show that the stated condition also holds for the states s' that are directly reachable from s (via a step or skip function). We consider the following cases:

1. $s = \{(s_i, p_i) \mid i \in [1,m]\}$ is a skip state. Then by the definition of $\mathcal{M}$, for all $i \in [1,m]$, s_i is a skip state. In that case, $s' = \sigma(s) = \{(s_i', p_i') \mid i \in [1,m]\}$. For all $i \in [1,m]$, there are two cases:

    - Either $p_i = \varepsilon$ or $next(p_i) = \varepsilon$. Then $p_i' = \varepsilon$ and $s_i' = \sigma_i(s_i)$.
    - $next(p_i) \succ \varepsilon$. Then $p_i' = next(p_i)$ and $s_i' = s_i$.

    For all $i \in [1,m]$, if $p_i' = \varepsilon$ then $s_i' = \sigma_i(s_i)$ could be any kind of state (step, or skip, or terminal), otherwise $s_i' = s_i$ is still a skip state. Thus, the stated condition also holds for state s'.

2. $s = \{(s_i, p_i) \mid i \in [1,m]\}$ is a step state. Then by the definition of $\mathcal{M}$, there exists at least one step state s_i with some $i \in [1,m]$. Suppose b is a symbol in Σ such that $\delta_i(s_i,b)$ is defined for all step state s_i with some $i \in [1,m]$. Then $s' = \delta(s,b) = \{(s'_i, p'_i) \mid i \in [1,m]\})$. For all $i \in [1,m]$, there are three cases:

   - s_i is a step state. Then $p'_i = \varepsilon$ and $s'_i = \delta_i(s_i,b)$.
   - s_i is a skip state and $follow(p_i,b) \succ \varepsilon$. Then $p'_i = follow(p_i,b)$ and $s'_i = s_i$.
   - s_i is a skip state and $follow(p_i,b) = \varepsilon$. Then $p'_i = follow(p_i,b) = \varepsilon$ and $s'_i = \sigma_i(s_i)$.

   For all $i \in [1,m]$, if $p'_i = \varepsilon$ then s'_i is either $\delta_i(s_i,b)$ (if s_i is a step state), or $\sigma_i(s_i)$ (if s_i is a skip state). In either case, s'_i could be any kind of state (step, or skip, or terminal). For all $i \in [1,m]$, if $p'_i \succ \varepsilon$, then $s'_i = s_i$ is still a skip state. Thus, the stated condition also holds for state s'.

□

Lemma 6.1.2 *Suppose automaton $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is obtained by applying the above construction on a collection of automata $\{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$, where*

$$\mathcal{M}_i = (Q_i, s_i^0, \delta_i, \sigma_i, color_i, \tau_i)$$

for all $i \in [1,m]$. Then the following condition holds for all positions p of a ground n-tuple t:

- *$C_{\mathcal{M},t}(p)$ is defined and is equal to a state $s = \{(s_i, p_i) \mid i \in [1,m]\}$ in $\mathcal{M}$ if and only if for all $i \in [1,m]$, $C_{\mathcal{M}_i,t}(p)$ is defined and is equal to s_i and $p^s_{\mathcal{M},t} \setminus p_i = p^{s_i}_{\mathcal{M}_i,t}$.*

Proof. By induction on positions p of a ground n-tuple t. If $p = n$, then $C_{\mathcal{M},t}(n) = s = s^0 = \{(s^0_i, \varepsilon) \mid i \in [1,m]\} = \{(C_{\mathcal{M}_i,t}(n), \varepsilon) \mid i \in [1,m]\}$. The stated condition holds in this case.

Suppose $p \succ \varepsilon$ is a position of t such that the stated condition holds and $t \ @ \ p = b$ with some symbol $b \in \Sigma$. We show that the stated condition also holds for position $follow(p,b)$ of t. Suppose $C_{\mathcal{M},t}(p) = s = \{(s_i, p_i) \mid i \in [1,m]\}$. By induction, for all $i \in [1,m]$, $C_{\mathcal{M}_i,t}(p) = s_i$ and $p^s_{\mathcal{M},t} \setminus p_i = p^{s_i}_{\mathcal{M}_i,t}$. We consider the following cases:

1. s is a skip state. In this case, $C_{\mathcal{M},t}(follow(p,b))$ is always defined. By the definition of $\mathcal{M}$, for all $i \in [1,m]$, s_i is a skip state. There are two cases:
   - $follow(p,b) \succ next(p^s_{\mathcal{M},t})$. In this case,

     $$C_{\mathcal{M},t}(follow(p,b)) = s.$$

     Since $p^s_{\mathcal{M},t} \succeq p \succ \varepsilon$, by Lemma 5.4.4,

     $$next(p^s_{\mathcal{M},t}) \succeq next(p^s_{\mathcal{M},t} \setminus p_i) = next(p^{s_i}_{\mathcal{M}_i,t})$$

     for all $i \in [1,m]$. It follows that for all $i \in [1,m]$, $follow(p,b) \succ next(p^{s_i}_{\mathcal{M}_i,t})$ and $C_{\mathcal{M}_i,t}(follow(p,b)) = s_i$. The stated condition still holds in this case.

- $follow(p,b) = next(p^{s}_{\mathcal{M},t})$. Then $C_{\mathcal{M},t}(follow(p,b)) = \sigma(s) = s'$ and $follow(p,b) = p^{s'}_{\mathcal{M},t}$. Suppose $s' = \{(s'_i, p'_i) \mid i \in [1,m]\}$. Then for all $i \in [1,m]$, there are two possibilities:

  (a) Either $p_i = \varepsilon$ or $next(p_i) = \varepsilon$. By condition (3) of Lemma 5.4.3,

  $$next(p^{s}_{\mathcal{M},t} \setminus p_i) = next(p^{s_i}_{\mathcal{M}_i,t}) = next(p^{s}_{\mathcal{M},t}) = follow(p,b).$$

  Thus, in this case, we have $C_{\mathcal{M}_i,t}(follow(p,b)) = \sigma_i(s_i) = s'_i$ and $p'_i = \varepsilon$. Moreover, $p^{s'}_{\mathcal{M},t} \setminus p'_i = next(p^{s_i}_{\mathcal{M}_i,t}) = p^{s'_i}_{\mathcal{M}_i,t}$.

  (b) $next(p_i) \succ \varepsilon$. Since $next(p_i) \succ \varepsilon$,

  $$follow(p,b) = next(p^{s_i}_{\mathcal{M}_i,t}\, p_i) = p^{s_i}_{\mathcal{M}_i,t}\, next(p_i).$$

  It follows that $follow(p,b) \succ next(p^{s_i}_{\mathcal{M}_i,t})$. Thus,

  $$C_{\mathcal{M}_i,t}(follow(p,b)) = s_i = s'_i$$

  and $p'_i = next(p_i)$. Moreover,

  $$p^{s'}_{\mathcal{M},t} \setminus p'_i = p^{s'}_{\mathcal{M},t} \setminus next(p_i) = p^{s_i}_{\mathcal{M}_i,t} = p^{s'_i}_{\mathcal{M}_i,t}.$$

Thus, the stated condition still holds in either case.

2. s is a step state. Then by the definition of $\mathcal{M}$, there exists at least one

step state s_i with some $i \in [1,m]$. Since s is a step state, by Lemma 5.5.1, $p = p^{s}_{\mathcal{M},t}$ and $p \setminus p_i = p^{s_i}_{\mathcal{M}_i,t}$ for all $i \in [1,m]$. If $C_{\mathcal{M},t}(follow(p,b))$ is defined, then $\delta_i(s_i,b)$ is defined for all step state s_i with some $i \in [1,m]$. Thus, $C_{\mathcal{M}_i,t}(follow(p,b))$ is defined for all $i \in [1,m]$. Conversely, if we have $C_{\mathcal{M}_i,t}(follow(p,b))$ is defined for all $i \in [1,m]$, then $\delta_i(s_i,b)$ is defined for all step state s_i with some $i \in [1,m]$. Thus, $C_{\mathcal{M},t}(follow(p,b))$ is defined and is equal to $\delta(s,b)$. Suppose $C_{\mathcal{M},t}(follow(p,b)) = \delta(s,b) = s'$. Then $follow(p,b) = p^{s'}_{\mathcal{M},t}$. Suppose $s' = \delta(s,b) = \{(s'_i,p'_i) \mid i \in [1,m]\})$. Then for all $i \in [1,m]$, there are three cases:

(a) s_i is a step state. Then $p'_i = p_i$ and $s'_i = \delta_i(s_i,b)$. By Lemma 6.1.1, $p'_i = p_i = \varepsilon$. Thus, $p = p^{s}_{\mathcal{M},t} = p^{s_i}_{\mathcal{M}_i,t}$. In this case, $C_{\mathcal{M}_i,t}(follow(p,b)) = \delta_i(s_i,b) = s'_i$. Moreover, we have $p^{s'}_{\mathcal{M},t} \setminus p'_i = follow(p,b) = p^{s'_i}_{\mathcal{M}_i,t}$.

(b) s_i is a skip state and $follow(p_i,b) \succ \varepsilon$. Since $follow(p_i,b) \succ \varepsilon$,

$$follow(p,b) = follow(p^{s_i}_{\mathcal{M}_i,t}\ p_i,b) \succ next(p^{s_i}_{\mathcal{M}_i,t}).$$

It follows that $C_{\mathcal{M}_i,t}(follow(p,b)) = s_i = s'_i$ and $p'_i = follow(p_i,b)$. By Lemma 5.4.5, $p^{s'}_{\mathcal{M},t} \setminus p'_i = follow(p,b) \setminus follow(p_i,b) = p \setminus p_i = p^{s_i}_{\mathcal{M}_i,t} = p^{s'_i}_{\mathcal{M}_i,t}$.

(c) s_i is a skip state and $follow(p_i,b) = \varepsilon$. Since $follow(p_i,b) = \varepsilon$, by condition (2) of Lemma 5.4.6,

$$follow(p,b) = follow(p^{s_i}_{\mathcal{M}_i,t}\ p_i,b) = next(p^{s_i}_{\mathcal{M}_i,t}).$$

It follows that $C_{\mathcal{M}_i,t}(follow(p,b)) = \sigma_i(s_i) = s_i'$ and $p_i' = \varepsilon$. Since $p_i' = \varepsilon$, $p^{s'}_{\mathcal{M},t} \setminus p_i' = follow(p,b) = next(p^{s_i}_{\mathcal{M}_i,t}) = p^{s_i'}_{\mathcal{M}_i,t}$.

Thus, the stated condition still holds in all the above cases.

□

Lemma 6.1.3 *Suppose automaton $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is obtained by applying the above construction on a collection of automata $\{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$, where*

$$\mathcal{M}_i = (Q_i, s_i^0, \delta_i, \sigma_i, color_i, \tau_i)$$

for all $i \in [1,m]$. Suppose further that $C_{\mathcal{M},t}$ is a computation of $\mathcal{M}$ on ground n-tuple t and c is a color. Then $t \models_{\mathcal{M}} c$ if and only if $t \models_{\mathcal{M}_i} c$ for all $i \in [1,m]$ such that $c \in C^{\mathcal{M}_i}$.

Proof. Suppose $t \models_{\mathcal{M}} c$. Then for all positions p and q of t such that $p = p^{s_1}_{\mathcal{M},t} \succ \varepsilon$ and $q = p^{s_2}_{\mathcal{M},t} \succ \varepsilon$ and c occurs at states s_1 and s_2, we have $t \diamond p = t \diamond q$. Since c appears in $\tau(C_{\mathcal{M},t}(\varepsilon))$, by the definition of $\mathcal{M}$, c occurs at the states of automaton $\mathcal{M}_i$ with some $i \in [1,m]$. Since c occurs at $s_1 = \{(s_j^1, p_j^1) \mid j \in [1,m]\}$ and $s_2 = \{(s_j^2, p_j^2) \mid j \in [1,m]\}$, c occurs at s_i^1 with $p_i^1 = \varepsilon$ and s_i^2 with $p_i^2 = \varepsilon$. By Lemma 6.1.2, $p^{s_i^1}_{\mathcal{M}_i,t} = p^{s_1}_{\mathcal{M},t} = p$ and $p^{s_i^2}_{\mathcal{M}_i,t} = p^{s_2}_{\mathcal{M},t} = q$. That is, $t \diamond p^{s_i^1}_{\mathcal{M}_i,t} = t \diamond p^{s_i^2}_{\mathcal{M}_i,t}$. It follows that $t \models_{\mathcal{M}_i} c$.

Conversely, Suppose $t \models_{\mathcal{M}_i} c$ with some $i \in [1,m]$. Then for all positions p and q of t such that $p = p^{s_i^1}_{\mathcal{M}_i,t} \succ \varepsilon$ and $q = p^{s_i^2}_{\mathcal{M}_i,t} \succ \varepsilon$ and c occurs at states s_i^1 and s_i^2, we have $t \diamond p = t \diamond q$. Since $C_{\mathcal{M},t}$ is a computation of $\mathcal{M}$ on t, by Lemma 6.1.2, there exist states s_1 and s_2 in $\mathcal{M}$ such that $C_{\mathcal{M},t}(p') = s_1 = \{(s^1_j, p^1_j) \mid j \in [1,m]\}$ in which $p^1_i = \varepsilon$ and $C_{\mathcal{M},t}(q') = s_2 = \{(s^2_j, p^2_j) \mid j \in [1,m]\}$ in which $p^2_i = \varepsilon$. By the definition of $\mathcal{M}$, c occurs at s_1 and s_2. Again, by Lemma 6.1.2, $p^{s_1}_{\mathcal{M},t} = p^{s^1_i}_{\mathcal{M}_i,t} = p$ and $p^{s_2}_{\mathcal{M},t} = p^{s^2_i}_{\mathcal{M}_i,t} = q$. That is, $t \diamond p^{s_1}_{\mathcal{M},t} = t \diamond p^{s_2}_{\mathcal{M},t}$. It follows that $t \models_{\mathcal{M}} c$. $\square$

Theorem 6.1.4 *Suppose $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is obtained by applying the above construction on a collection of automata $\{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$, where*

$$\mathcal{M}_i = (Q_i, s^0_i, \delta_i, \sigma_i, color_i, \tau_i)$$

for all $i \in [1,m]$. Then $\mathcal{M}$ is a conjunction of $\{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$.

Proof. Suppose t is a ground n-tuple that is accepted by $\mathcal{M}_1$ and … and $\mathcal{M}_m$. Then for all $i \in [1,m]$, $C_{\mathcal{M}_i,t}(\varepsilon)$ is a terminal state s_i and $t \models_{\mathcal{M}_i} \tau_i(s_i)$. By Lemma 6.1.1 and Lemma 6.1.2, $C_{\mathcal{M},t}(\varepsilon)$ is defined and is equal to $s = \{(s_i, \varepsilon) \mid i \in [1,m]\}$. Clearly, $C_{\mathcal{M},t}$ is terminating. By the definition of $\mathcal{M}$, $\tau(s)$ is

$$\{C_1 \cup \ldots \cup C_m \mid C_1 \in \tau_1(s_1) \text{ and } \ldots \text{ and } C_m \in \tau_m(s_m)\}.$$

Since $t \models_{\mathcal{M}_i} \tau_i(s_i)$ for all $i \in [1,m]$, there exists a set $C = C_1 \cup \ldots \cup C_m$ such that $C_i \in \tau_i(s_i)$ and $t \models_{\mathcal{M}_i} C_i$ for all $i \in [1,m]$. Following Lemma 6.1.3, we have $t \models_{\mathcal{M}} C$. Since C is a set in $\tau(s)$, $t \models_{\mathcal{M}} \tau(s)$. It follows that t is also accepted by $\mathcal{M}$.

Conversely, suppose t is a ground n-tuple that is accepted by $\mathcal{M}$. Then $C_{\mathcal{M},t}(\varepsilon)$ is defined and is equal to a terminal state s and $t \models_{\mathcal{M}} \tau(s)$. By Lemma 6.1.2, $s = \{(s_i, \varepsilon) \mid i \in [1,m]\}$ and for all $i \in [1,m]$, $C_{\mathcal{M}_i,t}(\varepsilon) = s_i$. Since $t \models_{\mathcal{M}} \tau(s)$,

$\tau(s) \neq \emptyset$. It follows that s_i is a terminal state for all $i \in [1,m]$. Clearly, for all $i \in [1,m]$, $C_{\mathcal{M}_i,t}(\varepsilon)$ is terminating. By the definition of $\mathcal{M}$, $\tau(s)$ is

$$\{C_1 \cup \ldots \cup C_m \mid C_1 \in \tau_1(s_1) \text{ and } \ldots \text{ and } C_m \in \tau_m(s_m)\}.$$

Since $t \models_{\mathcal{M}} \tau(s)$, there exists a set $C = C_1 \cup \ldots \cup C_m$ such that $C_i \in \tau_i(s_i)$ for all $i \in [1,m]$ and $t \models_{\mathcal{M}} C$. Following Lemma 6.1.3, we have $t \models_{\mathcal{M}_i} C_i$ for all $i \in [1,m]$. Since C_i is a set in $\tau_i(s_i)$, $t \models_{\mathcal{M}_i} \tau_i(s_i)$ for all $i \in [1,m]$. It follows that t is also accepted by $\mathcal{M}_1$ and ... and $\mathcal{M}_m$. $\square$

6.2 Disjunction of Automata

We say an automaton $\mathcal{M}$ is *a disjunction* of a collection of automata $\{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$, if $\mathcal{M}$ accepts a set S of ground n-tuples such that $S = S_1 \cup \ldots, \cup S_m$ in which S_1 and ... and S_m are sets of ground n-tuples accepted by $\mathcal{M}_1$ and ... and $\mathcal{M}_m$ respectively.

In the sequel, we denote a subset of $\{1,...,m\}$ by I. Informally, to obtain a disjunction of a collection of automata, we shall construct an automaton $\mathcal{M}$ such that each state in $\mathcal{M}$ is a set $\{(s_i, p_i) \mid i \in I\}$ such that s_i is a state in $\mathcal{M}_i$ and p_i is the relative position of s_i that has the same meaning as it is used in the conjunction operation. Unlike the conjunction operation, a state of $\mathcal{M}$ may have fewer than m pairs. If (s_i, p_i) with some $i \in [1,m]$ does not appear in a state of $\mathcal{M}$, then it means that $\mathcal{M}_i$ has (at some previous point) failed to execute a transition at a step state.

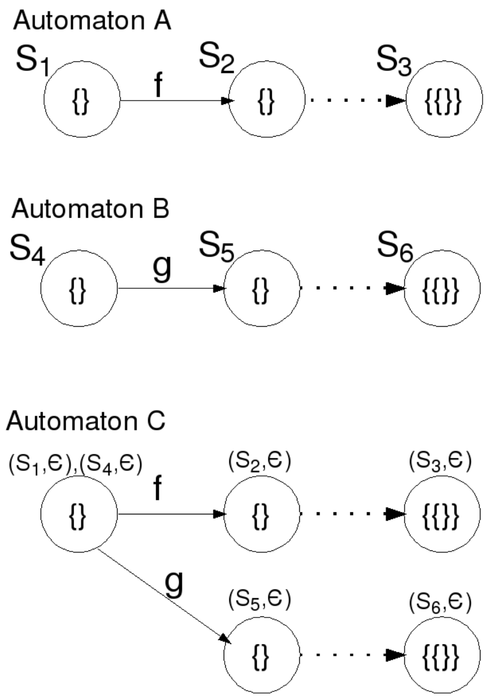

Figure 6.2: Example of disjunction

For example, two automata A and B are shown in Figure 6.2 with $\#_f = \#_g = 1$. If we build a disjunction (automaton C shown in Figure 6.2) of those two automata, then we have a sequence of composite states $\{(s_1,\varepsilon),(s_4,\varepsilon)\}$,$\{(s_2,\varepsilon)\}$, and $\{(s_3,\varepsilon)\}$ that read a ground substitution instance of n-tuple fx. Similarly, we have a sequence of composite states $\{(s_1,\varepsilon),(s_4,\varepsilon)\}$,$\{(s_5,\varepsilon)\}$, and $\{(s_6,\varepsilon)\}$ that read a ground substitution instance of n-tuple gx. In the composite state $\{(s_2,\varepsilon)\}$, we only have one pair because step state s_4 from another automaton cannot execute a transition to read symbol f. A similar situation happens in the composite state $\{(s_5,\varepsilon)\}$.

Formally, suppose $\{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$ is a collection of automata such that

$$\mathcal{M}_i = (Q_i, s_i^0, \delta_i, \sigma_i, color_i, \tau_i)$$

for all $i \in [1,m]$. We assume that for all $i, j \in [1,m]$ with $i \neq j$, $C^{\mathcal{M}_i}$ and $C^{\mathcal{M}_j}$ are disjoint sets. We have already shown how to implement this assumption in the conjunction operation. Now we construct $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ as follows:

1. $s^0 = \{(s_i^0, \varepsilon) \mid i \in [1,m]\}$.

2. Q is a set whose elements are indexed sets $\{(s_i, p_i) \mid i \in I\}$ in which s_i is a state in $\mathcal{M}_i$ and p_i is a position.

3. If $s = \{(s_i, p_i) \mid i \in I\}$ $(I \neq \emptyset)$ is a non-terminal state in $\mathcal{M}$, then $color(s) = \bigcup\{color_i(s_i) \mid i \in I \text{ and } p_i = \varepsilon\}$. Note that by our assumption, for all $i, j \in I$ with $i \neq j$, $color_i(s_i)$ and $color_j(s_j)$ are disjoint sets.

4. $s = \{(s_i, p_i) \mid i \in I\}$ $(I \neq \emptyset)$ is a step state in $\mathcal{M}$ if s_i is a step state for at least one $i \in I$. Suppose b is a symbol in Σ and I^b denotes the following set

   $$\{i \mid i \in I \text{ and either } s_i \text{ is a skip state or } \delta_i(s_i, b) \text{ is defined}\}.$$

   If $I^b = \emptyset$ then $\delta(s,b)$ is not defined, otherwise $\delta(s,b) = \{(s_i', p_i') \mid i \in I^b\})$ such that for all $i \in I^b$, we have:

   - If s_i is a step state such that $\delta_i(s_i, b)$ is defined, then $p_i' = \varepsilon$ and $s_i' = \delta_i(s_i, b)$.

- If s_i is a skip state and $follow(p_i,b) \succ \varepsilon$, then $p_i' = follow(p_i,b)$ and $s_i' = s_i$.
- If s_i is a skip state and $follow(p_i,b) = \varepsilon$, then $p_i' = \varepsilon$ and $s_i' = \sigma_i(s_i)$.

5. $s = \{(s_i,p_i) \mid i \in I\}$ ($I \neq \emptyset$) is a skip state in $\mathcal{M}$ if $I \neq \emptyset$ and s_i is a skip state for all $i \in I$. In that case, $\sigma(s)$ is $\{(s_i',p_i') \mid i \in I\}$ such that:
   - If either $p_i = \varepsilon$ or $next(p_i) = \varepsilon$, then $p_i' = \varepsilon$ and $s_i' = \sigma_i(s_i)$.
   - If $next(p_i) \succ \varepsilon$, then $p_i' = next(p_i)$ and $s_i' = s_i$.

6. A terminal state s in $\mathcal{M}$ has the form $\{(s_i,p_i) \mid i \in I\}$ such that either $I = \emptyset$ or there is at least one terminal state s_i with some $i \in I$. If $I = \emptyset$ then $\tau(s) = \emptyset$, otherwise $\tau(s)$ is

$$\bigcup\{\tau_i(s_i) \mid i \in I \text{ and } s_i \text{ is a terminal state}\}.$$

We claim the following facts about the above construction.

Lemma 6.2.1 *Suppose automaton $\mathcal{M} = (Q,s^0,\delta,\sigma,color,\tau)$ is obtained by applying the above construction on a collection of automata $\{\mathcal{M}_1,\ldots,\mathcal{M}_m\}$, where*

$$\mathcal{M}_i = (Q_i,s_i^0,\delta_i,\sigma_i,color_i,\tau_i)$$

for all $i \in [1,m]$. Then for all the states $\{(s_i,p_i) \mid i \in I\}$ of $\mathcal{M}$ that are reachable from s^0, if s_i is not a skip state for some $i \in I$, then $p_i = \varepsilon$.

Proof. We verify this by induction on states s of $\mathcal{M}$ that are reachable from s^0

that if s has the form $\{(s_i, p_i) \mid i \in I\}$ and s_i is not a skip state with some $i \in I$, then $p_i = \varepsilon$. Suppose $s = s^0 = \{(s_i^0, \varepsilon) \mid i \in [1, m]\}$. Then the stated condition holds for s.

Suppose s is a non-terminal state of $\mathcal{M}$ such that s is a successor of s^0 and the stated condition holds for s. We show that the stated condition also holds for the states s' that are directly reachable from s (via a step or skip function). We consider the following cases:

1. $s = \{(s_i, p_i) \mid i \in I\}$ is a skip state. Then by the definition of $\mathcal{M}$, $I \neq \emptyset$ and for all $i \in I$, s_i is a skip state. In that case, $s' = \sigma(s) = \{(s'_i, p'_i) \mid i \in I\}$. For all $i \in I$, there are two cases:

   - Either $p_i = \varepsilon$ or $next(p_i) = \varepsilon$. Then $p'_i = \varepsilon$ and $s'_i = \sigma_i(s_i)$.
   - $next(p_i) \succ \varepsilon$. Then $p'_i = next(p_i)$ and $s'_i = s_i$.

   For all $i \in I$, if $p'_i = \varepsilon$ then $s'_i = \sigma_i(s_i)$ could be any kind of state (step, or skip, or terminal), otherwise $s'_i = s_i$ is still a skip state. Thus, the stated condition also holds for state s'.

2. $s = \{(s_i, p_i) \mid i \in I\}$ is a step state. Then by the definition of $\mathcal{M}$, there exists at least one step state s_i with some $i \in I$. Suppose b is a symbol in Σ. Then $s' = \delta(s, b) = \{(s'_i, p'_i) \mid i \in I^b\})$, where I^b is not empty and is defined to be

   $$\{i \mid i \in I \text{ and either } s_i \text{ is a skip state or } \delta_i(s_i, b) \text{ is defined}\}.$$

   Moreover, for all $i \in I$, there are four cases:

- s_i is a step state and $\delta(s_i, b)$ is not defined. Then i is not in I^b. The pair (s_i, p_i) has no successor in s'.
- s_i is a step state and $\delta(s_i, b)$ is defined. Then $p'_i = \varepsilon$ and $s'_i = \delta_i(s_i, b)$.
- s_i is a skip state and $follow(p_i, b) \succ \varepsilon$. Then $p'_i = follow(p_i, b)$ and $s'_i = s_i$.
- s_i is a skip state and $follow(p_i, b) = \varepsilon$. Then $p'_i = follow(p_i, b) = \varepsilon$ and $s'_i = \sigma_i(s_i)$.

For all $i \in I^b$, if $p'_i = \varepsilon$ then s'_i is either $\delta_i(s_i, b)$ (if s_i is a step state with $\delta(s_i, b)$ is defined), or $\sigma_i(s_i)$ (if s_i is a skip state). In that case, s'_i could be any kind of state (step, or skip, or terminal). For all $i \in I^b$, if $p'_i \succ \varepsilon$, then $s'_i = s_i$ is still a skip state. Thus, the stated condition also holds for state s'.

$\square$

Lemma 6.2.2 *Suppose automaton $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is obtained by applying the above construction on a collection of automata $\{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$, where*

$$\mathcal{M}_i = (Q_i, s_i^0, \delta_i, \sigma_i, color_i, \tau_i)$$

for all $i \in [1, m]$. Suppose t is a ground n-tuple and set $I(t, p)$ is defined to be $\{i \mid i \in [1, m] \text{ and } C_{\mathcal{M}_i,t}(p) \text{ is defined}\}$. *Then the following condition holds for all positions p of ground n-tuple t:*

- *$C_{\mathcal{M},t}(p)$ is defined and is equal to $\{(s_i, p_i) \mid i \in I(t, p)\}$ if and only if $I(t, p) \neq \emptyset$ and for all $i \in I(t, p)$, $C_{\mathcal{M}_i,t}(p)$ is equal to s_i and $p^s_{\mathcal{M},t} \setminus p_i = p^{s_i}_{\mathcal{M}_i,t}$.*

Proof. By induction on positions p of ground n-tuple t. If $p = n$, then $C_{\mathcal{M},t}(n) = s = s^0 = \{(s_i^0, \varepsilon) \mid i \in [1,m]\} = \{(C_{\mathcal{M}_i,t}(n), \varepsilon) \mid i \in [1,m]\}$. The stated condition holds in this case.

Suppose $p \succ \varepsilon$ is a position of t such that the stated condition holds and $t \mathbin{@} p = b$ with some symbol $b \in \Sigma$. We show that the stated condition also holds for position $follow(p,b)$ of t. Suppose $C_{\mathcal{M},t}(p) = s = \{(s_i, p_i) \mid i \in I(t,p)\}$. By induction, $I(t,p) \neq \emptyset$ and for all $i \in I(t,p)$, $C_{\mathcal{M}_i,t}(p) = s_i$ and $p^s_{\mathcal{M},t} \setminus p_i = p^{s_i}_{\mathcal{M}_i,t}$. We consider the following cases:

1. $s = \{(s_i, p_i) \mid i \in I(t,p)\}$ is a skip state. In this case, $C_{\mathcal{M},t}(follow(p,b))$ is always defined. By the definition of $\mathcal{M}$, for all $i \in I(t,p)$, s_i is a skip state. There are two cases:

   - $follow(p,b) \succ next(p^s_{\mathcal{M},t})$. In this case,

     $$C_{\mathcal{M},t}(follow(p,b)) = s.$$

     Since $p^s_{\mathcal{M},t} \succeq p \succ \varepsilon$, by Lemma 5.4.4,

     $$next(p^s_{\mathcal{M},t}) \succeq next(p^s_{\mathcal{M},t} \setminus p_i) = next(p^{s_i}_{\mathcal{M}_i,t})$$

     for all $I(t,p)$. It follows that for all $i \in I(t,p)$,

     $$follow(p,b) \succ next(p^{s_i}_{\mathcal{M}_i,t})$$

     and $C_{\mathcal{M}_i,t}(follow(p,b)) = s_i$. The stated condition still holds in this case.

- $follow(p,b) = next(p^s_{\mathcal{M},t})$. Then $C_{\mathcal{M},t}(follow(p,b)) = \sigma(s) = s'$ and $follow(p,b) = p^{s'}_{\mathcal{M},t}$. Suppose $s' = \{(s'_i, p'_i) \mid i \in I(t,p)\}$. Then for all $i \in I(t,p)$, there are two possibilities:

  (a) Either $p_i = \varepsilon$ or $next(p_i) = \varepsilon$. By condition (3) of Lemma 5.4.3,

  $$next(p^s_{\mathcal{M},t} \setminus p_i) = next(p^{s_i}_{\mathcal{M}_i,t}) = next(p^s_{\mathcal{M},t}) = follow(p,b).$$

  Thus, in this case, we have $C_{\mathcal{M}_i,t}(follow(p,b)) = \sigma_i(s_i) = s'_i$ and $p'_i = \varepsilon$. Moreover, $p^{s'}_{\mathcal{M},t} \setminus p'_i = next(p^{s_i}_{\mathcal{M}_i,t}) = p^{s'_i}_{\mathcal{M}_i,t}$.

  (b) $next(p_i) \succ \varepsilon$. Since $next(p_i) \succ \varepsilon$,

  $$follow(p,b) = next(p^{s_i}_{\mathcal{M}_i,t}\, p_i) = p^{s_i}_{\mathcal{M}_i,t}\, next(p_i).$$

  It follows that $follow(p,b) \succ next(p^{s_i}_{\mathcal{M}_i,t})$. Thus,

  $$C_{\mathcal{M}_i,t}(follow(p,b)) = s_i = s'_i$$

  and $p'_i = next(p_i)$. Moreover,

  $$p^{s'}_{\mathcal{M},t} \setminus p'_i = p^{s'}_{\mathcal{M},t} \setminus next(p_i) = p^{s_i}_{\mathcal{M}_i,t} = p^{s'_i}_{\mathcal{M}_i,t}.$$

  Thus, the stated condition still holds in either case.

2. $s = \{(s_i, p_i) \mid i \in I(t,p)\}$ is a step state. Then by the definition of $\mathcal{M}$, there

exists at least one step state s_i with some $i \in I(t,p)$. Since s is a step state, by Lemma 5.5.1, $p = p^{s}_{\mathcal{M},t}$ and $p \setminus p_i = p^{s_i}_{\mathcal{M}_i,t}$ for all $i \in I(t,p)$. Suppose b is in Σ and $I^b(t,p)$ is

$$\{i \mid i \in I(t,p) \text{ and either } s_i \text{ is a skip state or } \delta_i(s_i,b) \text{ is defined}\}.$$

Suppose $C_{\mathcal{M},t}(follow(p,b))$ is defined. Then $I^b(t,p) \neq \emptyset$ and $\delta_i(s_i,b)$ is defined for all the step states s_i with some $i \in I^b(t,p)$. Thus, we have that $C_{\mathcal{M}_i,t}(follow(p,b))$ is defined for all $i \in I^b(t,p)$. Conversely, suppose $I^b(t,p) \neq \emptyset$ and $C_{\mathcal{M}_i,t}(follow(p,b))$ is defined for all $i \in I^b(t,p)$. Then $\delta_i(s_i,b)$ is defined for all the step state s_i with some $i \in I^b(t,p)$. Thus, $C_{\mathcal{M},t}(follow(p,b))$ is defined and is equal to $\delta(s,b)$. Suppose $I^b(t) \neq \emptyset$ and $C_{\mathcal{M},t}(follow(p,b)) = \delta(s,b) = s'$. Then $follow(p,b) = p^{s'}_{\mathcal{M},t}$. Suppose $s' = \delta(s,b) = \{(s'_i, p'_i) \mid i \in I^b(t)\})$. Then for all $i \in I^b(t)$, there are three cases:

(a) s_i is a step state and $\delta_i(s_i,b)$ is defined. Then $p'_i = p_i$ and $s'_i = \delta_i(s_i,b)$. By Lemma 6.2.1, $p'_i = p_i = \varepsilon$. Thus, $C_{\mathcal{M}_i,t}(follow(p,b)) = \delta_i(s_i,b) = s'_i$. Since $p'_i = \varepsilon$, $p^{s'}_{\mathcal{M},t} \setminus p'_i = follow(p,b) = p^{s'_i}_{\mathcal{M}_i,t}$.

(b) s_i is a skip state and $follow(p_i,b) \succ \varepsilon$. Since $follow(p_i,b) \succ \varepsilon$,

$$follow(p,b) = follow(p^{s_i}_{\mathcal{M}_i,t}\ p_i, b) \succ next(p^{s_i}_{\mathcal{M}_i,t}).$$

It follows that $C_{\mathcal{M}_i,t}(follow(p,b)) = s_i = s'_i$ and $p'_i = follow(p_i,b)$.

By Lemma 5.4.5, $p^{s'}_{\mathcal{M},t} \setminus p'_i = follow(p,b) \setminus follow(p_i,b) = p \setminus p_i = p^{s_i}_{\mathcal{M}_i,t} = p^{s'_i}_{\mathcal{M}_i,t}$.

(c) s_i is a skip state and $follow(p_i,b) = \varepsilon$. Since $follow(p_i,b) = \varepsilon$, by condition (2) of Lemma 5.4.6,

$$follow(p,b) = follow(p^{s_i}_{\mathcal{M}_i,t}\ p_i,b) = next(p^{s_i}_{\mathcal{M}_i,t}).$$

It follows that $C_{\mathcal{M}_i,t}(follow(p,b)) = \sigma_i(s_i) = s'_i$ and $p'_i = \varepsilon$. Since $p'_i = \varepsilon$, $p^{s'}_{\mathcal{M},t} \setminus p'_i = follow(p,b) = next(p^{s_i}_{\mathcal{M}_i,t}) = p^{s'_i}_{\mathcal{M}_i,t}$.

Thus, the stated condition still holds in the above cases.

□

Lemma 6.2.3 *Suppose automaton $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is obtained by applying the above construction on a collection of automata $\{\mathcal{M}_1, \dots, \mathcal{M}_m\}$, where*

$$\mathcal{M}_i = (Q_i, s^0_i, \delta_i, \sigma_i, color_i, \tau_i)$$

for all $i \in [1,m]$. Suppose further that $C_{\mathcal{M},t}$ is a computation of $\mathcal{M}$ on ground n-tuple t and c is a color. Let $I(t,\varepsilon) = \{i \mid i \in [1,m]$ and $C_{\mathcal{M}_i,t}(\varepsilon)$ is defined$\}$. Then $t \models_{\mathcal{M}} c$ if and only if there exists some $i \in I(t,\varepsilon)$ with $t \models_{\mathcal{M}_i} c$.

Proof. Suppose $t \models_{\mathcal{M}} c$. Then for all positions p and q of t such that $p = p^{s_1}_{\mathcal{M},t} \succ \varepsilon$ and $q = p^{s_2}_{\mathcal{M},t} \succ \varepsilon$ and c occurs at states s_1 and s_2, we have $t \diamond p = t \diamond q$. Since c appears in $\tau(C_{\mathcal{M},t}(\varepsilon))$, by the definition of $\mathcal{M}$, c occurs at the states of automaton $\mathcal{M}_i$ with some $i \in I(t,\varepsilon)$. Since c occurs at $s_1 = \{(s_j^1, p_j^1) \mid j \in I^1\}$ and $s_2 = \{(s_j^2, p_j^2) \mid j \in I^2\}$, by the definition of $\mathcal{M}$, c occurs at s_i^1 with $p_i^1 = \varepsilon$ and s_i^2 with $p_i^2 = \varepsilon$. By Lemma 6.2.2, $p_{\mathcal{M}_i,t}^{s_i^1} = p_{\mathcal{M},t}^{s_1} = p$ and $p_{\mathcal{M}_i,t}^{s_i^2} = p_{\mathcal{M},t}^{s_2} = q$. That is, $t \diamond p_{\mathcal{M}_i,t}^{s_i^1} = t \diamond p_{\mathcal{M}_i,t}^{s_i^2}$. It follows that $t \models_{\mathcal{M}_i} c$.

Conversely, Suppose $t \models_{\mathcal{M}_i} c$ with some $i \in I(t,\varepsilon)$. Then for all positions p and q of t such that $p = p_{\mathcal{M}_i,t}^{s_i^1} \succ \varepsilon$ and $q = p_{\mathcal{M}_i,t}^{s_i^2} \succ \varepsilon$ and c occurs at states s_i^1 and s_i^2, we have $t \diamond p = t \diamond q$. Since $C_{\mathcal{M},t}$ is a computation of $\mathcal{M}$ on t, by Lemma 6.2.2, there exist states s_1 and s_2 in $\mathcal{M}$ such that $C_{\mathcal{M},t}(p') = s_1 = \{(s_j^1, p_j^1) \mid j \in I^1\}$ in which $p_i^1 = \varepsilon$ and $C_{\mathcal{M},t}(q') = s_2 = \{(s_j^2, p_j^2) \mid j \in I^2\}$ in which $p_i^2 = \varepsilon$ and $i \in I^1 \cap I^2$. By the definition of $\mathcal{M}$, c occurs at s_1 and s_2. Again, by Lemma 6.2.2, $p_{\mathcal{M},t}^{s_1} = p_{\mathcal{M}_i,t}^{s_i^1} = p$ and $p_{\mathcal{M},t}^{s_2} = p_{\mathcal{M}_i,t}^{s_i^2} = q$. That is, $t \diamond p_{\mathcal{M},t}^{s_1} = t \diamond p_{\mathcal{M},t}^{s_2}$. It follows that $t \models_{\mathcal{M}} c$. □

Theorem 6.2.4 *Suppose $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is obtained by applying the above construction on a collection of automata $\{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$, where*

$$\mathcal{M}_i = (Q_i, s_i^0, \delta_i, \sigma_i, color_i, \tau_i)$$

for all $i \in [1,m]$. Then $\mathcal{M}$ is a disjunction of $\{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$.

Proof. Let t be a ground n-tuple and $I(t,p)$ be

$$\{i \mid i \in [1,m] \text{ and } C_{\mathcal{M}_i,t}(p) \text{ is defined}\}.$$

Suppose t is accepted by $\mathcal{M}_1$ or $\ldots$ or $\mathcal{M}_m$. Then $I(t,\varepsilon)$ contains at least one integer $i \in [1,m]$ and $C_{\mathcal{M}_i,t}(\varepsilon)$ is a terminal state s_i and $t \models_{\mathcal{M}_i} \tau_i(s_i)$. By Lemma 6.2.2, $C_{\mathcal{M},t}(\varepsilon)$ is defined and is equal to $s = \{(s_j,\varepsilon) \mid j \in I(t,\varepsilon)\}$. Since s_i is a terminal state, by the definition of $\mathcal{M}$, s is a terminal state of $\mathcal{M}$. Thus, $C_{\mathcal{M},t}$ is terminating. By the definition of $\mathcal{M}$, $\tau(s)$ is

$$\bigcup\{\tau_j(s_j) \mid j \in I(t,\varepsilon) \text{ and } s_j \text{ is a terminal state}\}.$$

Since $t \models_{\mathcal{M}_i} \tau_i(s_i)$, following Lemma 6.2.3, $t \models_{\mathcal{M}} \tau_i(s_i)$. Thus, $t \models_{\mathcal{M}} \tau(s)$. It follows that t is also accepted by $\mathcal{M}$.

Conversely, suppose t is a ground n-tuple that is accepted by $\mathcal{M}$. Then $C_{\mathcal{M},t}(\varepsilon)$ is defined and is equal to s and $t \models_{\mathcal{M}} \tau(s)$. By Lemma 6.2.2, $I(t,\varepsilon) \neq \emptyset$ and $s = \{(s_j,\varepsilon) \mid j \in I(t,\varepsilon)\}$ and for all $j \in I(t,\varepsilon)$, $C_{\mathcal{M}_j,t}(\varepsilon) = s_j$. Since s is a terminal state, there is at least one terminal state s_i with some $i \in I(t,\varepsilon)$. Clearly, $C_{\mathcal{M}_i,t}(\varepsilon)$ is terminating. By the definition of $\mathcal{M}$, $\tau(s)$ is

$$\bigcup\{\tau_j(s_j) \mid j \in I(t,\varepsilon) \text{ and } s_j \text{ is a terminal state}\}.$$

Since $t \models_{\mathcal{M}} \tau(s)$, following Lemma 6.2.3, there exists at least one integer i in $I(t,\varepsilon)$ such that $t \models_{\mathcal{M}} \tau_i(s_i)$ and $t \models_{\mathcal{M}_i} \tau_i(s_i)$, i.e., t is accepted by $\mathcal{M}_i$. It follows that t is also accepted by $\mathcal{M}_1$ or ... or $\mathcal{M}_m$. $\square$

Chapter 7

Normalization of Automata

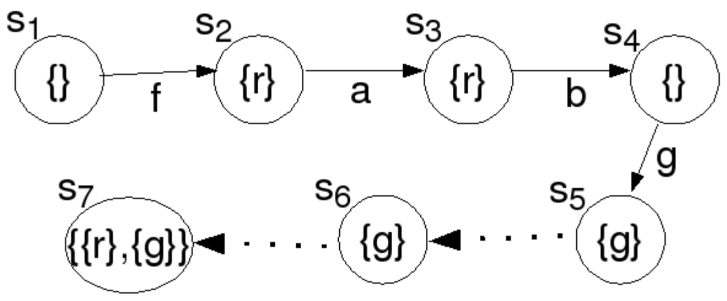

Figure 7.1: An automaton that contains unsatisfiable colors

7.1 Definition of Normalized Automata

We say an automaton $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is *normalized* if for all step states s in Q, $color(s) = \emptyset$. Normalized automata have the following advantages:

- There are no "unsatisfiable" colors in the acceptance conditions of a normalized automaton. A color is *unsatisfiable with respect to acceptance condition* $\tau(s)$ if there does not exist any ground n-tuple t such that $C_{\mathcal{M},t}(\varepsilon) = s$ and $t \models_{\mathcal{M}} c$. For example, in the automaton shown in Figure 7.1, color r is unsatisfiable since $a \neq b$.
- In this chapter, we present the following operations on automata: *conjunction*, *disjunction*, *grouping*, *ungrouping*, *expansion* and *projection*. Those are used to implement the operations in TopDown algorithm and they require as a condition of correctness that the automata are normalized.

7.2 Eliminating a Single Color from Step States

To transform an arbitrary automaton to a normalized one, we have to eliminate the colors on those step states and make appropriate changes so that the resulting automaton is equivalent to the original one. We first consider how to safely eliminate a single color from step states.

Suppose s is a step state in automaton $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$. We define *a related successor of s at position p* as follows:

1. $\delta(s, f)$ is a related successor of s at position $\#_f$ if $\#_f > 0$.

2. If s_1 is a related successor of s at position p with $p \succ \varepsilon$ and s_1 is a skip state and $next(p) \succ \varepsilon$, then $\sigma(s_1)$ is a related successor of s at position $next(p)$.

3. If s_1 is a related successor of s at position p with $p \succ \varepsilon$ and s_1 is a step state and $s_2 = \delta(s_1, b)$ is a successor of s_1 with $follow(p, b) \succ \varepsilon$, then s_2 is a related successor of s at position $follow(p, b)$.

If s' is a related successor of s at position k and k is an integer, then we say *s' is a direct successor of s at position k*.

Suppose c occurs at some step state s of automaton $\mathcal{M}$. We define $\equiv_c$ to be the least equivalence relation on states that relates s and s' whenever s and s' are both colored by c and lie on some path to a terminal state whose acceptance condition contains c. We then define $\Phi^c_{\mathcal{M}}$ to be the least set closed under the following conditions:

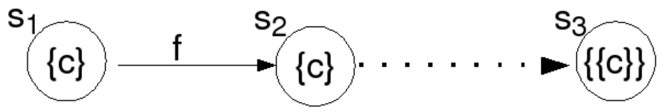

Figure 7.2: An example of occur check failure

1. If s is a step state with $c \in color(s)$ and there does not exists a state s' such that $s \equiv_c s'$ and s is a related successor of s', then s is in $\Phi^c_{\mathcal{M}}$.

2. If s is a skip state and $s \equiv_c s_1$ with some $s_1 \in \Phi^c_{\mathcal{M}}$ and there does not exists a state $s_2 \in \Phi^c_{\mathcal{M}}$ such that s is a related successor of s_2, then s' is also in $\Phi^c_{\mathcal{M}}$.

In order to safely remove color c from all the step states of $\mathcal{M}$, we have to eliminate color c from all the states in $\Phi^c_{\mathcal{M}}$ to make the resulting automaton be equivalent to $\mathcal{M}$. Note that by the above definition, there are no states s and s' in $\Phi^c_{\mathcal{M}}$ such that s' is a related successor of step state s. If a step state s and a related successor s' of s are colored by the same color c, then it is called "occur check" failure. For example, there is an "occur check" failure in Figure 7.2 with $\#_f = 1$. In that case, there is no ground n-tuple that can satisfy color c.

Suppose s is a skip state in automaton $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$. We define the *branching operation on s* to be the following process:

1. Transform s into a step state.

2. For every symbol $b \in \Sigma$ with $\#_b = 0$, we define $\delta(s, b) = \sigma(s)$.

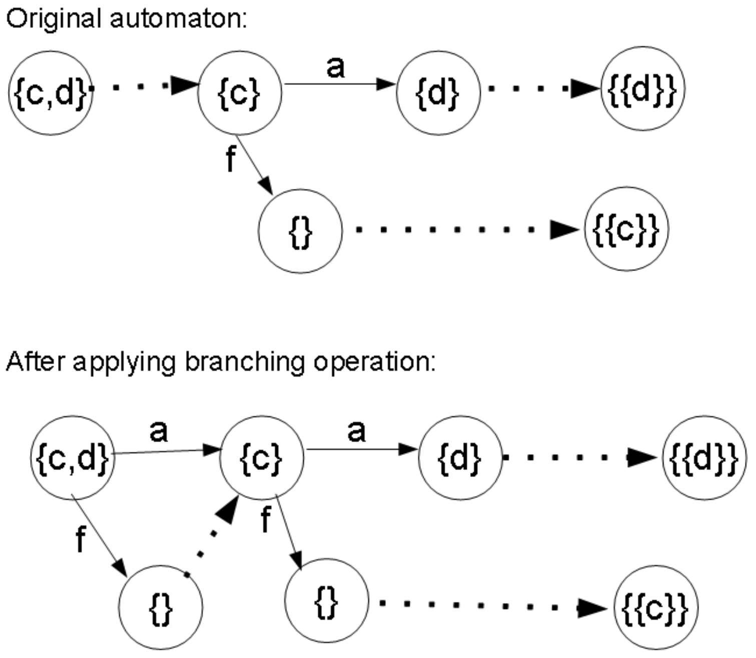

Figure 7.3: Branching on skip states

3. For every symbol $b \in \Sigma$ with $\#_b > 0$, we create a sequence of skip states $s^b_{\#_b}, \ldots, s^b_1$ and define $\delta(s,b) = s^b_{\#_b}$, $\sigma(s^b_{\#_b}) = s^b_{\#_b - 1}, \ldots$, $\sigma(s^b_1) = \sigma(s)$. That sequence of skip states are all direct successors of step state s.

An example of branching operation is shown in Figure 7.3 with $\#_f = 1$ and $\#_a = 0$.

Informally, to eliminate color c from all the states in $\Phi^c_{\mathcal{M}}$, we first transform all the skip states in $\Phi^c_{\mathcal{M}}$ to be step states by using branching operations. Then we add corresponding colors derived from c on direct successors of all the (step) states in $\Phi^c_{\mathcal{M}}$. After that, the equality constraints defined by color c are reduced to the equality constraints defined by those derived colors. Finally, color c can be safely eliminated from all the (step) states in $\Phi^c_{\mathcal{M}}$.

Informally, we construct $\mathcal{M}_1$ from $\mathcal{M}$ to eliminate color c from all the states in $\Phi^c_{\mathcal{M}}$. The branching operations are automatically performed in the construction $\mathcal{M}_1$. Suppose $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$. A state in the new construction $\mathcal{M}_1$ is a tuple (s, b, p) such that:

- s is a state in $\mathcal{M}$.
- b is a symbol in $\Sigma \cup \{\perp, \top\}$. We use symbol b to memorize the symbols that are read by the states in $\Phi^c_{\mathcal{M}}$. If $b \in \Sigma$, then all the predecessors of s that are in $\Phi^c_{\mathcal{M}}$ read the same symbol b. If $b = \perp$, then none of the predecessors of s is in $\Phi^c_{\mathcal{M}}$. If $b = \top$, then one of the following errors occurs:

  1. "Contradictory path", i.e., both $\delta(s_1, b_1)$ and $\delta(s_2, b_2)$ are predecessors of s with $s_1 \in \Phi^c_{\mathcal{M}}$ and $s_2 \in \Phi^c_{\mathcal{M}}$ and $b_1 \neq b_2$. In this case, color c is unsatisfiable.
  2. "Occur check" failure, i.e., there are predecessors s_1 and s_2 of s such that s_1 is a step state and s_2 is a related successor of s_1 and both s_1 and s_2 are colored by c.

- k is a "relative" position. We use k to keep track of the relative position from a state in $\Phi^c_{\mathcal{M}}$. If k is a positive integer, then s is a direct successor of some state s' ($s' \in \Phi^c_{\mathcal{M}}$) at position k. The pair (b, k) will be used to generate a color $c_{b,k}$ that is derived from color c. Those derived colors are used to specify the equality constraints among the corresponding direct successors

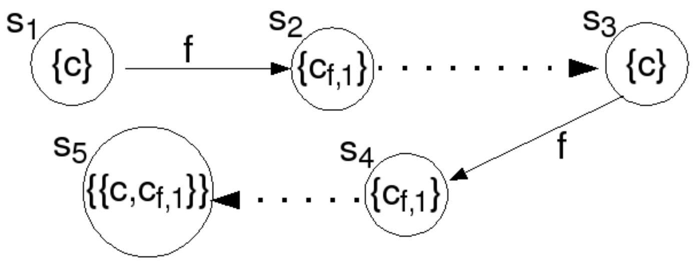

Figure 7.4: Examples of derived colors

of the states in $\Phi^c_{\mathcal{M}}$. For example, we have $\#_f = 1$ in Figure 7.4. States s_1 and s_3 are in $\Phi^c_{\mathcal{M}}$, states s_2 and s_4 are states that have the same relative position 1 from the states in $\Phi^c_{\mathcal{M}}$. Then derived color $c_{f,1}$ is used to color states s_2 and s_4 since the terms skipped at them must be identical in order to satisfy the color c. After removing color c from states s_1 and s_3, the resulting automaton is equivalent to the original one due to the derived color $c_{f,1}$.

Formally, we construct an automaton $\mathcal{M}_1 = (Q_1, s^0_1, \delta_1, \sigma_1, color_1, \tau_1)$ from automaton $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ as follows:

1. $s^0_1 = (s^0, \bot, \varepsilon)$.

2. Q is the least subset of $Q \times (\Sigma \cup \{\bot, \top\}) \times \text{Pos}$ that contains s^0_1 and is closed under skip and step functions, as described in the sequel.

3. (s, b, ε) is a step state if s is a step state and $s \notin \Phi^c_{\mathcal{M}}$. In that case, we have $color_1(s, b, \varepsilon) = color(s)$ and for all symbol $b' \in \Sigma$, if $\delta(s, b')$ is defined then $\delta_1((s, b, \varepsilon), b') = (\delta(s, b'), b, \varepsilon)$, otherwise $\delta_1((s, b, \varepsilon), b')$ is not defined.

4. (s,b,ε) is a skip state if s is a skip state and $s \notin \Phi^c_{\mathcal{M}}$. In that case, we have $color_1(s,b,\varepsilon) = color(s)$ and $\sigma_1(s,b,\varepsilon) = (\sigma(s),b,\varepsilon)$.

5. (s,b,ε) is a step state if s is a step state and $s \in \Phi^c_{\mathcal{M}}$. In that case, we have $color_1(s,b,\varepsilon) = color(s) \setminus \{c\}$. Moreover, for all symbol $b' \in \Sigma$ such that $\delta(s,b')$ is not defined, $\delta_1((s,b,\varepsilon),b')$ is not defined. For all symbol $b' \in \Sigma$ such that $\delta(s,b')$ is defined,

$$\delta_1((s,b,\varepsilon),b') = (\delta(s,b'),b'',follow(\varepsilon,\#_{b'}))$$

in which:

- $b'' = \top$ if $b = \top$.
- $b'' = b'$ if $b = \bot$.
- b'' is either $\top$ if $b \in \Sigma$ and $b \neq b'$, or b if $b = b'$. In the former case, a contradictory symbol has been read.

6. (s,b,ε) is a step state if s is a skip state and $s \in \Phi^c_{\mathcal{M}}$. In that case, we have $color_1(s,b,\varepsilon) = color(s) \setminus \{c\}$. Moreover, for all symbol $b' \in \Sigma$,

$$\delta_1((s,b,\varepsilon),b') = (s',b'',follow(\varepsilon,\#_{b'})),$$

where s' is either s if $follow(\varepsilon,\#_{b'}) \succ \varepsilon$ or $\sigma(s)$ if $follow(\varepsilon,\#_{b'}) = \varepsilon$ and:

- $b'' = \top$ if $b = \top$.
- $b'' = b'$ if $b = \bot$.
- b'' is either $\top$ if $b \in \Sigma$ and $b \neq b'$, or b if $b = b'$. In the former case, a

contradictory symbol has been read.

7. (s,b,k) is a skip state if s is a skip state in $\Phi^c_{\mathcal{M}}$ and $k \succ \varepsilon$. In that case, if $b \in \Sigma$ and k is an integer, then $color_1(s,b,k) = \{c_{b,k}\}$, otherwise $color_1(s,b,k) = \emptyset$. In the former case, a color $c_{b,k}$ derived from c is added. In addition, we have $\sigma_1(s,b,k) = (s', b, next(k))$ where s' is either s if $next(k) \succ \varepsilon$, or $\sigma(s)$ if $next(k) = \varepsilon$.

8. (s,b,k) is a skip state if s is a skip state and $s \notin \Phi^c_{\mathcal{M}}$ and $k \succ \varepsilon$. In that case, if $b \in \Sigma$ and k is an integer, then $color_1(s,b,k) = color(s) \cup \{c_{b,k}\}$, otherwise $color_1(s,b,k) = color(s)$. In the former case, a color $c_{b,k}$ derived from c is added. In addition, we have $\sigma_1(s,b,k) = (\sigma(s), b', next(k))$, where b' is either b if $c \notin color(s)$ or $\top$ if $c \in color(s)$. If $c \in color(s)$, then we find an "occur check" failure.

9. (s,b,k) is a step state if s is a step state and $k \succ \varepsilon$. In that case, if $b \in \Sigma$ and k is an integer, then $color_1(s,b,k) = color(s) \cup \{c_{b,k}\}$, otherwise $color_1(s) = color(s)$. In the former case, a color $c_{b,k}$ derived from c is added. In addition, we have:

   - $\delta_1((s,b,k),b')$ is not defined if $\delta(s,b')$ is not defined.
   - $\delta_1((s,b,k),b') = (\delta(s,b'), b, follow(k,b'))$ if $\delta(s,b')$ is defined and $c \notin color(s)$.
   - $\delta_1((s,b,k),b') = (\delta(s,b'), \top, follow(k,b'))$ if $\delta(s,b')$ is defined and $c \in color(s)$. In this case, we find an "occur check" failure.

10. $(s,\bot,\varepsilon)$ is a terminal state if s is a terminal state. In that case, $\tau_1(s,\bot,\varepsilon) = \tau(s)$.

11. $(s,\top,\varepsilon)$ is a terminal state if s is a terminal state. In that case, $\tau_1(s,\top,\varepsilon)$ is obtained from $\tau(s)$ by eliminating any set that contains c.

12. (s,b,ε) is a terminal state if s is a terminal state and $b \in \Sigma$. In that case, $\tau_1(s,b,\varepsilon)$ is obtained from $\tau(s)$ by replacing any set C that contains c with $(C \setminus \{c\}) \cup \{c_{b,\#_b},\ldots,c_{b,1}\}$.

We call the above construction*a c-normalized automaton* and call the procedure that builds such construction *a c-normalization procedure*. The following facts are related to c-normalized automata.

Lemma 7.2.1 *Suppose*

$$\mathcal{M}_1 = (Q_1, s_1^0, \delta_1, \sigma_1, color_1, \tau_1)$$

is a c-normalized automaton obtained from $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ and (s,b,k) is a state in $\mathcal{M}_1$ with some state s in $\mathcal{M}$ and some symbol b in $\Sigma \cup \{\bot,\top\}$ and some position k. Then the following conditions hold for all positions p of a ground n-tuple t:

- *$C_{\mathcal{M}_1,t}(p)$ is defined if and only if $C_{\mathcal{M},t}(p)$ is defined. Moreover, if we have $C_{\mathcal{M}_1,t}(p) = (s,b,k)$ then $C_{\mathcal{M},t}(p) = s$.*
- *If $p = p^{(s,b,k)}_{\mathcal{M}_1,t}$, then p is either $p^s_{\mathcal{M},t}\,k$ whenever s is skip state in $\Phi^c_{\mathcal{M}}$ and $k \succ \varepsilon$, or $p^s_{\mathcal{M},t}$ in all other cases.*

Proof. We verify this by induction on positions p of ground n-tuple t that the stated conditions hold for p. If $p = n$, then $p^s_{\mathcal{M},t} = n$ and $C_{\mathcal{M}_1,t}(n) = (s^0, \perp, \varepsilon) = (C_{\mathcal{M},t}(n), \perp, \varepsilon)$. The stated conditions hold in this case with $s = s^0$ and $b = \perp$ and $k = \varepsilon$.

Suppose p is a position of t such that $n \succeq p \succ \varepsilon$ and $t \ @ \ p = b'$ with some symbol $b' \in \Sigma$ and the stated conditions hold for p. We show that for position $follow(p,b')$ of t, the stated conditions also hold. Let $C_{\mathcal{M},t}(p) = s$. We consider the following cases:

1. $C_{\mathcal{M}_1,t}(p) = (s,b,k)$ is a skip state. Then s is a skip state. In this case, Both $C_{\mathcal{M}_1,t}(follow(p,b'))$ and $C_{\mathcal{M},t}(follow(p,b'))$ are defined. There are two cases:

   (a) $k = \varepsilon$. Then s is not in $\Phi^c_{\mathcal{M}}$. By condition (2) of the induction hypothesis, $p^{(s,b,k)}_{\mathcal{M}_1,t} = p^s_{\mathcal{M},t}$. This leaves two possibilities:

      i. $follow(p,b') \succ next(p^s_{\mathcal{M},t})$. Then $C_{\mathcal{M},t}(follow(p,b')) = s$ and $C_{\mathcal{M}_1,t}(follow(p,b')) = (s,b,\varepsilon)$. The stated conditions still hold in this case.

      ii. $follow(p,b') = next(p^s_{\mathcal{M},t})$. Then $C_{\mathcal{M},t}(follow(p,b')) = \sigma(s)$. Moreover, $C_{\mathcal{M}_1,t}(follow(p,b')) = \sigma_1(s,b,\varepsilon) = (\sigma(s),b,\varepsilon)$ by the definition of $\mathcal{M}_1$. In this case, $follow(p,b') = p^{(\sigma(s),b,\varepsilon)}_{\mathcal{M}_1,t} = p^{\sigma(s)}_{\mathcal{M},t}$. The stated conditions hold in this case.

   (b) $k \succ \varepsilon$. This leaves two possibilities:

      i. $s \in \Phi^c_{\mathcal{M}}$. By condition (2) of the induction hypothesis, $p^{(s,b,k)}_{\mathcal{M}_1,t} =$

$p^s_{\mathcal{M},t}\ k$. By Lemma 5.4.4, $next(p^s_{\mathcal{M},t}\ k) \succeq next(p^s_{\mathcal{M},t})$. We consider two cases:

A. $follow(p,b') \succ next(p^s_{\mathcal{M},t}\ k)$. Then $C_{\mathcal{M},t}(follow(p,b')) = s$ and $C_{\mathcal{M}_1,t}(follow(p,b')) = (s,b,k)$. The stated conditions still hold in this case.

B. $follow(p,b') = next(p^s_{\mathcal{M},t}\ k)$. Suppose $next(k) > \varepsilon$. By condition (2) of 5.4.3,

$$follow(p,b') = next(p^s_{\mathcal{M},t}\ k) = p^s_{\mathcal{M},t}\ next(k) \succ next(p^s_{\mathcal{M},t}).$$

By the definition of $\mathcal{M}_1$,

$$C_{\mathcal{M}_1,t}(follow(p,b')) = (s,b,next(k)).$$

Moreover, we have $C_{\mathcal{M},t}(follow(p,b')) = s$. Thus, condition (1) holds in this case. Now $p^{(s,b,next(k))}_{\mathcal{M}_1,t} = p^s_{\mathcal{M},t}\ next(k)$. Thus, condition (2) also holds in this case. Now suppose $next(k) = \varepsilon$. By condition (3) of Lemma 5.4.3,

$$follow(p,b') = next(p^s_{\mathcal{M},t}\ k) = next(p^s_{\mathcal{M},t}).$$

By the definition of $\mathcal{M}_1$,

$$C_{\mathcal{M}_1,t}(next(p^s_{\mathcal{M},t})) = (\sigma(s),b,\varepsilon).$$

Moreover, we have $C_{\mathcal{M},t}(next(p^s_{\mathcal{M},t})) = \sigma(s)$. Thus, condition (1) holds in this case. Now $p^{(\sigma(s),b,\varepsilon)}_{\mathcal{M}_1,t} = next(p^s_{\mathcal{M},t}) = p^{\sigma(s)}_{\mathcal{M},t}$. Thus, condition (2) also holds in this case.

ii. $s \notin \Phi^c_{\mathcal{M}}$. By condition (2) of the induction hypothesis, $p^{(s,b,k)}_{\mathcal{M}_1,t} = p^s_{\mathcal{M},t}$. We consider two cases:

A. $follow(p,b') \succ next(p^s_{\mathcal{M},t})$. Then $C_{\mathcal{M},t}(follow(p,b')) = s$ and $C_{\mathcal{M}_1,t}(follow(p,b')) = (s,b,k)$. The stated conditions still hold in this case.

B. $follow(p,b') = next(p^s_{\mathcal{M},t})$. By the definition of $\mathcal{M}_1$,

$$C_{\mathcal{M}_1,t}(follow(p,b')) = (\sigma(s),b',next(k)),$$

where b' is either b if $c \notin color(s)$ or $\top$ if $c \in color(s)$. Moreover, we have $C_{\mathcal{M},t}(follow(p,b')) = \sigma(s)$. Thus, condition (1) holds in this case. Now $p^{(\sigma(s),b,next(k))}_{\mathcal{M}_1,t} = next(p^s_{\mathcal{M},t}) = p^{\sigma(s)}_{\mathcal{M},t}$. Thus, condition (2) also holds in this case.

2. $C_{\mathcal{M}_1,t}(p) = (s,b,k)$ is a step state. By Lemma 5.5.1, $p = p^{(s,b,k)}_{\mathcal{M}_1,t}$. Moreover, by the definition of $\mathcal{M}_1$, s is either a step state in $\mathcal{M}$, or a skip state in $\mathcal{M}$ with $s \in \Phi^c_{\mathcal{M}}$ and $k = \varepsilon$. In the former case, if $C_{\mathcal{M},t}(follow(p,b'))$ is defined, then $\delta(s,b')$ is defined since we have $C_{\mathcal{M},t}(follow(p,b')) = \delta(s,b')$. It follows by the definition of $\mathcal{M}_1$ that $C_{\mathcal{M}_1,t}(follow(p,b'))$ is also defined. If $C_{\mathcal{M}_1,t}(follow(p,b'))$ is defined, then $\delta(s,b')$ is defined. It follows that $C_{\mathcal{M},t}(follow(p,b'))$ is also defined since $C_{\mathcal{M},t}(follow(p,b')) = \delta(s,b')$. In

the latter case, we have that $C_{\mathcal{M}_1,t}(follow(p,b'))$ and $C_{\mathcal{M},t}(follow(p,b'))$ are defined for all symbol $b' \in \Sigma$ since s is a skip state in $\mathcal{M}$. Now suppose $C_{\mathcal{M},t}(follow(p,b'))$ is defined. There are two cases:

(a) $k = \varepsilon$. By condition (2) of the induction hypothesis, $p^{(s,b,\varepsilon)}_{\mathcal{M}_1,t} = p^{s}_{\mathcal{M},t}$. This leaves two possibilities:

i. $s \notin \Phi^c_{\mathcal{M}}$. Then s is a step state and $C_{\mathcal{M},t}(follow(p,b')) = \delta(s,b')$. By the definition of $\mathcal{M}_1$,

$$C_{\mathcal{M}_1,t}(follow(p,b')) = \delta_1((s,b,\varepsilon),b') = (\delta(s,b'),b,\varepsilon).$$

Clearly, the stated conditions hold in this case.

ii. $s \in \Phi^c_{\mathcal{M}}$. Then we have two cases:

A. s is a step state. Then $C_{\mathcal{M},t}(follow(p,b')) = \delta(s,b')$ and by the definition of $\mathcal{M}_1$,

$$\begin{aligned} C_{\mathcal{M}_1,t}(follow(p,b')) &= \delta_1((s,b,\varepsilon),b') \\ &= (\delta(s,b'),b'',follow(\varepsilon,\#_{b'})), \end{aligned}$$

where $b'' = \top$ if either $b = \top$ or $b \in \Sigma$ and $b \neq b'$, or else $b'' = b'$ if either $b = b'$ or $b = \bot$. Clearly, the stated conditions hold in this case.

B. s is a skip state. Then we have the following cases:

- $follow(\varepsilon, \#_{b'}) \succ \varepsilon$. Then

$$follow(p, b') \succ next(p) = next(p^s_{\mathcal{M},t}).$$

Thus, $C_{\mathcal{M},t}(follow(p, b')) = s$. By the definition of $\mathcal{M}_1$,

$$\begin{aligned} C_{\mathcal{M}_1,t}(follow(p, b')) &= \delta_1((s, b, \varepsilon), b') \\ &= (s, b'', follow(\varepsilon, \#_{b'})), \end{aligned}$$

where $b'' = \top$ if either $b = \top$ or $b \in \Sigma$ and $b \neq b'$, or else $b'' = b'$ if either $b = b'$ or $b = \bot$. Clearly, condition (1) holds in this case. Now

$$p^{(s, b'', follow(\varepsilon, \#_{b'}))}_{\mathcal{M}_1,t} = p^s_{\mathcal{M},t}\ follow(\varepsilon, \#_{b'}).$$

Thus, condition (2) also holds in this case.

- $follow(\varepsilon, \#_{b'}) = \varepsilon$. Then

$$follow(p, b') = next(p) = next(p^s_{\mathcal{M},t}).$$

Thus, $C_{\mathcal{M},t}(follow(p, b')) = \sigma(s)$. By the definition of $\mathcal{M}_1$,

$$\begin{aligned} C_{\mathcal{M}_1,t}(follow(p, b')) &= \delta_1((s, b, \varepsilon), b') \\ &= (\sigma(s), b'', \varepsilon), \end{aligned}$$

where $b'' = \top$ if either $b = \top$ or $b \in \Sigma$ and $b \neq b'$, or else $b'' = b'$ if either $b = b'$ or $b = \bot$. Clearly, condition (1) holds in this case. Now $p^{(\sigma(s),b'',\varepsilon)}_{\mathcal{M}_1,t} = follow(p,b') = next(p^{s}_{\mathcal{M},t}) = p^{\sigma(s)}_{\mathcal{M},t}$. Thus, condition (2) also holds in this case.

(b) $k \succ \varepsilon$. Then s is a step state and $C_{\mathcal{M},t}(follow(p,b')) = \delta(s,b')$. By condition (2) of the induction hypothesis, $p^{(s,b,k)}_{\mathcal{M}_1,t} = p^{s}_{\mathcal{M},t}$. By the definition of $\mathcal{M}_1$,

$$C_{\mathcal{M}_1,t}(follow(p,b')) = (\delta(s,b'),b'',follow(k,b')),$$

where b'' is either b if $c \notin color(s)$, or $\top$ if $c \in color(s)$. Clearly, condition (1) holds in this case. Now

$$p^{(\delta(s,b'),b'',follow(k,b')))}_{\mathcal{M}_1,t} = follow(p,b') = p^{\delta(s,b')}_{\mathcal{M},t}.$$

Thus, condition (2) also holds in this case.

□

Theorem 7.2.2 *Suppose* $\mathcal{M}_1 = (Q_1, s^0_1, \delta_1, \sigma_1, color_1, \tau_1)$ *is a c-normalized automaton obtained from* $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$. *Then* $\mathcal{M}_1$ *is equivalent to* $\mathcal{M}$.

Proof. Suppose t is a ground n-tuple that is accepted by $\mathcal{M}$. Then $C_{\mathcal{M},t}(\varepsilon)$ is a terminal state s in $\mathcal{M}$ and $t \models_{\mathcal{M}} \tau(s)$. By Lemma 7.2.1, $C_{\mathcal{M}_1,t}(\varepsilon) = (s,b,\varepsilon)$ with some symbol b in $\Sigma \cup \{\bot, \top\}$. By the definition of $\mathcal{M}_1$, $C_{\mathcal{M}_1,t}$ is terminating. We consider the following cases:

1. $b = \bot$. Then by the definition of $\mathcal{M}_1$, $\tau_1(s, \bot, \varepsilon) = \tau(s)$. Since $t \models_{\mathcal{M}} \tau(s)$, $t \models_{\mathcal{M}} \tau_1(s, \bot, \varepsilon)$.

2. $b = \top$. Then color c is unsatisfiable. Thus, $t \not\models_{\mathcal{M}} c$. Since $t \models_{\mathcal{M}} \tau(s)$, there is at least one set C in $\tau(s)$ such that C does not contain color c and $t \models_{\mathcal{M}} C$. Moreover, by the definition of $\mathcal{M}_1$, $\tau_1(s, \top, \varepsilon)$ is obtained from $\tau(s)$ by eliminating any set that contains c. Thus, $\tau_1(s, \top, \varepsilon)$ contains the set C. Since $t \models_{\mathcal{M}} C$, $t \models_{\mathcal{M}_1} \tau_1(s, \top, \varepsilon)$.

3. $b \in \Sigma$. Then by the definition of $\mathcal{M}_1$, $\tau_1(s, b, \varepsilon)$ is obtained from $\tau(s)$ by replacing any set C that contains c with set C', where $C' = (C \setminus \{c\}) \cup \{c_{b,\#_b}, \ldots, c_{b,1}\}$. There are two cases:

   (a) $t \not\models_{\mathcal{M}} c$. By following the proof in case (2), we have $t \models_{\mathcal{M}} \tau_1(s, b, \varepsilon)$.

   (b) $t \models_{\mathcal{M}} c$. Since $b \in \Sigma$, by the definition of $\mathcal{M}_1$, for all the predecessors (s', b', p') of (s, b, ε) such that $c \in color(s')$ ($s' \in \Phi^c_{\mathcal{M}}$), (s', b', p') reads the same symbol b. Moreover, direct successors of s' are colored by $c_{b,\#_b}, \ldots, c_{b,1}$ (that are derived from c) respectively. Since $t \models_{\mathcal{M}} c$, for all $i \in [1, \#_b]$, we have $t \models_{\mathcal{M}_1} c_{b,i}$. That is, if C is a set of $\tau(s)$ that contains c, then $t \models_{\mathcal{M}_1} C'$ whenever $t \models_{\mathcal{M}} C$. It follows that $t \models_{\mathcal{M}_1} \tau_1(s, b, \varepsilon)$.

Since $C_{\mathcal{M}_1,t}$ is terminating and $t \models_{\mathcal{M}_1} \tau_1(s, b, \varepsilon)$, t is also accepted by $\mathcal{M}_1$.

Conversely, suppose t is a ground n-tuple that is accepted by $\mathcal{M}_1$. Then $C_{\mathcal{M}_1,t}(\varepsilon)$ is a terminal state (s,b,ε) in $\mathcal{M}_1$ and $t \models_{\mathcal{M}_1} \tau_1(s,b,\varepsilon)$ with some symbol b in $\Sigma \cup \{\bot,\top\}$. By Lemma 7.2.1, $s = C_{\mathcal{M},t}(\varepsilon)$ is a terminal state in $\mathcal{M}$. Thus, $C_{\mathcal{M},t}$ is terminating. We consider the following cases:

1. $b = \bot$. Then by the definition of $\mathcal{M}_1$, we have $\tau(s) = \tau_1(s,\bot,\varepsilon)$. Since $t \models_{\mathcal{M}_1} \tau_1(s,\bot,\varepsilon)$, $t \models_{\mathcal{M}} \tau(s)$.

2. $b = \top$. Then color c is unsatisfiable. Thus, $t \not\models_{\mathcal{M}} c$. Since $t \models_{\mathcal{M}_1} \tau_1(s,\top,\varepsilon)$, there is at least one set C in $\tau_1(s,\top,\varepsilon)$ such that C does not contain color c and $t \models_{\mathcal{M}} C$. Moreover, by the definition of $\mathcal{M}_1$, $\tau_1(s,\top,\varepsilon)$ is obtained from $\tau(s)$ by eliminating any set that contains c. Thus, $\tau(s)$ contains the set C. Since $t \models_{\mathcal{M}} C$, $t \models_{\mathcal{M}} \tau(s)$.

3. $b \in \Sigma$. Then by the definition of $\mathcal{M}_1$, $\tau_1(s,b,\varepsilon)$ is obtained from $\tau(s)$ by replacing any set C that contains c with set C', where $C' = (C \setminus \{c\}) \cup \{c_{b,\#_b},\ldots,c_{b,1}\}$. Since $t \models_{\mathcal{M}_1} \tau_1(s,b,\varepsilon)$, there exists at least one set C' in $\tau_1(s,b,\varepsilon)$ such that $t \models_{\mathcal{M}_1} C'$. Suppose C' is obtained from the set $C \in \tau(s)$. Then there are two cases:

   (a) $c \in C$. Then $C' = (C \setminus \{c\}) \cup \{c_{b,\#_b},\ldots,c_{b,1}\}$. Since $t \models_{\mathcal{M}_1} C'$, $t \models_{\mathcal{M}_1} c_{b,i}$ for all $i \in [1,\#_b]$. Moreover, any state in the computation of t on $\mathcal{M}$ colored by c is a state that starts reading or skipping symbol b. It follows that $t \models_{\mathcal{M}} c$. Thus, we also have $t \models_{\mathcal{M}} C$, and in turn, $t \models_{\mathcal{M}} \tau(s)$.

(b) $c \notin C$. Then $C' = C$. Since $t \models_{\mathcal{M}_1} C'$, $t \models_{\mathcal{M}} C$. It follows that $t \models_{\mathcal{M}} \tau(s)$.

Since $C_{\mathcal{M},t}$ is terminating and $t \models_{\mathcal{M}} \tau(s)$, t is also accepted by $\mathcal{M}$. □

A c-normalization procedure is basically a unification procedure that performs the following reductions:

1. Reduces an equation $f(t_1,\ldots,t_n) = f(t'_1,\ldots,t'_n)$ to a set of equations $t_1 = t'_1,\ldots,t_n = t'_n$. This happens when color c occurs at two step states and we eliminate c from both states.

2. Reduces an equation $x = f(t_1,\ldots,t_n)$ to a set of equations $x_1 = t_1,\ldots,x_n = t_n$ in which x is reduced to $f(x_1,\ldots,x_n)$. This happens when color c occurs at a skip state and a step state and we eliminate c from both states.

This procedure is called *term reduction* in [35].

7.3 Stratified Automata

We have defined c-normalized automata in the previous section. Clearly, an automaton $\mathcal{M}$ is normalized if and only if it is c-normalized for all colors c. We can construct a normalized automaton by repeatedly applying c-normalization procedure on the original automaton for all colors c. To show that the iterative application finally terminates, we need the following definition.

We say $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is *stratified* if there exists a mapping $pos_{\mathcal{M}} : Q \rightarrow \textsc{Pos}$ such that for all ground n-tuples t, if $p = p^s_{\mathcal{M},t}$ is a position of t with some state s in $\mathcal{M}$, then p must be $pos_{\mathcal{M}}(s)$. We call the mapping $pos_{\mathcal{M}}$ a *position assignment* of $\mathcal{M}$.

We claim that the following facts about stratified automata.

Theorem 7.3.1 *For any automaton $\mathcal{M}_1$, there exists an equivalent stratified automaton $\mathcal{M}$.*

Proof. Suppose $\mathcal{M}_1 = (Q_1, s_1^0, \delta_1, \sigma_1, color_1, \tau_1)$ accepts ground n-tuples. Let $\mathcal{M}$ be a conjunction of $\mathcal{M}_1$ and $\mathcal{M}_2$ in which $\mathcal{M}_2$ is defined to be a tuple

$$(Q_2, s_2^0, \delta_2, \sigma_2, color_2, \tau_2)$$

as follows:

1. $Q_2 = \{u_n, \ldots, u_1, u_0\}$.
2. $s_2^0 = u_n$.
3. $u_n, \ldots, u_1$ are all skip states such that $\forall i \in [1, n]$, $\sigma_2(u_i) = u_{i-1}$.
4. There are no step states in Q_2.
5. u_0 is the only terminal state in Q_2 with $\tau_2(u_0) = \mathbf{T}$.

Clearly, $\mathcal{M}_2$ accepts all ground n-tuples. By Theorem 6.1.4, $\mathcal{M}$ is equivalent to $\mathcal{M}_1$.

Now we show that $\mathcal{M}$ is stratified. Suppose t is a ground n-tuple and p is a position of t with $p^s_{\mathcal{M},t}$, where $s = ((\varepsilon, s_1), (p_2, u_i))$. By Lemma 6.1.2, $C_{\mathcal{M}_1,t}(p \setminus$

$\varepsilon) = s_1$ and $C_{\mathcal{M}_2,t}(p \setminus p_2) = u_i$. Clearly, p is either $i\ p_2$ if $i > 0$ or ε if $i = 0$ (note that, by the definition of $\mathcal{M}$, if $i = 0$ then $p_2 = \varepsilon$). That is, there exists a mapping $pos_{\mathcal{M}} : Q \rightarrow \textsc{Pos}$ such that for all ground n-tuples t, if $p = p^s_{\mathcal{M},t}$, then p must be $pos_{\mathcal{M}}(s)$, where $s = ((\varepsilon, s_1), (p_2, u_i))$ and $pos_{\mathcal{M}}(s)$ is either $i\ p_2$ if $i > 0$ or ε if $i = 0$. It follows that $\mathcal{M}$ is stratified. □

Lemma 7.3.2 *Suppose $\mathcal{M}'$ is a c-normalized automaton obtained from stratified automaton $\mathcal{M}$. Then $\mathcal{M}'$ is also stratified.*

Proof. Suppose $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ and $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$. Since $\mathcal{M}$ is stratified, there exists a mapping $pos_{\mathcal{M}} : Q \rightarrow \textsc{Pos}$ such that for all ground n-tuples t, if $p = p^s_{\mathcal{M},t}$ is a position of t with some state s in $\mathcal{M}$, then p must be $pos_{\mathcal{M}}(s)$. Suppose (s, b, k) is a state of $\mathcal{M}'$ with some state s in $\mathcal{M}$ and some symbol $b \in \Sigma \cup \{\top, \bot\}$ and some relative position k. Suppose further, t is a ground n-tuple and $pos_{\mathcal{M}}(s) = p^s_{\mathcal{M},t}$.

By condition (2) of Lemma 7.2.1, we obtain a mapping $pos'_{\mathcal{M}} : Q' \rightarrow \textsc{Pos}$ as follows:

- If s is a skip state in $\Phi^c_{\mathcal{M}}$ and $k \succ \varepsilon$, then $pos_{\mathcal{M}'}(s) = pos_{\mathcal{M}}(s)\ k = p^{(s,b,k)}_{\mathcal{M}',t}$.
- In all other cases, $pos_{\mathcal{M}'}(s) = pos_{\mathcal{M}}(s) = p^{(s,b,k)}_{\mathcal{M}',t}$.

It follows that $\mathcal{M}'$ is also stratified. □

7.4 Eliminating Colors from Step States

Now we claim that a normalized automaton equivalent to any given stratified automaton always exists.

Theorem 7.4.1 *Suppose $\mathcal{M}$ is an arbitrary stratified automaton. Then there is a normalized automaton $\mathcal{M}'$ that is equivalent to $\mathcal{M}$.*

Proof. If for all the step states s in $\mathcal{M}$, $color(s) = \emptyset$, then there is nothing to do.

Suppose there exists a step state s in $\mathcal{M}$ such that $color(s)$ contains at least one color c. Since $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is stratified, there exists a mapping $pos_{\mathcal{M}} : Q \to \textsc{Pos}$. We define $\parallel \mathcal{M} \parallel$ to be the pair (p,k), where p is a position and k is a natural number, defined as follows:

- p is the greatest position for which there exists a color c and a state s such that $pos_{\mathcal{M}}(s) = p$ and $s \in \Phi^c_{\mathcal{M}}$.
- k is the number of colors c for which there exists a state s such that we have $pos_{\mathcal{M}}(s) = p$ and $s \in \Phi^c_{\mathcal{M}}$.

Suppose $\parallel \mathcal{M} \parallel = (p,k)$. Then when we do c-normalization on $\mathcal{M}$, we always choose a color c such that there exists a state s with $pos_{\mathcal{M}}(s) = p$ and $s \in \Phi^c_{\mathcal{M}}$. This is called “picking a color with the greatest position” strategy.

Suppose $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ is obtained from $\mathcal{M}$ by repeatedly applying c-normalization with the “picking a color with the greatest position” strategy. Suppose further, p is the greatest position for which there exists a color c' and a state s such that $pos'_{\mathcal{M}}(s) = p$ and $s \in \Phi^{c'}_{\mathcal{M}'}$. Then there are four cases for state s:

1. s is a step state in the original automaton $\mathcal{M}$.

2. s is a skip state in the original automaton $\mathcal{M}$.

3. s is a step state in $\mathcal{M}'$ and is a skip state in the original automaton $\mathcal{M}$. That is, s is transformed into a step state by a c-normalization procedure.

4. s is a skip state in $\mathcal{M}'$ and not a state in the original automaton $\mathcal{M}$. Then s is new skip state introduced by the last c-normalization procedure and $color'(s) = \{c'\}$ in which c' is a derived color.

In the first three cases, p is a position of some state in $\mathcal{M}$. In the last case, p has the form $\alpha\,\beta$ with some positions $\alpha \succ \varepsilon$ and $\beta \succ \varepsilon$ such that the following conditions hold:

1. There exists a skip state s_1 in $\mathcal{M}$ with $pos_{\mathcal{M}}(s_1) = \alpha$ that is transformed into a step state in $\mathcal{M}'$ with a c-normalization on some automaton $\mathcal{M}''$. That is, s_1 is in $\Phi^c_{\mathcal{M}''}$. Moreover, s is a related successor of s_1 in $\mathcal{M}'$.

2. There exists a step state s_2 in $\mathcal{M}$ with $pos_{\mathcal{M}}(s_2) = q$ for some position q and s_2 is also in $\Phi^c_{\mathcal{M}''}$.

3. There is a step state s_3 in the original automaton $\mathcal{M}$ such that s_3 is a related successor of s_2 with $pos_{\mathcal{M}}(s_3) = q\,\beta$ and s_3 is in $\Phi^{c'}_{\mathcal{M}'}$.

That is, in the last case, position α is a position of some state in $\mathcal{M}$ and β is also a relative position of some state in $\mathcal{M}$. In summary, since $\mathcal{M}$ has finite number of states, we can only choose finite number of positions p to form pairs (p,k) with respect to the original automaton $\mathcal{M}$.

Now we define a total ordering on the pairs as the following:

- If $p_1 \succ p_2$, or $p_1 = p_2$ and $k_1 > k_2$, then we say $(p_1,k_1) \succ (p_2,k_2)$.

Since we can only choose finite number of positions p from set $\textsc{Pos}$ to form pairs (p,k) with respect to the original automaton $\mathcal{M}$, no infinite decreasing sequence can be obtained from the set $\textsc{Pos} \times N$ with the above ordering.

Suppose $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ is a c-normalized automaton obtained from $\mathcal{M}$ by using the "picking a color with the greatest position" strategy. By Lemma 7.3.2, $\mathcal{M}'$ is also stratified with a mapping $pos_{\mathcal{M}'} : Q' \to \textsc{Pos}$. Suppose $\parallel \mathcal{M} \parallel = (p_1,k_1)$ and $\parallel \mathcal{M}' \parallel = (p_2,k_2)$. By the definition of $\mathcal{M}'$, we have the following facts:

1. For all colors c' that is derived from c, c' only occurs at states s of $\mathcal{M}'$ with $p_1 \succ pos_{\mathcal{M}'}(s)$. That is, new colors appearing $\mathcal{M}'$ but not in $\mathcal{M}$ only occur at states s with $p_1 \succ pos_{\mathcal{M}'}(s)$.

2. For all colors c' appearing in $\mathcal{M}$ with $c' \neq c$, if c' occurs at a state s in $\mathcal{M}$ with $pos_{\mathcal{M}}(s) = p$, then $p_1 \succeq p$ and c' occurs at a state s' in $\mathcal{M}'$ with $pos_{\mathcal{M}'}(s) = p$.

It follows that $p_1 \succeq p_2$. If $p_1 \succ p_2$, then we have $(p_1,k_1) \succ (p_2,k_2)$. If $p_1 = p_2$, then $k_1 > k_2$. That is, there is one fewer color c for which there exists a state s such that $pos_{\mathcal{M}'}(s) = p_1 = p_2$ and $s \in \Phi^c_{\mathcal{M}'}$. This is because such a color c has been eliminated.

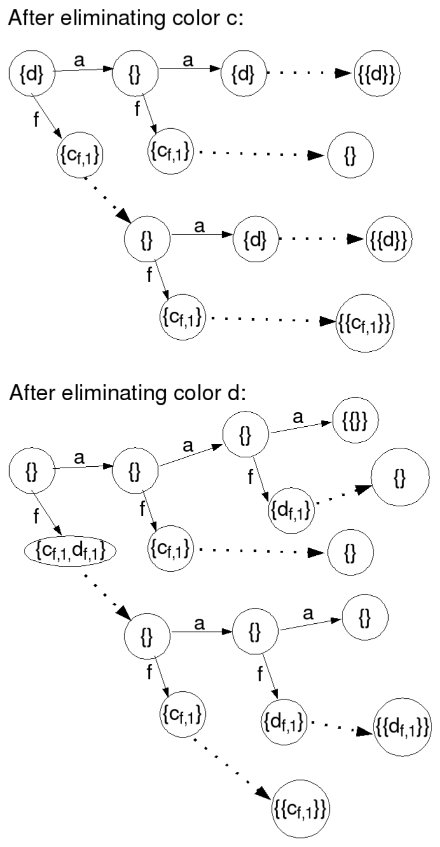

Figure 7.5: Normalization of automata

In summary, we always have $\parallel \mathcal{M} \parallel \succ \parallel \mathcal{M}' \parallel$. It follows that repeatedly applying c-normalization procedure as described above will finally terminate. By Theorem 7.2.2, the resulting automaton is equivalent to $\mathcal{M}$. $\square$

We show an example of normalization of automata in Figure 7.5. The original automaton $\mathcal{M}$ is shown in Figure 7.3. After eliminating color c, we obtain an automaton shown in the top of Figure 7.5. After eliminating color d, we obtain a normalized automaton shown in the bottom of Figure 7.5. In the sequel, without special notation, an automaton is always normalized (and stratified).

Chapter 8

Grouping and Ungrouping of Automata

8.1 Grouping of Automata

We need the following definitions for the grouping operation of automata. Suppose μ is a linear m-tuple with free variables $x_n, \ldots, x_1$ and $t_n \ldots t_1$ is an n-tuple, we denote m-tuple $\mu\theta$ by $\mu\{t_n \ldots t_1\}$, where θ is a substitution $[t_n/x_n, \ldots, t_1/x_1]$.

We assume that for linear m-tuple μ and all $i, j \in [1, n]$ with $i > j$, if μ @ $p_i = x_i$ and μ @ $p_j = x_j$ then $p_i \succ p_j$. That is, in m-tuple μ, if $n \geq i > j \geq 1$ then variable x_i always occurs on the left of x_j. Moreover, we denote position p_i by $p_{\mu,i}$ (read as the position of x_i in μ).

We say an automaton $\mathcal{M}'$ is *a grouping of automaton $\mathcal{M}$ with respect to m-tuple* μ, if the following condition holds:

- a ground m-tuple t' is accepted by $\mathcal{M}'$ if and only if ground n-tuple t is accepted by $\mathcal{M}$, where $t' = \mu\{t\}$.

Informally, to obtain a grouping of $\mathcal{M}$ with respect to μ, we shall construct an automaton $\mathcal{M}'$ such that each state in $\mathcal{M}'$ is a tuple (p, q, s) such that:

- p is a position of linear m-tuple μ,
- s is a state of automaton $\mathcal{M}$,
- and q is *the relative position* of s.

The initial state of $\mathcal{M}'$ is (m, ε, s_0) in which s_0 is the initial state of $\mathcal{M}$. Suppose after reading a prefix of a ground m-tuple $\mu\{t_n \ldots t_1\}$, we reach a state (p, q, s) of automaton $\mathcal{M}'$. If μ @ p is a function symbol or a constant, then q is always ε. In this case, (p, ε, s) is a step state and has only one transition, that is, read the symbol μ @ p (that is equal to $(\mu\{t_n \ldots t_1\})$ @ p) and go to the next step. Note that, in this

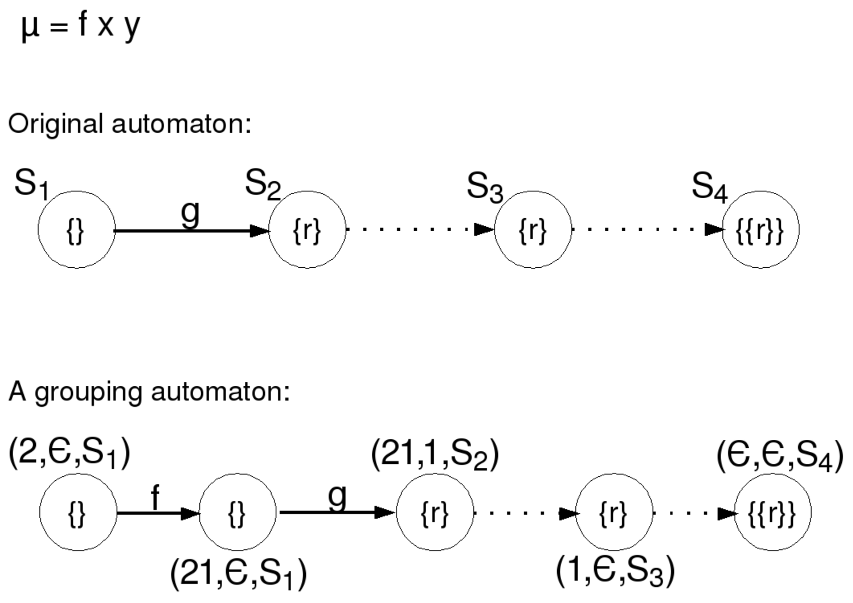

Figure 8.1: Example of grouping

case, no transition is executed on state s by the $\mathcal{M}$ component. If $p = p_{\mu,i}$ with some $i \in [1,n]$, then a transition is executed on state s with the following cases:

1. If s is a step state, then (p,q,s) is a step state that reads either the first symbol of t_i if $q = \varepsilon$, or the symbol t_i @ q if $q \succ \varepsilon$.

2. If s is a skip state, then (p,q,s) is a skip state that skips either the complete term t_i if $q = \varepsilon$, or the complete sub-term $t_i \diamond q$ if $q \succ \varepsilon$.

That is, the relative position q is meaningful only when $p = p_{\mu,i}$ with some $i \in [1,n]$, and in that case, it tells us which symbol of t_i is read by s or which complete sub-term of t_i is skipped by s.

For example, linear 2-tuple $\mu = fxy$ and automaton $\mathcal{M}$ are shown in the top of Figure 8.1 with $\#_f = \#_g = 1$. Then a grouping of $\mathcal{M}$ with respect to μ is shown in the bottom of Figure 8.1.

Formally, given a linear m-tuple μ with free variables $x_n, \ldots, x_1$ and an automaton $\mathcal{M}$ that is defined to be $(Q, s^0, \delta, \sigma, color, \tau)$, we define a construction

$$\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$$

as follows:

1. $s'^0 = (m, \varepsilon, s^0)$.

2. Q' is the least subset of $\textsc{Pos} \times \textsc{Pos} \times Q$ that contains s'^0 and is closed under skip and step functions, as described in the sequel.

3. (p, ε, s) is a step state in $\mathcal{M}'$ if $\mu \mathbin{@} p = b \in \Sigma$. In that case, $color'(p, \varepsilon, s) = \emptyset$ and $\delta'((p, \varepsilon, s), b')$ is:
   - not defined if $b' \neq b$.
   - $(follow(p, b), \varepsilon, s)$ if $b' = b$.

4. (p, q, s) is a step state in $\mathcal{M}'$ if $p = p_{\mu,i}$ with some $i \in [1, n]$ and s is a step state. In that case, $color'(p, q, s) = color(s)$ and $\delta'((p, \varepsilon, s), b)$ is:
   - not defined if $\delta(s, b)$ is not defined.
   - $(p, follow(q, b), \delta(s, b))$ if $\delta(s, b)$ is defined and $follow(q, b) \succ \varepsilon$.
   - $(next(p), \varepsilon, \delta(s, b))$ if $\delta(s, b)$ is defined and $follow(q, b) = \varepsilon$.

5. (p,q,s) is a skip state in $\mathcal{M}'$ if $p = p_{\mu,i}$ with some $i \in [1,n]$ and s is a skip state. In that case, $color'(p,q,s) = color(s)$ and $\sigma'(p,q,s)$ is:
   - $(p, next(q), \sigma(s))$ if $next(q) \succ \varepsilon$.
   - $(next(p), \varepsilon, \sigma(s))$ if $next(q) = \varepsilon$.

6. $(\varepsilon,\varepsilon,s)$ is a terminal state in $\mathcal{M}'$ if s is a terminal state. In that case, $\tau'(p,q,s) = \tau(s)$.

Suppose $\mu\{t\}$ is a ground m-tuple and p is a position of $\mu\{t\}$, we define a function $map_\mu : \textsc{Pos} \to \textsc{Pos}$ that maps p to a position of ground n-tuple t as follows:

- If $p \succeq p_{\mu,n}$, then $map_\mu(p) = n$ (if μ has n free variables).
- If $p = p_{\mu,i}\, w$ with some position w and some $i \in [1,n]$, then $map_\mu(p) = i\, w$.
- If $next(p_{\mu,i}) \succeq p \succeq p_{\mu,i-1}$ with some $i \in [2,n]$, or $next(p_{\mu,i}) \succeq p$ with $i = 1$, then $map_\mu(p) = next(i)$.

The following facts are related to function map_μ.

Lemma 8.1.1 *Suppose $\mu\{t\}$ is a ground m-tuple. Then the following conditions all hold for all positions p of $\mu\{t\}$ with $p \succ \varepsilon$:*

1. *If μ @ $p = b \in \Sigma$ then $map_\mu(follow(p,b)) = map_\mu(p)$.*

2. *If $p = p_{\mu,i}\, w$ with some position w and some $i \in [1,n]$ and $\mu\{t\}$ @ $p = b$, then $map_\mu(follow(p,b)) = follow(map_\mu(p),b)$.*

Proof. We consider two cases:

1. μ @ $p = b \in \Sigma$. Condition (2) holds vacuously in this case. There are two cases:

   (a) $p \succ p_{\mu,n}$. Then $map_\mu(p) = n$ (if μ has n free variables). This leaves two possibilities:

      i. μ @ $(follow(p,b)) \in \Sigma$. Then we have $map_\mu(follow(p,b)) = map_\mu(p) = n$.

      ii. μ @ $(follow(p,b)) = x_n$. Then we have $map_\mu(follow(p,b)) = n$.

   In either case, we have $map_\mu(follow(p,b)) = map_\mu(p)$. Thus, condition (1) holds in this case.

   (b) $next(p_{\mu,i}) \succeq p \succ p_{\mu,i-1}$ with some $i \in [2,n]$, or $next(p_{\mu,i}) \succeq p \succ \varepsilon$ with $i = 1$. Then $map_\mu(p) = next(i)$. This leaves three possibilities:

      i. μ @ $(follow(p,b)) \in \Sigma$. Then we have $map_\mu(follow(p,b)) = map_\mu(p) = next(i)$.

      ii. μ @ $(follow(p,b)) = x_{i-1}$ $(i > 1)$. Then we have

$$map_\mu(follow(p,b)) = i - 1 = next(i) = map_\mu(p).$$

      iii. $follow(p,b) = \varepsilon$. Then $i = 1$ and

$$map_\mu(follow(p,b)) = next(i) = \varepsilon = map_\mu(p).$$

   In either case, we have $map_\mu(follow(p,b)) = map_\mu(p)$. Thus, condition (1) holds in this case.

2. $p = p_{\mu,i}\, w$ with some position w and some $i \in [1,n]$ and $\mu\{t\}$ @ $p = b$. Then condition (1) holds vacuously in this case. Moreover, we have $map_\mu(p) = i\, w$. There are two cases:

   (a) $follow(p,b) \succ next(p_{\mu,i})$. This leaves two possibilities:

      i. $\#_b = 0$. Since $p = p_{\mu,i}\, w$ and $follow(p,b) = next(p) \succ next(p_{\mu,i})$, $next(w) \succ \varepsilon$. Thus, we have $follow(p,b) = p_{\mu,i}\, next(w)$. Since $map_\mu(p) = i\, w$ and $next(w) \succ \varepsilon$, $follow(map_\mu(p),b) = i\, next(w)$.

      ii. $\#_b > 0$. Since $p = p_{\mu,i}\, w$, we have $follow(p,b) = p_{\mu,i}\, w\, \#_b$. Since $map_\mu(p) = i\, w$, $follow(map_\mu(p),b) = i\, w\, \#_b$.

   In either case, if $follow(p,b) = p_{\mu,i}\, w'$ with some position w' and some $i \in [1,n]$, then

$$map_\mu(follow(p,b)) = i\, w' = follow(map_\mu(p),b).$$

   Thus, condition (2) holds in this case.

   (b) $follow(p,b) = next(p_{\mu,i})$. Then $map_\mu(follow(p,b)) = next(i)$. Since $p = p_{\mu,i}\, w$ and $follow(p,b) = next(p_{\mu,i})$, $\#_b = 0$ and $next(w) = \varepsilon$. It follows that $follow(map_\mu(p),b) = next(i\ w) = next(i)$. That is, $map_\mu(follow(p,b)) = follow(map_\mu(p),b) = next(i)$. Thus, condition (2) holds in this case.

□

Following Lemma 8.1.1, if $p = p_{\mu,i}\, w$ with some position w and some $i \in [1,n]$, then $\mu\{t\} \diamond p = t \diamond map_\mu(p)$.

Lemma 8.1.2 *Suppose $\mu\{t\}$ is a ground m-tuple and p is a position of $\mu\{t\}$ with $\mu\{t\}$ @ $p = b$. Suppose further, $p = p_{\mu,i}\, w$ with some position w and some $i \in [1,n]$, and z is a position such that $z \succeq w \succ next(z)$. Then the following conditions all hold:*

1. *$follow(p,b) \succ next(p_{\mu,i}\, z)$ if and only if*

$$follow(map_\mu(p),b) \succ next(map_\mu(p_{\mu,i}\, z)).$$

2. *$follow(p,b) = next(p_{\mu,i}\, z)$ if and only if*

$$follow(map_\mu(p),b) = next(map_\mu(p_{\mu,i}\, z)).$$

Proof. We consider the following cases:

1. $z = \varepsilon$ or $next(z) = \varepsilon$. Then $next(p_{\mu,i}\, z) = next(p_{\mu,i})$ and $next(map_\mu(p_{\mu,i}\, z)) = next(i)$. There are two cases:

   (a) $\#_b = 0$. Then $follow(p,b) = next(p) = next(p_{\mu,i}\, w)$ and

   $$follow(map_\mu(p),b) = follow(i\, w,b) = next(i\, w).$$

   This leaves two possibilities:

i. $next(w) \succ \varepsilon$. Then $follow(p,b) = p_{\mu,i}\, next(w)$ and

$$follow(map_{\mu}(p),b) = follow(i\,w,b) = i\, next(w).$$

Clearly, $follow(p,b) \succ next(p_{\mu,i}) = next(p_{\mu,i}\,z)$ and

$$follow(map_{\mu}(p),b) \succ next(i) = next(map_{\mu}(p_{\mu,i}\,z)).$$

Thus, condition (2) holds vacuously and condition (1) holds in this case.

ii. $w = \varepsilon$ or $next(w) = \varepsilon$. Then $follow(p,b) = next(q)$ and

$$follow(map_{\mu}(p),b) = follow(i\,w,b) = next(i).$$

Clearly, $follow(p,b) = next(p_{\mu,i}) = next(p_{\mu,i}\,z)$ and

$$follow(map_{\mu}(p),b) = next(i) = next(map_{\mu}(p_{\mu,i}\,z)).$$

Thus, condition (1) holds vacuously and condition (2) holds in this case.

(b) $\#_b > 0$. Then $follow(p,b) = p_{\mu,i}\, w\, \#_b \succ next(p_{\mu,i}) = next(p_{\mu,i}\,z)$ and

$$follow(map_{\mu}(p),b) = i\, w\, \#_b \succ next(map_{\mu}(p_{\mu,i}\,z)).$$

Thus, condition (2) holds vacuously and condition (1) holds in this case.

2. $next(z) \succ \varepsilon$. Then $next(p_{\mu,i}\, z) = p_{\mu,i}\, next(z)$ and $next(i\, z) = i\, next(z)$. Since $z \succeq w \succ next(z) \succ \varepsilon$, by Lemma 5.4.2, $next(w) \succeq next(z) \succ \varepsilon$. There are two cases:

   (a) $\#_b = 0$. Then $follow(p,b) = next(p_{\mu,i}\, w)$ and

   $$follow(map_\mu(p),b) = follow(i\, w,b) = next(i\, w).$$

   Since $next(w) \succeq next(z) \succ \varepsilon$, $follow(p,b) = p_{\mu,i}\, next(w)$ and

   $$follow(map_\mu(p),b) = follow(i\, w,b) = i\, next(w).$$

   If $next(w) \succ next(z)$, then

   $$follow(p,b) \succ p_{\mu,i}\, next(z) = next(p_{\mu,i}\, z)$$

   and

   $$follow(map_\mu(p),b) \succ i\, next(z) = next(map_\mu(p_{\mu,i}\, z)).$$

   Thus, condition (2) holds vacuously and condition (1) holds in this case. If $next(w) = next(z)$, then we have $follow(p,b) = p_{\mu,i}\, next(z) =$

$next(p_{\mu,i}\, z)$ and

$$follow(map_\mu(p),b) = i\, next(z) = next(map_\mu(p_{\mu,i}\, z)).$$

Thus, condition (1) holds vacuously and condition (2) holds in this case.

(b) $\#_b > 0$. Then $follow(p,b) = p_{\mu,i}\, w\, \#_b$ and

$$follow(map_\mu(p),b) = i\, w\, \#_b.$$

Since $w \succ next(z)$, we have $follow(p,b) \succ p_{\mu,i}\, next(z) = next(p_{\mu,i}\, z)$ and $follow(map_\mu(p),b) \succ i\, next(z) = next(map_\mu(p_{\mu,i}\, z))$. Thus, condition (2) holds vacuously and condition (1) holds in this case.

$\square$

Now we claim the following facts about the construction mentioned above.

Lemma 8.1.3 *Suppose μ is a linear m-tuple with free variables $x_n,\ldots,x_1$ and $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ is obtained by applying the above construction on automaton $\mathcal{M}$, where $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$. Suppose $\mu\{t\}$ is a ground m-tuple in which t is a ground n-tuple. Then the following condition holds for all positions p of ground m-tuple $\mu\{t\}$:*

- *$C_{\mathcal{M}',\mu\{t\}}(p)$ is defined and is equal to a state (α,β,s) with some positions α and β if and only if $C_{\mathcal{M},t}(map_\mu(p))$ is defined and is equal to s. Moreover, $\alpha\,\beta = p^{(\alpha,\beta,s)}_{\mathcal{M}',\mu\{t\}}$ and $map_\mu(\alpha\,\beta) = p^{s}_{\mathcal{M},t}$.*

Proof. By induction on positions p of ground m-tuple $\mu\{t\}$. If $p = m$, then $C_{\mathcal{M}',\mu\{t\}}(m) = s'^0 = (m, \varepsilon, s^0)$. Moreover, we have $map_\mu(m) = n$ and $C_{\mathcal{M},t}(n) = s^0$. The stated condition holds in this case.

Suppose $p \succ \varepsilon$ is a position of $\mu\{t\}$ such that the stated condition holds and $\mu\{t\}$ @ $p = b$ with some symbol $b \in \Sigma$. We show that the stated condition also holds for position $follow(p,b)$ of $\mu\{t\}$. Suppose $C_{\mathcal{M},\mu\{t\}}(p) = (\alpha, \beta, s)$ with some positions α and β. By induction, $C_{\mathcal{M},t}(map_\mu(p)) = s$ and $\alpha\,\beta = p^{(\alpha,\beta,s)}_{\mathcal{M}',\mu\{t\}}$ and $map_\mu(\alpha\,\beta) = p^s_{\mathcal{M},t}$. We consider the following cases:

1. (α, β, s) is a step state. Then there are two cases:

   (a) μ @ $\alpha = b \in \Sigma$. By the definition of $\mathcal{M}'$, $\beta = \varepsilon$. Since (α, ε, s) is a step state, by Lemma 5.5.1, $p = \alpha = p^{(\alpha,\varepsilon,s)}_{\mathcal{M}',\mu\{t\}}$. That is, μ @ $p = b$. It follows that $C_{\mathcal{M}',\mu\{t\}}(follow(p,b))$ is defined and is equal to $(follow(p,b), \varepsilon, s)$. Since μ @ $p \in \Sigma$, by condition (1) of Lemma 8.1.1, $map_\mu(follow(p,b)) = map_\mu(p)$. Since $C_{\mathcal{M},t}(map_\mu(p)) = s$, $C_{\mathcal{M},t}(map_\mu(follow(p,b))) = s$. Now $follow(p,b) = p^{(follow(p,b),\varepsilon,s)}_{\mathcal{M}',\mu\{t\}}$. Since $map_\mu(\alpha\,\beta) = map_\mu(p) = p^s_{\mathcal{M},t}$ and

   $$map_\mu(follow(p,b)) = map_\mu(p),$$

   $map_\mu(follow(p,b)) = p^s_{\mathcal{M},t}$. Clearly, the stated condition also holds in this case.

   (b) $\alpha = p_{\mu,i}$ with some $i \in [1,n]$ and s is a step state. Since (α, β, s) is a step state, by Lemma 5.5.1, $p = \alpha\,\beta = p^{(\alpha,\beta,s)}_{\mathcal{M}',\mu\{t\}}$. Then we have

$C_{\mathcal{M}',\mu\{t\}}(follow(p,b))$ is defined if and only if $\delta(s,b)$ is defined. Since $C_{\mathcal{M},t}(map_\mu(p)) = s$, $C_{\mathcal{M},t}(follow(map_\mu(p),b))$ is defined if and only if $\delta(s,b)$ is defined. Since $p = \alpha\beta$ with $\alpha = p_{\mu,i}$, by condition (2) of Lemma 8.1.1,

$$follow(map_\mu(p),b) = map_\mu(follow(p,b)).$$

Thus, $C_{\mathcal{M},t}(map_\mu(follow(p,b))$ is defined if and only if $\delta(s,b)$ is defined. It follows that $C_{\mathcal{M}',\mu\{t\}}(follow(p,b))$ is defined if and only if $C_{\mathcal{M},t}(map_\mu(follow(p,b))$ is defined. Suppose $\delta(s,b)$ is defined. This leaves two possibilities:

i. $follow(\beta,b) \succ \varepsilon$. Then we have $follow(p,b) = \alpha\, follow(\beta,b)$ and $C_{\mathcal{M}',\mu\{t\}}(follow(p,b)) = (\alpha, follow(\beta,b), \delta(s,b))$.

ii. $follow(\beta,b) = \varepsilon$. Then $follow(p,b) = next(\alpha)$ and

$$C_{\mathcal{M}',\mu\{t\}}(follow(p,b)) = (next(\alpha), \varepsilon, \delta(s,b)).$$

In either case, we have

$$C_{\mathcal{M},t}(map_\mu(follow(p,b))) = C_{\mathcal{M},t}(follow(map_\mu(p),b)) = \delta(s,b).$$

Now $follow(p,b) = \alpha'\beta' = p^{(\alpha',\beta',\delta(s,b))}_{\mathcal{M}',\mu\{t\}}$ with some positions α' and β'. Moreover, $map_\mu(follow(p,b)) = p^{\delta(s,b)}_{\mathcal{M},t}$. Thus, the stated condition holds in this case.

2. (α, β, s) is a skip state. Then by the definition of $\mathcal{M}'$, $\alpha = p_{\mu,i}$ with some $i \in [1,n]$ and s is a skip state. In this case, $C_{\mathcal{M}',\mu\{t\}}(follow(p,b))$ and $C_{\mathcal{M},t}(map_{\mu}(follow(p,b))$ are defined. There are two cases:

   (a) $follow(p,b) \succ next(\alpha\,\beta)$. Then $C_{\mathcal{M}',\mu\{t\}}(follow(p,b)) = (\alpha, \beta, s)$. Since $follow(p,b) \succ next(\alpha\,\beta)$, by condition (1) of Lemma 8.1.2,

   $$follow(map_{\mu}(p), b) \succ next(map_{\mu}(\alpha\,\beta)).$$

   Since $C_{\mathcal{M},t}(map_{\mu}(p)) = s$,

   $$C_{\mathcal{M},t}(follow(map_{\mu}(p), b)) = s.$$

   Thus, the stated condition holds in this case.

   (b) $follow(p,b) = next(\alpha\,\beta)$. Then

   $$C_{\mathcal{M}',\mu\{t\}}(follow(p,b)) = \sigma'(\alpha, \beta, s) = s',$$

   where s' is either $(\alpha, next(\beta), \sigma(s))$ provided $next(\beta) \succ \varepsilon$, or

   $$(next(\alpha), \varepsilon, \sigma(s))$$

   provided $next(\beta) = \varepsilon$. Since $follow(p,b) = next(\alpha\,\beta)$, by condition

(2) of Lemma 8.1.2,

$$follow(map_{\mu}(p),b) = next(map_{\mu}(\alpha\,\beta)).$$

Since $C_{\mathcal{M},t}(map_{\mu}(p)) = s$,

$$C_{\mathcal{M},t}(follow(map_{\mu}(p),b)) = \sigma(s).$$

Moreover, $follow(p,b) = p^{s'}_{\mathcal{M}',\mu\{t\}}$ and $follow(map_{\mu}(p),b) = p^{\sigma(s)}_{\mathcal{M},t}$. By condition (2) of Lemma 8.1.1,

$$map_{\mu}(follow(p,b)) = follow(map_{\mu}(p),b) = p^{\sigma(s)}_{\mathcal{M},t}.$$

Thus, the stated condition holds in this case.

□

Lemma 8.1.4 *Suppose $\mathcal{M}' = (Q', s'^{0}, \delta', \sigma', color', \tau')$ is obtained by applying the above construction on automaton $\mathcal{M} = (Q, s^{0}, \delta, \sigma, color, \tau)$. Suppose further, t is a ground n-tuple such that $C_{\mathcal{M},t}$ is a computation of $\mathcal{M}$ on t. Then $t \models_{\mathcal{M}} c$ if and only if $\mu\{t\} \models_{\mathcal{M}'} c$.*

Proof. Suppose $t \models_{\mathcal{M}} c$. Then for all positions p and q of t such that $p = p^{s_1}_{\mathcal{M},t} \succ \varepsilon$ and $q = p^{s_2}_{\mathcal{M},t} \succ \varepsilon$ and c occurs at states s_1 and s_2, we have $t \diamond p = t \diamond q$. Since $C_{\mathcal{M},t}$ is a computation of $\mathcal{M}$ on t, by Lemma 8.1.3 and the definition of $\mathcal{M}'$, c occurs at the state $C_{\mathcal{M}',\mu\{t\}}(p') = s'_1 = (\alpha_1, \beta_1, s_1)$ with some positions α_1

and β_1 such that $\alpha_1 = p_{\mu,i}$ with some $i \in [1,n]$, and on the state $C_{\mathcal{M}',\mu\{t\}}(q') = s'_2 = (\alpha_2, \beta_2, s_2)$ with some positions α_2 and β_2 such that $\alpha_2 = p_{\mu,i}$ with some $i \in [1,n]$. Again, by Lemma 8.1.3, $\alpha_1\ \beta_1 = p^{s'_1}_{\mathcal{M}',\mu\{t\}}$ and $map_\mu(\alpha_1\ \beta_1) = p = p^{s_1}_{\mathcal{M},t}$. Similarly, $\alpha_2\ \beta_2 = p^{s'_2}_{\mathcal{M}',\mu\{t\}}$ and $map_\mu(\alpha_2\ \beta_2) = q = p^{s_2}_{\mathcal{M},t}$. Since $\mu\{t\} \diamond \alpha_1\ \beta_1 = t \diamond p$ and $\mu\{t\} \diamond \alpha_2\ \beta_2 = t \diamond q$, we have $\mu\{t\} \diamond \alpha_1\ \beta_1 = \mu\{t\} \diamond \alpha_2\ \beta_2$. It follows that $\mu\{t\} \models_{\mathcal{M}'} c$.

Conversely, suppose $\mu\{t\} \models_{\mathcal{M}'} c$. Then for all positions p and q of $\mu\{t\}$ such that $p = p^{s'_1}_{\mathcal{M}',\mu\{t\}} \succ \varepsilon$ and $q = p^{s'_2}_{\mathcal{M}',\mu\{t\}} \succ \varepsilon$ and c occurs at states s'_1 and s'_2, we have $\mu\{t\} \diamond p = \mu\{t\} \diamond q$. By the definition of $\mathcal{M}'$, c occurs at the state $s'_1 = (\alpha_1, \beta_1, s_1)$ with some positions α_1 and β_1 such that $\alpha_1 = p_{\mu,i}$ with some $i \in [1,n]$, and on the state $s'_2 = (\alpha_2, \beta_2, s_2)$ with some positions α_2 and β_2 such that $\alpha_2 = p_{\mu,i}$ with some $i \in [1,n]$. Moreover, c also occurs at states s_1 and s_2 of $\mathcal{M}$. By Lemma 8.1.3, $p = p^{s'_1}_{\mathcal{M}',\mu\{t\}}$ and $map_\mu(p) = p^{s_1}_{\mathcal{M},t}$. Similarly, $q = p^{s'_2}_{\mathcal{M}',\mu\{t\}}$ and $map_\mu(q) = p^{s_2}_{\mathcal{M},t}$. Since $\mu\{t\} \diamond p = t \diamond map_\mu(p)$ and $\mu\{t\} \diamond q = t \diamond map_\mu(q)$, we have $t \diamond map_\mu(p) = t \diamond map_\mu(q)$. It follows that $t \models_{\mathcal{M}} c$. □

Theorem 8.1.5 *Suppose μ is a linear m-tuple with free variables $x_n, \ldots, x_1$ and $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ is obtained by applying the above construction on automaton $\mathcal{M}$, where $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$. Then $\mathcal{M}'$ is a grouping of $\mathcal{M}$ with respect to μ.*

Proof. Suppose t is a ground n-tuple that is accepted by $\mathcal{M}$. Then $C_{\mathcal{M},t}(\varepsilon)$ is a terminal state s and $t \models_{\mathcal{M}} \tau(s)$. By Lemma 8.1.3, $C_{\mathcal{M}',\mu\{t\}}(\varepsilon)$ is defined and is equal to a terminal state $(\varepsilon, \varepsilon, s)$. Thus, $C_{\mathcal{M}',\mu\{t\}}(\varepsilon)$ is terminating. By the definition of $\mathcal{M}'$, $\tau'(\varepsilon,\varepsilon,s) = \tau(s)$. Following Lemma 8.1.4, we have $\mu\{t\} \models_{\mathcal{M}'} \tau'(\varepsilon,\varepsilon,s)$. Thus, $\mu\{t\}$ is also accepted by $\mathcal{M}'$.

Conversely, suppose $\mu\{t\}$ is a ground m-tuple that is accepted by $\mathcal{M}'$. Then $C_{\mathcal{M}',\mu\{t\}}(\varepsilon)$ is a terminal state $(\varepsilon,\varepsilon,s)$ and $\mu\{t\} \models_{\mathcal{M}'} \tau'(\varepsilon,\varepsilon,s)$. By Lemma 8.1.3, $C_{\mathcal{M},t}(\varepsilon)$ is defined and is equal to terminal state s. Thus, $C_{\mathcal{M},t}(\varepsilon)$ is terminating. By the definition of $\mathcal{M}$, $\tau(s) = \tau'(\varepsilon,\varepsilon,s)$. Following Lemma 8.1.4, we have $t \models_{\mathcal{M}} \tau(s)$. Thus, t is also accepted by $\mathcal{M}$. $\square$

8.2 Ungrouping of Automata

Suppose μ is a linear m-tuple as considered in the definition of grouping of automata. We say an automaton $\mathcal{M}'$ is an ungrouping of automaton $\mathcal{M}$ with respect to m-tuple μ if the following condition holds:

- a ground n-tuple t' is accepted by $\mathcal{M}'$ if and only if ground m-tuple t is accepted by $\mathcal{M}$, where $t = \mu\{t'\}$.

For example, linear 2-tuple $\mu = fxy$ and automaton $\mathcal{M}$ are shown in the top of Figure 8.2 with $\#_f = \#_g = 1$. Then an ungrouping of $\mathcal{M}$ with respect to μ is shown in the bottom of Figure 8.2.

Suppose p is a position of linear m-tuple μ such that $p \succ p_{\mu,n}$, or $next(p_{\mu,i}) \succeq p \succ p_{\mu,i-1}$ with some $i \in (1,n]$, or $next(p_{\mu,1}) \succeq p \succ \varepsilon$. Then we say p is a *non-variable position of* μ.

To obtain an ungrouping of $\mathcal{M}$ with respect to μ, we first construct a stratified automaton $\mathcal{M}_1$ that is a conjunction of $\mathcal{M}^\mu$ and $\mathcal{M}$, where $\mathcal{M}^\mu$ is a stratified

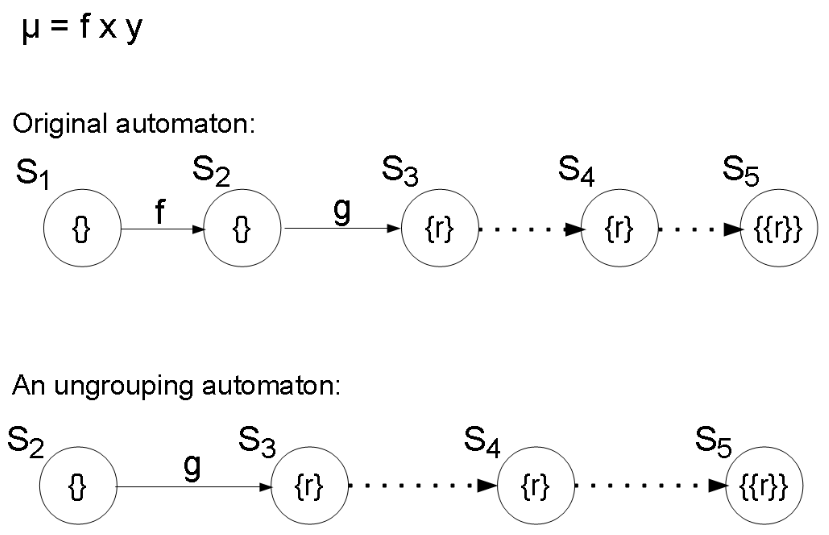

Figure 8.2: Example of ungrouping

automaton for m-tuple μ and use $\mathcal{M}_1$ as the original structure to construct an ungrouping of $\mathcal{M}$. By doing so, we guarantee that $\mathcal{M}_1$ is an automaton that accepts ground m-tuples that are substitution instances of μ. By the definition of $\mathcal{M}^\mu$, $\mathcal{M}^\mu$ contains only one path from the root to the terminal state and for all the states $p \succ \varepsilon$ in $\mathcal{M}^\mu$, p is a position of μ with the following properties:

- If p is non-variable position of μ, then p is a step state. Otherwise p is a skip state.

We assume that $\mathcal{M}_1$ does not contain any non-terminal state s such that there is no terminal state reachable from s. If there exists such state s then we can safely remove it since it does not lead to any terminal state. If $\mathcal{M}_1$ does not have any state, then an ungrouping of $\mathcal{M}$ with respect to μ does not exist. By the definition of conjunction of automata, there is a mapping pos_1 that maps all the states s in $\mathcal{M}_1$ to a position $pos_1(s)$ with the following properties:

1. If $pos_1(s) \succ p_{\mu,n}$ then there is an unique path in $\mathcal{M}_1$ from s to a state s' with $pos_1(s') = p_{\mu,n}$.

2. If $next(p_{\mu,i}) \succeq pos_1(s) \succ p_{\mu,i-1}$ with some $i \in (1,n]$, then there is an unique path in $\mathcal{M}_1$ from s to a state s' such that $pos_1(s') = p_{\mu,i-1}$.

3. If $next(p_{\mu,1}) \succeq pos_1(s)$ then there is an unique path in $\mathcal{M}_1$ from s to a terminal state s' with $pos_1(s') = \varepsilon$.

In the above cases, the states on the path from s to s' (excluding s') are all step states.

We also assume that $\mathcal{M}_1 = (Q_1, s_1^0, \delta_1, \sigma_1, color_1, \tau_1)$ is normalized. This requirement is very important. Consider the example shown in Figure 8.3. Suppose we have a linear 2-tuple $\mu = fxy$ and an automaton A that is shown in Figure 8.3 with $\#_f = 1$. In that figure, s_1 is a step state with $color_1(s_1) = \{r\}$. If we do the ungrouping by skipping the states that correspond to the function symbols or the constants in μ, then we would obtain an automaton B in Figure 8.3. This is not correct, according to the equality constraint on states s_1 and s_3, state s_3 should skip a ground substitution instance of term fx instead of an arbitrary ground term. A correct ungrouping automaton C is shown in Figure 8.3. If $\mathcal{M}_1$ is not normalized, then by using normalization operation mentioned in Theorem 7.4.1, we can always arrange that $\mathcal{M}_1$ is normalized.

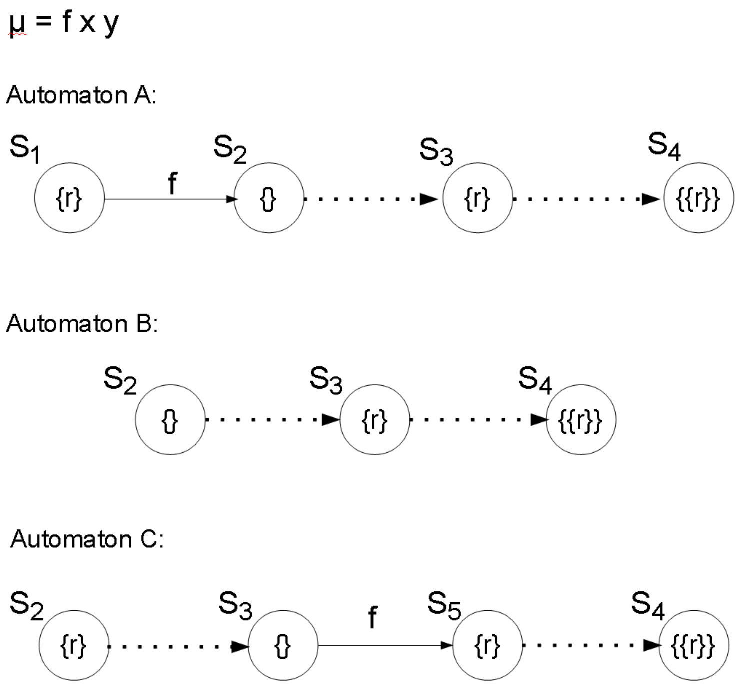

Figure 8.3: Non-empty color sets on step states cause problems in ungrouping

Given a ground m-tuple $t = \mu\{t'\}$ as the input, in order to read ground n-tuple t' from t (skipping other parts from t), we have the following choices when we are at a state s of stratified automaton $\mathcal{M}_1$:

1. If $pos_1(s)$ is a non-variable position of μ, then s is a step state and the current input b is a function symbol or a constant appearing in μ. In this case, b is not a symbol in t'. Thus, we skip symbol b and jump to the next state $\delta_1(s,b)$ to check the next input. By the assumption about $\mathcal{M}_1$ (every non-terminal state leads to a terminal state), $\delta_1(s,b)$ is always defined.

2. If $p_{\mu,i} \succeq pos_1(s) \succ next(p_{\mu,i})$ with some $i \in [1,m]$, then b is a symbol in t'. In this case, we read b or skip b as we do in $\mathcal{M}_1$.

3. If $pos_1(s) = \varepsilon$, then s is a terminal state. In this case, we are done.

Suppose $\mathcal{M}_1 = (Q_1, s_1^0, \delta_1, \sigma_1, color_1, \tau_1)$. For all states s in Q_1, we define a function $jump : Q_1 \to Q_1$ to handle the above cases as follows:

1. If $pos_1(s) \succ \varepsilon$ is not a non-variable position of μ or $pos_1(s) = \varepsilon$, then $jump(s) = s$.

2. If $p = pos_1(s) \succ \varepsilon$ is a non-variable position of μ and μ @ $p = b$, then $jump(s) = jump(\delta_1(s,b))$. Again, by the assumption about $\mathcal{M}_1$ (every non-terminal state leads to a terminal state), $\delta_1(s,b)$ is always defined.

Formally, given a stratified and normalized automaton $\mathcal{M}_1$ such that:

1. $\mathcal{M}_1$ is a conjunction of $\mathcal{M}^\mu$ and $\mathcal{M}$, where $\mathcal{M}^\mu$ is an automaton for linear m-tuple μ.

2. and $\mathcal{M}_1$ is defined to be $(Q_1, s_1^0, \delta_1, \sigma_1, color_1, \tau_1)$ with position assignment pos_1,

we construct $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ as follows:

1. $s'^0 = jump(s_1^0)$.

2. Q' is the least subset of Q_1 that contains s'^0 and is closed under skip and step functions, as described in the sequel.

3. For all state in Q', $color'(s) = color_1(s)$.

4. s is a step state in $\mathcal{M}'$ if s is a step state in $\mathcal{M}_1$. In that case, $\delta'(s,b)$ is $jump(\delta_1(s,b))$.

5. s is a skip state in $\mathcal{M}'$ if s is a skip state in $\mathcal{M}_1$. In that case, $\sigma'(s) = jump(\sigma_1(s))$.

6. s is a terminal state in $\mathcal{M}'$ if s is a terminal state in $\mathcal{M}_1$. In that case, $\tau'(s) = \tau_1(s)$.

We claim the following facts about the above construction.

Lemma 8.2.1 *Suppose stratified and normalized automaton $\mathcal{M}_1$ is a conjunction of $\mathcal{M}^\mu$ and $\mathcal{M}$, where $\mathcal{M}^\mu$ is an automaton for linear m-tuple μ, and $\mathcal{M}_1$ is defined to be $(Q_1, s_1^0, \delta_1, \sigma_1, color_1, \tau_1)$. Suppose further, $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ is obtained by applying the above construction on automaton $\mathcal{M}_1$. Then $\mathcal{M}'$ is also stratified with a position assignment pos' and for all the states s of $\mathcal{M}'$ that are reachable from s'^0, $pos'(s) = map_\mu(pos_1(s))$.*

Proof. Since Q' is a subset of Q, it is easy to verify that $\mathcal{M}'$ is also stratified with a position assignment pos' by following the definition of $\mathcal{M}'$. By the definition of $\mathcal{M}'$, for all states s in $\mathcal{M}'$, $s = jump(s')$ with some state s' in $\mathcal{M}_1$. By the definition of $jump$, if s is a non-terminal state in $\mathcal{M}_1$ then $pos_1(s)$ is not a non-variable position of μ, if s is a terminal state in $\mathcal{M}_1$, then $pos_1(s) = \varepsilon$. In either case, we have $pos'(s) = map_\mu(pos_1(s))$ for all the states s in $\mathcal{M}'$ starting from $pos'(s'^0) = map_\mu(pos_1(jump(s^0))) = p_{\mu,n}$. $\square$

Lemma 8.2.2 *Suppose stratified and normalized automaton $\mathcal{M}_1$ is a conjunction of $\mathcal{M}^{\mu}$ and $\mathcal{M}$, where $\mathcal{M}^{\mu}$ is an automaton for linear m-tuple μ, and $\mathcal{M}_1$ is defined to be $(Q_1, s_1^0, \delta_1, \sigma_1, color_1, \tau_1)$. Suppose further, $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ is obtained by applying the above construction on automaton $\mathcal{M}_1$. Then the following condition holds for all positions p of a ground n-tuple t:*

- *$C_{\mathcal{M}',t}(p)$ is defined and is equal to a state s if and only if $C_{\mathcal{M}_1,\mu\{t\}}(p')$ is defined and is equal to s, where p' is the least position of $\mu\{t\}$ with $p = map_{\mu}(p')$.*

Proof. By induction on positions p of ground n-tuple t. If $p = n$, then we have $C_{\mathcal{M}',t}(n) = s'^0 = jump(s_1^0) = C_{\mathcal{M}_1,\mu\{t\}}(p_{\mu,n})$. Clearly, the stated condition holds in this case.

Suppose $p \succ \varepsilon$ is a position of t such that the stated condition holds and $t \mathbin{@} p = b$ with some symbol $b \in \Sigma$. We show that the stated condition also holds for position $follow(p,b)$ of t. Suppose $C_{\mathcal{M}',t}(p) = s$. By induction, $C_{\mathcal{M}_1,\mu\{t\}}(p') = s$ and p' is the least position of $\mu\{t\}$ with $p = map_{\mu}(p')$. Since $p \succ \varepsilon$, s is a non-terminal state in $\mathcal{M}'$ and $p = i\,q$ with some $i \in [1,m]$ and some position q. By Lemma 8.2.1, $p' = p_{\mu,i}\,q$. We consider the following cases:

1. $follow(p,b) \succ next(i)$. By condition (1) of Lemma 8.1.2,

$$follow(p',b) \succ next(p_{\mu,i}).$$

In this case, after state s reads or skips symbol b, the complete term $t \diamond i$ has not been read or skipped in automaton $\mathcal{M}'$, and the complete term $\mu\{t\} \diamond p_{\mu,i}$

has not been read or skipped in automaton $\mathcal{M}_1$. By the definition of $\mathcal{M}'$ and the definition of *jump*, $C_{\mathcal{M}',t}(follow(p,b))$ is defined and is equal to a state s' if and only if $C_{\mathcal{M}_1,\mu\{t\}}(follow(p',b))$ is defined and is equal to s'. It follows that the stated condition holds in this case.

2. $follow(p,b) = next(i)$. By condition (2) of Lemma 8.1.2,

$$follow(p',b) = next(p_{\mu,i}).$$

In this case, after state s reads or skips symbol b, the complete term $t \diamond i$ has been read or skipped in automaton $\mathcal{M}'$, and the complete term $\mu\{t\} \diamond p_{\mu,i}$ has been read or skipped in automaton $\mathcal{M}_1$. Thus, $C_{\mathcal{M}',t}(next(i))$ is defined and is equal to a state $s'' = jump(s')$ if and only if $C_{\mathcal{M}_1,\mu\{t\}}(next(p_{\mu,i}))$ is defined and is equal to s'. Moreover, $C_{\mathcal{M}_1,\mu\{t\}}(next(p_{\mu,i}))$ is defined and is equal to s' if and only if $C_{\mathcal{M}_1,\mu\{t\}}(p'')$ is defined and is equal to s'', where p'' is either $p_{\mu,i-1}$ $(i > 1)$ or ε $(i = 1)$. It follows that $C_{\mathcal{M}',t}(next(i))$ is defined and is equal to s'' if and only if $C_{\mathcal{M}_1,\mu\{t\}}(p'')$ is defined and is equal to s'', where p'' is either $p_{\mu,i-1}$ $(i > 1)$ or ε $(i = 1)$. Clearly, the stated condition holds in this case.

□

Lemma 8.2.3 *Suppose stratified and normalized automaton $\mathcal{M}_1$ is a conjunction of $\mathcal{M}^\mu$ and $\mathcal{M}$, where $\mathcal{M}^\mu$ is an automaton for linear m-tuple μ, and $\mathcal{M}_1$ is defined to be $(Q_1, s_1^0, \delta_1, \sigma_1, color_1, \tau_1)$. Suppose further, $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ is* *obtained by applying the above construction on automaton $\mathcal{M}_1$ and t is a ground n-tuple such that $C_{\mathcal{M}',t}$ is a computation of $\mathcal{M}'$ on t. Then $t \models_{\mathcal{M}'} c$ if and only if $\mu\{t\} \models_{\mathcal{M}_1} c$.*

Proof. Suppose $t \models_{\mathcal{M}'} c$. Then for all positions p and q of t such that $p = p^{s_1}_{\mathcal{M}',t} \succ \varepsilon$ and $q = p^{s_2}_{\mathcal{M}',t} \succ \varepsilon$ and c occurs at states s_1 and s_2 of $\mathcal{M}'$, we have $t \diamond p = t \diamond q$. By the definition of $\mathcal{M}'$, c also occurs at the states s_1 and s_2 of $\mathcal{M}_1$. By Lemma 8.2.2, $map_\mu(p^{s_1}_{\mathcal{M}_1,\mu\{t\}}) = p^{s_1}_{\mathcal{M}',t} = p$ and $map_\mu(p^{s_2}_{\mathcal{M}_1,\mu\{t\}}) = p^{s_2}_{\mathcal{M}',t} = q$. Thus, $\mu\{t\} \diamond p^{s_1}_{\mathcal{M}_1,\mu\{t\}} = t \diamond p$ and $\mu\{t\} \diamond p^{s_2}_{\mathcal{M}_1,\mu\{t\}} = t \diamond q$. Clearly, we have $\mu\{t\} \diamond p^{s_1}_{\mathcal{M}_1,\mu\{t\}} = \mu\{t\} \diamond p^{s_2}_{\mathcal{M}_1,\mu\{t\}}$. It follows that $\mu\{t\} \models_{\mathcal{M}_1} c$.

Conversely, suppose $\mu\{t\} \models_{\mathcal{M}_1} c$. Then for all positions p and q of $\mu\{t\}$ such that $p = p^{s_1}_{\mathcal{M}_1,\mu\{t\}} \succ \varepsilon$ and $q = p^{s_2}_{\mathcal{M}_1,\mu\{t\}} \succ \varepsilon$ and c occurs at states s_1 and s_2 of $\mathcal{M}_1$, we have $\mu\{t\} \diamond p = \mu\{t\} \diamond q$. Since $\mathcal{M}_1$ is normalized, both s_1 and s_2 are skip states. Thus, $p = p_{\mu,i}\, w_1$ with some $i \in [1,m]$ and $q = p_{\mu,j}\, w_2$ with some $j \in [1,m]$. It follows that $jump(s_1) = s_1$ and $jump(s_2) = s_2$. Thus, by the definition of $\mathcal{M}'$, c also occurs at the states s_1 and s_2 of $\mathcal{M}'$. By Lemma 8.2.2, $map_\mu(p^{s_1}_{\mathcal{M}_1,\mu\{t\}}) = map_\mu(p) = p^{s_1}_{\mathcal{M}',t}$ and $map_\mu(p^{s_2}_{\mathcal{M}_1,\mu\{t\}}) = map_\mu(q) = p^{s_2}_{\mathcal{M}',t}$. Thus, $\mu\{t\} \diamond p = t \diamond p^{s_1}_{\mathcal{M}',t}$ and $\mu\{t\} \diamond q = t \diamond p^{s_2}_{\mathcal{M}',t}$. Clearly, we have $t \diamond p^{s_1}_{\mathcal{M}',t} = t \diamond p^{s_2}_{\mathcal{M}',t}$. It follows that $t \models_{\mathcal{M}'} c$. □

Theorem 8.2.4 *Suppose $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ is obtained by applying the above construction on automaton $\mathcal{M}_1$, where $\mathcal{M}_1$ is defined to be*

$$(Q_1, s_1^0, \delta_1, \sigma_1, color_1, \tau_1)$$

and has the following properties:

1. *$\mathcal{M}_1$ is a conjunction of $\mathcal{M}^\mu$ and $\mathcal{M}$, where $\mathcal{M}^\mu$ is an automaton for linear m-tuple μ.*

2. *and $\mathcal{M}_1$ is defined to be $(Q_1, s_1^0, \delta_1, \sigma_1, color_1, \tau_1)$ with position assignment pos_1,*

Then $\mathcal{M}'$ is an ungrouping of $\mathcal{M}_1$ with respect to μ.

Proof. Suppose t is a ground n-tuple that is accepted by $\mathcal{M}'$. Then $C_{\mathcal{M}',t}(\varepsilon)$ is a terminal state s of $\mathcal{M}'$ and $t \models_{\mathcal{M}'} \tau'(s)$. By Lemma 8.2.2, $C_{\mathcal{M}_1,\mu\{t\}}(\varepsilon)$ is defined and is equal to s since ε is the least position of $\mu\{t\}$ with $map_\mu(\varepsilon) = \varepsilon$. By the definition of $\mathcal{M}'$, s is also a terminal state of $\mathcal{M}_1$. Thus, $C_{\mathcal{M}_1,\mu\{t\}}(\varepsilon)$ is terminating. By the definition of $\mathcal{M}'$, $\tau'(s) = \tau_1(s)$. Following Lemma 8.2.3, we have $\mu\{t\} \models_{\mathcal{M}_1} \tau_1(s)$. Thus, $\mu\{t\}$ is accepted by $\mathcal{M}_1$.

Conversely, suppose $\mu\{t\}$ is a ground m-tuple that is accepted by $\mathcal{M}_1$. Then $C_{\mathcal{M}_1,\mu\{t\}}(\varepsilon)$ is a terminal state s of $\mathcal{M}_1$ and $t \models_{\mathcal{M}_1} \tau_1(s)$. By Lemma 8.2.2, $C_{\mathcal{M}',t}(\varepsilon)$ is defined and is equal to s since ε is the least position of $\mu\{t\}$ with $map_\mu(\varepsilon) = \varepsilon$. By the definition of $\mathcal{M}'$, s is also a terminal state of $\mathcal{M}'$. Thus, $C_{\mathcal{M}',t}(\varepsilon)$ is terminating. By the definition of $\mathcal{M}'$, $\tau'(s) = \tau_1(s)$. Following Lemma 8.2.3, we have $t \models_{\mathcal{M}'} \tau'(s)$. It follows that t is accepted by $\mathcal{M}'$. □

Chapter 9

Expansion and Projection of Automata

9.1 Expansion of Automata

We say an automaton $\mathcal{M}'$ is *an expansion* of automaton $\mathcal{M}$ with respect to integer m if the following condition holds:

- a ground m-tuple $t_m \ldots t_1$ is accepted by $\mathcal{M}'$ if and only if ground n-tuple $t_n \ldots t_1$ is accepted by $\mathcal{M}$ $(m > n)$.

To obtain an expansion of $\mathcal{M}$ with respect to m, we simply create a sequence of $(m-n)$ skip states such that each one of them skips a complete sub-terms and the last skip state reaches the initial state of $\mathcal{M}$. Given an automaton $\mathcal{M}$ that is defined to be a tuple $(Q, s^0, \delta, \sigma, color, \tau)$, we define a construction $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ as follows:

1. $Q' = Q \cup \{s_m, \ldots, s_{n+1}\}$.

2. $s'^0 = s_m$.

3. For all i in $[n+1, m]$, s_i is a skip state in Q' and $\sigma'(s_i) = s_{i-1}$ and $color'(s_i) = \emptyset$. In addition, $\sigma'(s_{n+1}) = s^0$.

4. s is a skip state in Q' if s is a skip state in Q. In that case, $\sigma'(s) = \sigma(s)$ and $color'(s) = color(s)$.

5. s is a step state in Q' if s is a step state in Q. In that case, if $\delta(s,b)$ is defined for some symbol $b \in \Sigma$, then $\delta'(s,b) = \delta(s,b)$, otherwise $\delta'(s,b)$ is not defined. In addition, $color'(s) = color(s)$.

6. s is a terminal state in Q' if s is a terminal state in Q. In that case, $\tau'(s) = \tau(s)$.

We claim the following facts about the above construction.

Lemma 9.1.1 *Suppose $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ is an automaton obtained by applying the above construction on an automaton $\mathcal{M}$, where*

$$\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$$

and $Q' = Q \cup \{s_m, \ldots, s_{n+1}\}$ $(n < m)$. Suppose further, $t' = t_m \ldots t_1$ is a ground m-tuple. Then the following conditions all hold for all positions p of t':

1. *If $p = i\,\alpha$ with some $i \in [n+1, m]$, then $C_{\mathcal{M}',t'}(p)$ is defined and is equal to s_i.*
2. *If $n \succeq p$, then $C_{\mathcal{M}',t'}(p)$ is defined and is equal to state s if and only if $C_{\mathcal{M},t_n \ldots t_1}(p)$ is defined and is equal to s.*

Proof. We verify this by induction on all positions p of t' that the stated conditions all hold for p. Let $t = t_n \ldots t_1$. Suppose $p = m$. Then $p \succ n$. By the definition of $\mathcal{M}'$, $C_{\mathcal{M}',t'}(p) = s_m$. It follows that condition (1) holds and condition (2) holds vacuously in this case.

Suppose p is a position of t' such that $p \succ \varepsilon$ and the stated conditions all hold for p and t' @ $p = b$. We show that the stated conditions all hold for position $follow(p, b)$ of t'. We consider the following cases:

1. $p = i\,\alpha$ with some $i \in [n+1,m]$. By condition (1) of induction hypothesis, $C_{\mathcal{M}',t'}(p) = s_i$. By the definition of $\mathcal{M}'$, s_i is a skip state. In this case,

$$C_{\mathcal{M}',t'}(follow(p,b))$$

is always defined. Suppose p' is the greatest position of t' with $C_{\mathcal{M}',t'}(p') = s_i$. By condition (1) of the induction hypothesis, $p' = i$. There are two cases:

   - $follow(p,b) \succ next(p')$. Then $C_{\mathcal{M}',t'}(follow(p,b)) = s_i$. Since

     $$follow(p,b) \succ next(p')$$

     and $p' = i$, we have $follow(p,b) \succ next(i)$. Thus, $follow(p,b) = i\,\beta$ with some position β. It follows that condition (1) holds and condition (2) holds vacuously in this case.
   - $follow(p,b) = next(p') = next(i)$. Then

     $$C_{\mathcal{M}',t'}(follow(p,b)) = \sigma'(s_i).$$

     This leaves two possibilities:

     (a) $i > n+1$. Then $follow(p,b) = i-1$ and $\sigma'(s_i) = s_{i-1}$. It follows that condition (1) holds and condition (2) holds vacuously in this case.

     (b) $i = n+1$. Then $follow(p,b) = n$ and $\sigma'(s_i) = s^0$. In this case,

$C_{\mathcal{M},t}(follow(p,b)) = s^0$. It follows that condition (1) holds vacuously and condition (2) holds in this case.

2. $n \succeq p$. Suppose $C_{\mathcal{M}',t'}(p) = s$. By condition (2) of the induction hypothesis, we have $C_{\mathcal{M},t}(p) = s$. In this case, $n \succ follow(p,b)$ and s is a state in $\mathcal{M}$. Thus, by the definition of $\mathcal{M}'$, $C_{\mathcal{M}',t'}(follow(p,b))$ is defined and is equal to s' if and only if $C_{\mathcal{M},t}(follow(p,b))$ is defined and is equal to s', where s' is either s or a state that is directly reachable from s via a step or a skip function. It follows that condition (1) holds vacuously and condition (2) holds in this case.

□

Lemma 9.1.2 *Suppose* $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ *is an automaton obtained by applying the above construction on an automaton* $\mathcal{M}$, *where*

$$\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$$

and $Q' = Q \cup \{s_m, \ldots, s_{n+1}\}$ $(n < m)$. *Suppose further,* c *is a color and* $t_m \ldots t_1$ *is a ground m-tuple such that* $C_{\mathcal{M}',t_m \ldots t_1}$ *is a computation of* $\mathcal{M}'$ *on* $t_m \ldots t_1$. *Then* $t_m \ldots t_1 \models_{\mathcal{M}'} c$ *if and only if* $t_n \ldots t_1 \models_{\mathcal{M}} c$.

Proof. Suppose $t_m \ldots t_1 \models_{\mathcal{M}'} c$. Then for all positions p and q of $t_m \ldots t_1$ such that $p = p^{s_1}_{\mathcal{M}',t_m \ldots t_1} \succ \varepsilon$ and $q = p^{s_2}_{\mathcal{M}',t_m \ldots t_1} \succ \varepsilon$ and c occurs at states s_1 and s_2 of $\mathcal{M}'$, we have $t_m \ldots t_1 \diamond p = t_m \ldots t_1 \diamond q$. By the definition of $\mathcal{M}'$, c also occurs at the states s_1 and s_2 of $\mathcal{M}$. By Lemma 9.1.1, $n \succeq p$ and $n \succeq q$. It is easy to verify that $p = p^{s_1}_{\mathcal{M},t_n\ldots t_1}$ and $q = p^{s_2}_{\mathcal{M},t_n\ldots t_1}$. Thus, $t_m\ldots t_1 \diamond p = t_n\ldots t_1 \diamond p$ and $t_m\ldots_1 \diamond q = t_n\ldots t_1 \diamond q$. Clearly, we have $t_n\ldots t_1 \diamond p = t_n\ldots t_1 \diamond q$. It follows that $t_n\ldots t_1 \models_{\mathcal{M}} c$.

Conversely, suppose $t_n\ldots t_1 \models_{\mathcal{M}} c$. Then for all positions p and q of $t_n\ldots t_1$ such that $p = p^{s_1}_{\mathcal{M},t_n\ldots t_1} \succ \varepsilon$ and $q = p^{s_2}_{\mathcal{M},t_n\ldots t_1} \succ \varepsilon$ and c occurs at states s_1 and s_2 of $\mathcal{M}'$, we have $t_n\ldots t_1 \diamond p = t_n\ldots t_1 \diamond q$. By the definition of $\mathcal{M}'$, c also occurs at the states s_1 and s_2 of $\mathcal{M}'$. It is easy to verify that $p = p^{s_1}_{\mathcal{M}',t_m\ldots t_1}$ and $q = p^{s_2}_{\mathcal{M}',t_m\ldots t_1}$. Thus, $t_m\ldots t_1 \diamond p = t_n\ldots t_1 \diamond p$ and $t_m\ldots_1 \diamond q = t_n\ldots t_1 \diamond q$. Clearly, we have $t_m\ldots t_1 \diamond p = t_m\ldots t_1 \diamond q$. It follows that $t_m\ldots t_1 \models_{\mathcal{M}'} c$. □

Theorem 9.1.3 *Suppose automaton $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ is obtained by applying the above construction on automaton $\mathcal{M}$ that is defined to be*

$$(Q, s^0, \delta, \sigma, color, \tau).$$

Then $\mathcal{M}'$ is an expansion of $\mathcal{M}$ with respect to m.

Proof. Let n be an integer that is less than m. Suppose ground m-tuple $t_m\ldots t_1$ is accepted $\mathcal{M}'$. Then $C_{\mathcal{M}',t_m\ldots t_1}(\varepsilon) = s$ and s is a terminal state and $t_m\ldots t_1 \models_{\mathcal{M}'} \tau'(s)$. By condition (2) of Lemma 9.1.1, we have $C_{\mathcal{M},t_n\ldots t_1}(\varepsilon) = s$. By the definition of $\mathcal{M}'$, s is a terminal state in $\mathcal{M}$ and $\tau(s) = \tau'(s)$. Thus, $C_{\mathcal{M},t_n\ldots t_1}(\varepsilon)$ is terminating. Following 9.1.2, we have $t_n\ldots t_1 \models_{\mathcal{M}} \tau(s)$. It follows that ground n-tuple $t_n\ldots t_1$ is accepted by $\mathcal{M}$.

Conversely, suppose ground n-tuple $t_n\ldots t_1$ is accepted $\mathcal{M}$. Then we have $C_{\mathcal{M},t_n\ldots t_1}(\varepsilon) = s$ and s is a terminal state and $t_n\ldots t_1 \models_{\mathcal{M}} \tau(s)$. Suppose $t_m, \ldots, t_{n+1}$

are arbitrary $(m-n)$ ground sub-terms. By condition (2) of Lemma 9.1.1,

$$C_{\mathcal{M}',t_m\ldots t_1}(\varepsilon) = C_{\mathcal{M},t}(\varepsilon) = s.$$

By the definition of $\mathcal{M}'$, s is a terminal state in $\mathcal{M}'$ and $\tau'(s) = \tau(s)$. Thus, $C_{\mathcal{M}',t_m\ldots t_1}(\varepsilon)$ is terminating. Following Lemma 9.1.2, we have $t_m \ldots t_1 \models_{\mathcal{M}'} \tau'(s)$. It follows that ground m-tuple $t_m \ldots t_1$ is accepted by $\mathcal{M}'$. $\square$

9.2 Projection of Automata

We say an automaton $\mathcal{M}'$ is *a projection* of automaton $\mathcal{M}$ if the following condition holds:

- a ground $(n-1)$-tuple $t_n \ldots t_2$ is accepted by $\mathcal{M}'$ if and only if there exists a term t_1 such that ground n-tuple $t_n \ldots t_1$ is accepted by $\mathcal{M}$.

We can define a projection construction assuming that

$$\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$$

is stratified. In this case, there is a mapping *pos* that maps a state in Q to a position. Suppose s is a state in $\mathcal{M}$. We denote the set of all the paths that start from s and end at a terminal state in $\mathcal{M}$ by Δ_s. Suppose ξ is such a path, we denote the terminal state on ξ by $term(\xi)$. We define an equivalence relation $\equiv_{(\xi,C)}$ on colors with respect to a path $\xi \in \Delta_s$ and a set C of colors as follows:

- We say $c_1 \equiv_{(\xi,C)} c_2$ if:

  1. both c_1 and c_2 occur at some non-terminal state s_1 in ξ,
  2. and both c_1 and c_2 are in the same set C of $\tau(term(\xi))$.

We denote the equivalence class of color c under the above equivalence relation $\equiv_{(\xi,C)}$ by $E^c_{(\xi,C)}$. For each class $E^c_{(\xi,C)}$, we associate a color with it and denote that color by $\upsilon^c_{(\xi,C)}$. Moreover, we denote the set $\{\upsilon^c_{(\xi,C)} \mid c \in C\}$ by C_ξ.

Informally, to perform a projection operation on $\mathcal{M}$, we use the following naive method:

- For all the states s such that $pos(s) \succ 1$, we build the new color set $color'(s)$ of s such that for each color c in $color(s)$ and for each successor s' of s with $pos(s') = 1$ and for each path $\xi \in \Delta_{s'}$ and for each set C in $\tau(term(\xi))$ that contains c, we add the color $\upsilon^c_{(\xi,C)}$ in $color'(s)$.
- For all the states s in $\mathcal{M}$ such that $pos(s) = 1$, we change s to a terminal state and build new acceptance condition $\tau'(s)$ such that for each path ξ that is in Δ_s and for each set C in $\tau(term(\xi))$, we add the set C_ξ to $\tau'(s)$.

Now we explain the usefulness of equivalence class $E^c_{(\xi,C)}$ and the associated color $\upsilon^c_{(\xi,C)}$. We consider an automaton A shown in Figure 9.1. In that figure, if colors r and g are satisfied by the input, then the ground sub-terms skipped at states s_1, s_2 and s_3 must be identical. If we remove state s_4 and turn state s_3 into a terminal state simply with $\tau'(s_3) = \tau(s_4)$, then we would obtain an automaton B in Figure 9.1. Clearly, the resulting automaton is not correct since the equality constraint on states s_1 and s_2 is lost. To solve this problem, we should treat colors

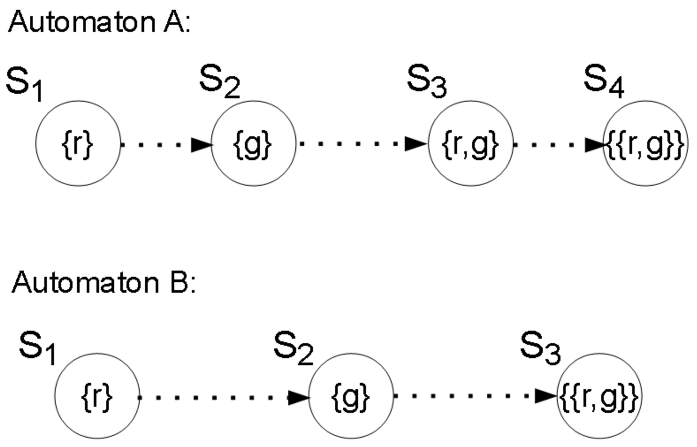

Figure 9.1: Colors occur at the same state

r and g as a single color since they occur at the same state and also appear in the same set of an acceptance condition.

We consider another example. An automaton A is shown in Figure 9.2 with $\#_f = \#_g = 2$. State s_3 will be changed into a terminal state in projection operation. We also change the colors on all the states to obtain an automaton B as shown in Figure 9.2. Note that, in projection operation, we do not perform any changes on the successors of state s_3 since those states will be removed. Color r_1 stands for $\upsilon^r_{(\xi_1,C')}$ and b_1 stands for $\upsilon^b_{(\xi_1,C')}$ and w_2 is for $\upsilon^r_{(\xi_2,C')}$ $(\upsilon^b_{(\xi_2,C')})$, where $C' = \{r, b\}$ and $\xi_1 = s_3 s_4 s_6 s_7$ and $\xi_2 = s_3 s_5 s_6 s_7$. After performing projection operation, we obtain an automaton C in Figure 9.2, which is indeed a projection of the original automaton.

To guarantee that the naive method is correct, we also assume that $\mathcal{M}$ is normalized. This requirement is very important. We consider an automaton A shown

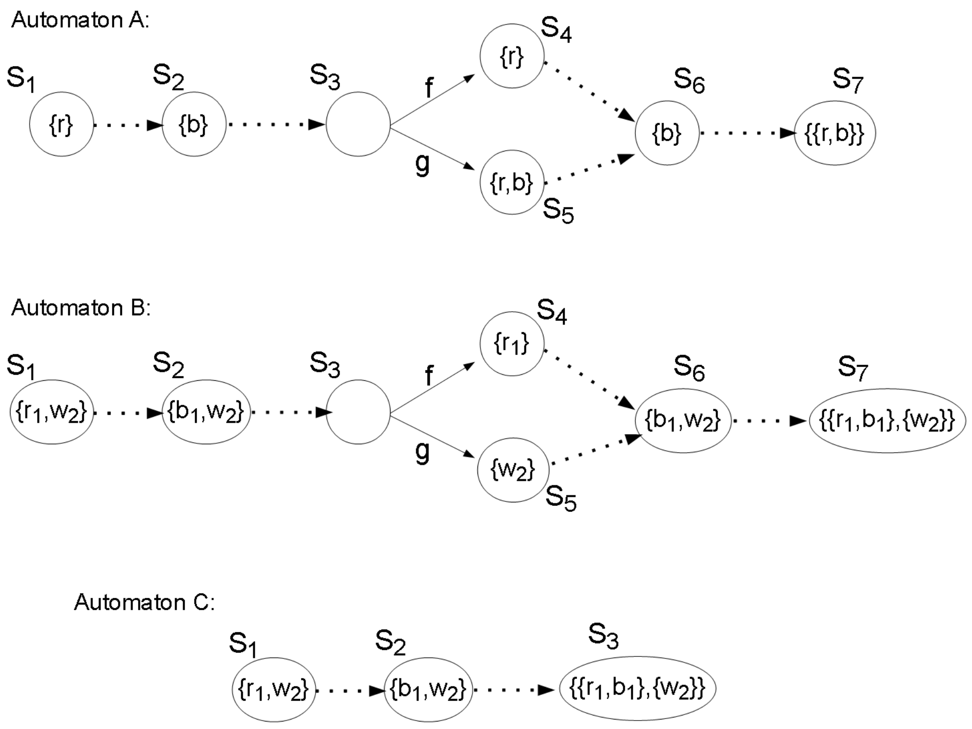

Figure 9.2: Changing colors in a projection operation

in Figure 9.3 with $\#_f = 1$. In that figure, $color(s_3) = \{r\}$ for step state s_3. If we remove states s_4 and s_5 and turn state s_3 into a terminal state (with $\tau'(s_3) = \tau(s_4)$) as described in the naive method, then we would obtain an automaton B in Figure 9.3. This is not correct, according to the equality constraint on states s_1, s_2, and s_3, states s_1 and s_2 should skip a ground substitution instance of term fx instead of an arbitrary ground term. A correct projection automaton C is shown in Figure 9.3. By Theorem 7.4.1, we can always arrange that $\mathcal{M}$ is normalized.

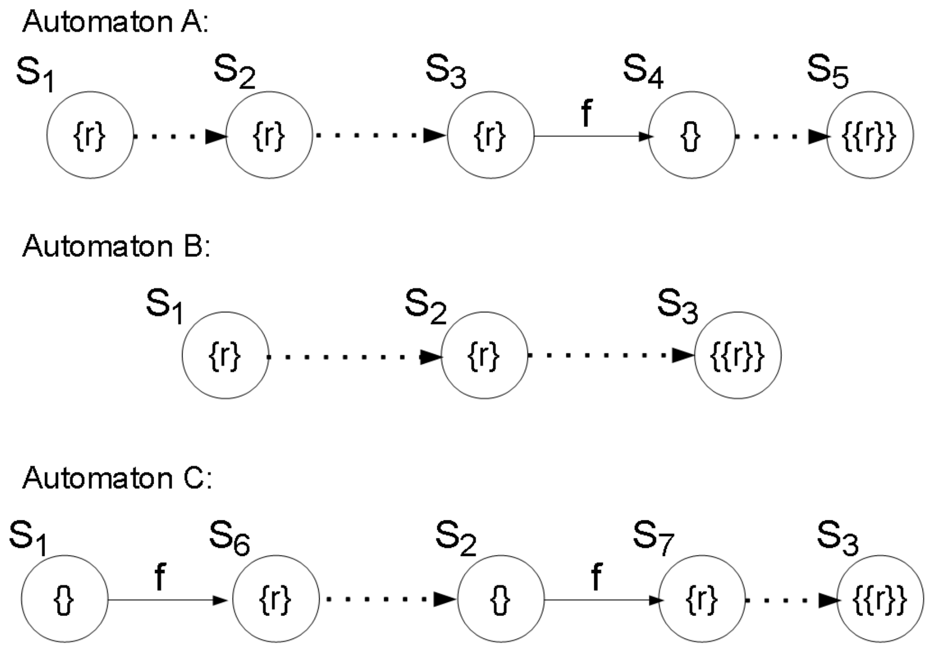

Figure 9.3: Non-empty color sets on step states cause problems in projection

Formally, suppose $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is a normalized automaton with a position assignment *pos*. We construct $\mathcal{M}'$, which is a tuple

$$(Q', s'^0, \delta', \sigma', color', \tau'),$$

as follows:

1. $s'^0 = s^0$.

2. Q' is the least subset of Q that contains s'^0 and is closed under skip and step functions, as described in the sequel.

3. For all non-terminal states s in Q', $color'(s) = C_{s_1} \cup \ldots \cup C_{s_m}$, where $\{s_1, \ldots, s_m\}$ is the set of all the successors s' of s in $\mathcal{M}$ with $pos(s') = 1$ and for all $i \in [1,m]$, C_{s_i} is

$$\{\upsilon^{c}_{(\xi,C)} \mid \xi \in \Delta_{s_i} \text{ and } C \in \tau(term(\xi)) \text{ and } c \in color(s) \cap C\}.$$

4. s is a step state in $\mathcal{M}'$ if $pos(s) \succ 1$ and s is a step state in $\mathcal{M}$. In that case, $\delta'(s,b)$ is:
   - not defined if $\delta(s,b)$ is not defined.
   - $\delta(s,b)$ if $\delta(s,b)$ is defined.

5. s is a skip state in $\mathcal{M}'$ if $pos(s) \succ 1$ and s is a skip state in $\mathcal{M}$. In that case, $\sigma'(s) = \sigma_1(s)$.

6. s is a terminal state in $\mathcal{M}'$ if s is a state in $\mathcal{M}$ with $pos(s) = 1$. Moreover, $\tau'(s) = S_{s_1} \cup \ldots \cup S_{s_m}$, where $s_1, \ldots, s_m$ are all the terminal states in $\mathcal{M}$ that are reachable from s and for all $i \in [1,m]$, $S_{s_i} = \bigcup_{\xi \in \Delta} \{C_\xi \mid C \in \tau(s_i)\}$ in which Δ is the set of all the paths in Δ_s that end at s_i.

We define a function $map_\downarrow : \textsc{Pos} \to \textsc{Pos}$ as follows:

$$map_\downarrow(i\,\alpha) = \begin{cases} (i-1)\,\alpha & \text{if } i > 1 \\ \varepsilon & \text{if } i = 1 \text{ and } \alpha = \varepsilon \\ \text{undefined} & \text{otherwise} \end{cases}$$

Now we claim the following facts about the above construction.

Lemma 9.2.1 *Suppose $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ is obtained by applying the above construction on normalized automaton $\mathcal{M}$. Then the following condition holds for all positions p of a ground n-tuple $t_n \ldots t_1$ with $p \succeq 1$:*

- *$C_{\mathcal{M}_1, t_n \ldots t_1}(p)$ is defined and is equal to a state s in $\mathcal{M}_1$ if and only if*

$$C_{\mathcal{M}', t_n \ldots t_2}(map_{\downarrow}(p))$$

is defined and is equal to state s in $\mathcal{M}'$. Moreover, $map_{\downarrow}(p^s_{\mathcal{M}, t_n \ldots t_1}) = p^s_{\mathcal{M}', t_n \ldots t_2}$.

Proof. We verify this by induction on positions p of $t_n \ldots t_1$ with $p \succeq 1$ that the stated condition holds for p. Suppose $p = n$. Then $C_{\mathcal{M}_1, t_n \ldots t_1}(n) = s^0$. By the definition of $\mathcal{M}'$, $C_{\mathcal{M}', t_n \ldots t_2}(n-1) = s^0$. Thus, the stated condition holds in this case.

Suppose p is a position of $t_n \ldots t_1$ such that $p \succ 1$ and the stated condition holds for p and $(t_n \ldots t_1)$ @ $p = b$. We show that the stated conditions all hold for position $follow(p, b)$ of $t_n \ldots t_1$. Suppose $C_{\mathcal{M}, t_n \ldots t_1}(p) = s$. By induction, $C_{\mathcal{M}', t_n \ldots t_2}(map_{\downarrow}(p)) = s$. By the definition of $\mathcal{M}'$, $C_{\mathcal{M}, t_n \ldots t_1}(follow(p, b))$ is defined and is equal to s' if and only if $C_{\mathcal{M}', t_n \ldots t_2}(map_{\downarrow}(follow(p, b)))$ is defined and is equal to s', where s' is either s or a state that is directly reachable from s via a step or a skip function. It follows that the stated condition holds in this case. □

Theorem 9.2.2 *Suppose automaton $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ is obtained by* *applying the above construction on normalized automaton*

$$\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$$

with a position assignment pos. Then $\mathcal{M}'$ is a projection of $\mathcal{M}$.

Proof. Suppose $t = t_n \dots t_1$ is a ground n-tuple that is accepted by automaton $\mathcal{M}$ and let $t' = t_n \dots t_2$. Then $C_{\mathcal{M},t}(\varepsilon)$ is defined and is equal to a terminal state s' in $\mathcal{M}$ and $t \models_{\mathcal{M}} \tau(s')$. Since $C_{\mathcal{M},t}(\varepsilon)$ is defined, $C_{\mathcal{M},t}(1)$ is also defined and is equal to a state s that is a predecessor of s'. By Lemma 9.2.1, $C_{\mathcal{M}',t'}(\varepsilon)$ is defined and is equal to state s. By the definition of $\mathcal{M}'$, s is a terminal state and $\tau'(s) = S_{s_1} \cup \dots \cup S_{s_m}$, where $s_1, \dots, s_m$ are all the terminal states in $\mathcal{M}$ that are reachable from s and for all $i \in [1, m]$, $S_{s_i} = \bigcup_{\xi \in \Delta} \{C_\xi \mid C \in \tau(s_i)\}$ in which Δ is the set of all the paths in Δ_s that end at s_i. Clearly, s' is in $\{s_1, \dots, s_m\}$. Suppose ξ is the path from s to s' following the definition of $C_{\mathcal{M},t}$ (ξ is unique with respect to t and $\mathcal{M}$). Suppose further, C is a set in $\tau(s')$ such that $t \models_{\mathcal{M}} C$. By the definition of $\mathcal{M}'$, for all the states s_1 in $\mathcal{M}'$ that are predecessors of s, a color c' in $E^c_{(\xi,C)} \subseteq C$ occurs at state s_1 in $\mathcal{M}$ if and only if color $\upsilon^c_{(\xi,C)}$ in C_ξ occurs at s_1 in $\mathcal{M}'$. Since $t \models_{\mathcal{M}} C$, $t \models_{\mathcal{M}} E^c_{(\xi,C)}$ for all the colors $c \in C$. By the definition of $E^c_{(\xi,C)}$, all the colors in $E^c_{(\xi,C)}$ can be treated as a single color $\upsilon^c_{(\xi,C)}$. Since t' is a prefix of t, we have $t' \models_{\mathcal{M}'} \upsilon^c_{(\xi,C)}$ for all the colors $c \in C$. Clearly, $t' \models_{\mathcal{M}'} C_\xi$. Since C_ξ is a set in $S_{s'}$, we have $t' \models_{\mathcal{M}'} S_{s'}$ and in turn, $t' \models_{\mathcal{M}'} \tau'(s)$. It follows that t' is accepted by $\mathcal{M}'$.

Conversely, suppose $t' = t_n \ldots t_2$ is a ground $(n-1)$-tuple that is accepted by automaton $\mathcal{M}'$. Then $C_{\mathcal{M}',t'}(\varepsilon)$ is defined and is equal to a terminal state s in $\mathcal{M}'$ and $t' \models_{\mathcal{M}'} \tau'(s)$. By the definition of $\mathcal{M}'$, $\tau'(s) = S_{s_1} \cup \ldots \cup S_{s_m}$, where $s_1, \ldots, s_m$ are all the terminal states in $\mathcal{M}$ that are reachable from s and for all $i \in [1,m]$, $S_{s_i} = \bigcup_{\xi \in \Delta} \{C_\xi \mid C \in \tau(s_i)\}$ in which Δ is the set of all the paths in Δ_s that end at s_i. Since $t' \models_{\mathcal{M}'} \tau'(s)$, there exists a terminal state s' of $\mathcal{M}$ such that $s' \in \{s_1, \ldots, s_m\}$ and $t' \models_{\mathcal{M}'} S_{s'}$. Suppose ξ is a path from s to s' such that C_ξ is a set in $S_{s'}$ and $t' \models_{\mathcal{M}'} C_\xi$. Clearly, C is a set in $\tau(s')$. Now we construct a complete ground sub-term t_1 based on ξ and C as follows:

1. If s_1 is a step state on path ξ with $pos(s_1) = 1\ p$ with some position p, then $t_1\ @\ p = b$ provided $\delta(s_1, b)$ is also on the path ξ. Since $\mathcal{M}$ is normalized, $color(s_1) = \emptyset$.

2. If s_1 is a skip state on path ξ with $pos(s_1) = 1\ p$ with some position p, then $t_1 \diamond p$ is:

   (a) an arbitrary ground sub-term if there is no color c such that c occurs at s_1 and c is also in the set C.

   (b) a particular ground sub-term $t_{E^c_{(\xi,C)}}$ if there exists a color c such that

   - c occurs at state s_1 and c is also in the set C,
   - $\upsilon^c_{(\xi,C)}$ does not occurs at any predecessor s_2 of s in $\mathcal{M}'$ such that $C_{\mathcal{M}',t'}(p_2) = s_2$ with some position $p_2 \succ 1$.

   Suppose color c' occurs at a state s_3 on path ξ with $pos(1\ q) = s_3$ and $c' \in E^c_{(\xi,C)}$. Then c' is also in C and $t_1 \diamond p = t_1 \diamond q = t_{E^c_{(\xi,C)}}$.

(c) $t' \diamond p_2$ if there exists a color c such that

- c occurs at state s_1 and c is also in the set C,
- color $\upsilon^c_{(\xi,C)}$ occurs at a predecessor s_2 of s in $\mathcal{M}'$ such that

$$C_{\mathcal{M}',t'}(p_2) = s_2$$

with some position $p_2 \succ 1$.

Since $t' \models_{\mathcal{M}'} C_\xi$, $t' \models_{\mathcal{M}'} \upsilon^c_{(\xi,C)}$. Thus, for all predecessors s_2 of s such that $\upsilon^c_{(\xi,C)}$ occurs at s_2 and $C_{\mathcal{M}',t'}(p_2) = s_2$, $t' \diamond p_2$ is unique. That is, $t_1 \diamond p$ is unique with respect to color c and path ξ and set C. Suppose color c' occurs at a state s_3 on path ξ with $pos(1\,q) = s_3$ and $c' \in E^c_{(\xi,C)}$. Then c' is also in C and $t_1 \diamond p = t_1 \diamond q$.

By the definition of $\mathcal{M}'$, for all the states s_2 in $\mathcal{M}'$ that are predecessors of s, color $\upsilon^c_{(\xi,C)}$ occurs at s_2 in $\mathcal{M}'$ if and only if a color c' in $E^c_{(\xi,C)} \subseteq C$ occurs at state s_2 in $\mathcal{M}$. Since $t' \models_{\mathcal{M}'} C_\xi$, $t' \models_{\mathcal{M}'} \upsilon^c_{(\xi,C)}$ for all the colors $c \in C$. Then by the definition of ground term t_1 and the definition of $E^c_{(\xi,C)}$, we have $t't_1 \models_{\mathcal{M}} E^c_{(\xi,C)}$ for all the colors $c \in C$. Clearly, $t't_1 \models_{\mathcal{M}} C$. Since C is a set in $\tau(s')$, we have $t't_1 \models_{\mathcal{M}} \tau(s')$. It follows that $t't_1 = t_n \dots t_1$ is accepted by $\mathcal{M}$. □

9.3 Canonically Colored Automata

In previous section, we show that there is an equivalence relation $\equiv_\xi$ on colors with respect to some path ξ in an automaton and colors that are equivalent under such relation can be represented by a single color. In this section, we show that we can construct an automaton $\mathcal{M}'$ from an automaton $\mathcal{M}$ such that colors that occur at the states of $\mathcal{M}'$ are pairs of states in $\mathcal{M}$ and the number of colors occurring in $\mathcal{M}'$ is at most $O(n^2)$, where n is the number of states in $\mathcal{M}$ ($\mathcal{M}'$).

Suppose $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is an automaton. We introduce a color $c_{s,s'}$ for each unordered pair of states $\{s, s'\}$ in $\mathcal{M}$ such that s and s' are on the same path and $color(s) \cap color(s') \neq \emptyset$. Color $c_{s,s'}$ is treated the same as $c_{s',s}$. Now we construct an automaton $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ as follows:

1. $s'^0 = s^0$.
2. Q' is the same as Q.
3. For all non-terminal states s in Q', $color'(s) = \{c_{s_1,s_2} \mid s = s_1 \text{ or } s = s_2\}$.
4. s is a step state in $\mathcal{M}'$ if s is a step state in $\mathcal{M}$. In that case, $\delta'(s,b)$ is:
   - not defined if $\delta(s,b)$ is not defined.
   - $\delta(s,b)$ if $\delta(s,b)$ is defined.
5. s is a skip state in $\mathcal{M}'$ if s is a skip state in $\mathcal{M}$. In that case, $\sigma'(s) = \sigma_1(s)$.
6. s is a terminal state in $\mathcal{M}'$ if s is a terminal state in $\mathcal{M}$. Moreover, $\tau'(s)$ is obtained from $\tau(s)$ by replacing each color c appearing in $\tau(s)$ with the set $\mathcal{C}(s,c)$ such that $\mathcal{C}(s,c)$ is

$$\begin{aligned}\{c_{s_1,s_2} \mid \ & s_1 \text{ and } s_2 \text{ are on the same path to } s \\ & \text{and } c \in color(s_1) \cap color(s_2)\}.\end{aligned}$$

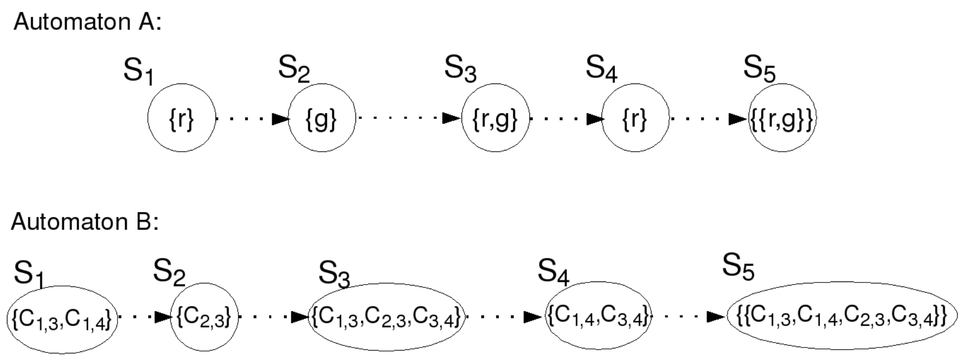

Figure 9.4: Using unordered pairs of states as colors

For example, an automaton A and automaton B obtained (as described above) from A are shown in Figure 9.4. Note that, in color $c_{i,j}$, i stands for state s_i and j stands for s_j.

We call the above construction *a canonically colored* automaton and we claim the following facts related to such construction.

Lemma 9.3.1 *Suppose $\mathcal{M}'$ is obtained by applying the above construction to automaton $\mathcal{M}$. Then the following condition holds for all positions p of a ground n-tuple t:*

- *$C_{\mathcal{M},t}(p)$ is defined and is equal to a state s in $\mathcal{M}$ if and only if $C_{\mathcal{M}',t}(p)$ is defined and is equal to state s in $\mathcal{M}'$. Moreover, $p^s_{\mathcal{M},t} = p^s_{\mathcal{M}',t}$.*

Proof. Directly from the definition of $\mathcal{M}'$. $\square$

Theorem 9.3.2 *Suppose automaton $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ is obtained by applying the above construction on an automaton $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$. Then*

$\mathcal{M}'$ is equivalent to $\mathcal{M}$.

Proof. Suppose ground n-tuple t is accepted by $\mathcal{M}$. Then $C_{\mathcal{M},t}(\varepsilon) = s$ and s is a terminal state in $\mathcal{M}$ and $t \models_{\mathcal{M}} \tau(s)$. By Lemma 9.3.1, $C_{\mathcal{M}',t}(\varepsilon) = s$. By the definition of $\mathcal{M}'$, s is a terminal state in $\mathcal{M}'$ and $\tau'(s)$ is obtained from $\tau(s)$ by replacing each color c appearing in $\tau(s)$ with $\mathcal{C}(s,c)$. Suppose C is a set of $\tau(s)$ such that $t \models_{\mathcal{M}} C$. Then for all colors c in C, $t \models_{\mathcal{M}} c$. Moreover, c occurs at states s_1 and s_2 in $\mathcal{M}$ if and only if $c_{s_1,s_2} \in \mathcal{C}(s,c)$ occurs at s_1 and s_2 in $\mathcal{M}'$. Since $t \models_{\mathcal{M}} c$, $t \models_{\mathcal{M}'} \mathcal{C}(s,c)$ and in turn, $t \models_{\mathcal{M}'} \bigcup_{c\in C} \mathcal{C}(s,c)$. Since $\bigcup_{c\in C} \mathcal{C}(s,c)$ is a set in $\tau'(s)$, we have $t \models_{\mathcal{M}} \tau'(s)$. Thus, t is also accepted by $\mathcal{M}'$.

Conversely, suppose ground n-tuple t is accepted $\mathcal{M}'$. Then $C_{\mathcal{M}',t}(\varepsilon) = s$ and s is a terminal state in $\mathcal{M}'$ and $t \models_{\mathcal{M}'} \tau'(s)$. By Lemma 9.3.1, $C_{\mathcal{M},t}(\varepsilon) = s$. By the definition of $\mathcal{M}'$, s is a terminal state in $\mathcal{M}$ and $\tau'(s)$ is obtained from $\tau(s)$ by replacing each color c appearing in $\tau(s)$ with $\mathcal{C}(s,c)$. Suppose C is a set of $\tau'(s)$ such that $t \models_{\mathcal{M}'} C$ and $\mathcal{C}(s,c)$ is a subset of C that is obtained from color c appearing in $\tau(s)$. Clearly, $t \models_{\mathcal{M}'} \mathcal{C}(s,c)$. Moreover, color c occurs at states s_1 and s_2 in $\mathcal{M}$ if and only if $c_{s_1,s_2} \in \mathcal{C}(s,c)$ occurs at s_1 and s_2 in $\mathcal{M}'$. It follows that $t \models_{\mathcal{M}} c$ and in turn, $t \models_{\mathcal{M}} \bigcup_{\mathcal{C}(s,c)\subseteq C}\{c\}$. Since $\bigcup_{\mathcal{C}(s,c)\subseteq C}\{c\}$ is a set in $\tau(s)$, we have $t \models_{\mathcal{M}} \tau(s)$. Thus, t is also accepted by $\mathcal{M}$. $\square$

Chapter 10

Other Automata Operations

10.1 Other Operations

In section 9.2, we presented a projection operation that projects away the states in an automaton $\mathcal{M}$ that read or skip the symbols of the last term of a ground n-tuple that is accepted by $\mathcal{M}$. It is more desirable to construct an automaton $\mathcal{M}'$ from automaton $\mathcal{M}$ such that a ground $(n-1)$-tuple $t_n \ldots t_{i+1}t_{i-1} \ldots t_1$ $(i > 1)$ is accepted by $\mathcal{M}'$ if and only if ground n-tuple $t_n \ldots t_1$ is accepted by $\mathcal{M}$, i.e., an operation that projects out an "internal" term. To obtain such a construction, we need to build an automaton $\mathcal{M}''$ from $\mathcal{M}$ such that a ground n-tuple

$$t_n \ldots t_{i+1}t_{i-1} \ldots t_1 t_i \ (i > 1)$$

is accepted by $\mathcal{M}''$ if and only if ground n-tuple $t_n \ldots t_1$ is accepted by $\mathcal{M}$. Then $\mathcal{M}'$ is a projection of $\mathcal{M}''$. Clearly, $\mathcal{M}''$ accepts a permutation of ground terms $\{t_1, \ldots, t_n\}$.

Let π be a permutation of $\{1, \ldots, n\}$. We say an automaton $\mathcal{M}'$ is *a permutation of stratified automaton $\mathcal{M}$ with respect to* π if the following condition holds:

- a ground n-tuple $t'_n \ldots t'_1$ is accepted by $\mathcal{M}'$ if and only if ground n-tuple $t_n \ldots t_1$ is accepted by $\mathcal{M}$, where $t'_{\pi(i)} = t_i$ for all $i \in [1, n]$.

Formally, we construct $\mathcal{M}'$ from stratified automaton $\mathcal{M}$ as follows:

1. We build an automaton $\mathcal{M}_1$ for $2n$-tuple $x_n \ldots x_1 y_n \ldots y_1$, as described in section 5.6, such that for all $i \in [1, n]$, x_i and y_i are variables and $x_{\pi(i)} = y_i$.

2. We obtain an expansion $\mathcal{M}_2$ of $\mathcal{M}$ with respect to integer $2n$ as described in

section 9.1.

3. We obtain a conjunction $\mathcal{M}_3$ of $\mathcal{M}_1$ and $\mathcal{M}_2$ as described in section 6.1.

4. We apply projection operation n times on automaton $\mathcal{M}_3$ to obtain an automaton $\mathcal{M}'$ as described in section 9.2.

Theorem 10.1.1 *Suppose $\mathcal{M}'$ is obtained by applying the above construction on stratified automaton $\mathcal{M}$. Then $\mathcal{M}'$ is a permutation of stratified automaton $\mathcal{M}$ with respect to π.*

Proof. By Theorem 5.6.3, automaton $\mathcal{M}_1$ accepts all the ground $2n$-tuples $t'_n \dots t'_1 t_n \dots t_1$ such that $t'_{\pi(i)} = t_i$ for all $i \in [1,n]$. By Theorem 9.1.3, $\mathcal{M}_2$ accepts all the ground $2n$-tuples $t'_n \dots t'_1 t_n \dots t_1$ such that $t_n \dots t_1$ is accepted by $\mathcal{M}$. By Theorem 6.1.4, $\mathcal{M}_3$ accepts all the ground $2n$-tuples $t'_n \dots t'_1 t_n \dots t_1$ such that $t'_{\pi(i)} = t_i$ for all $i \in [1,n]$ and $t_n \dots t_1$ is accepted by $\mathcal{M}$. Finally, by Theorem 9.2.2, $\mathcal{M}'$ accepts all the ground n-tuples $t'_n \dots t'_1$ such that $t'_{\pi(i)} = t_i$ for all $i \in [1,n]$ and $t_n \dots t_1$ is accepted by $\mathcal{M}$. □

We also need an operation to determine whether S_1 is a subset of S_2, where S_1 and S_2 are sets of ground n-tuples that are accepted by automata $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively. We say an automaton $\mathcal{M}'$ *is subsumed by* automaton $\mathcal{M}$ if every ground n-tuple t accepted by $\mathcal{M}'$ is also accepted by $\mathcal{M}$.

Formally, suppose $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is a stratified and normalized automaton with a position assignments *pos*. We construct

$$\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$$

as follows:

1. $s'^0 = \{s^0\}$.

2. Q' is the least subset of 2^Q that contains s'^0 and is closed under skip and step functions, as described in the sequel.

3. For all non-terminal states S in Q', $color'(S) = color(s)$, where $s \in S$ and $pos(s)$ is the least position in $\{pos(s') \mid s' \in S\}$. We denote the state s by $least(S)$. Such state always exists since $\succ$ is a total ordering and the set S is finite.

4. S is a step state in $\mathcal{M}'$ if $s = least(S)$ is a step state in $\mathcal{M}$. In that case, $\delta'(S,b)$ is:
   - not defined if $\delta(s,b)$ is not defined.
   - $S \cup \{s'\}$ if $\delta(s,b)$ is defined and $s' = \delta(s,b)$.

5. S is a skip state in $\mathcal{M}'$ if $s = least(S)$ is a skip state in $\mathcal{M}$. In that case, $\sigma'(S) = S \cup \{s'\}$.

6. S is a terminal state in $\mathcal{M}'$ if $s = least(S)$ is a terminal state in $\mathcal{M}$. In that case, $\tau'(S) = \tau(s)$.

Clearly, $\mathcal{M}'$ is equivalent to $\mathcal{M}$. Each state S in $\mathcal{M}'$ is formed by all the states on an unique path from s^0 to the state $s = least(S)$.

We say the above construction $\mathcal{M}'$ is *an unwinding of* $\mathcal{M}$. Suppose S is a set of states that is a state in automaton $\mathcal{M}'$ and C is a set of colors. We define an equivalence relation $R^1_{(S,C)}$ on positions with respect to S and C and an equivalence relation $R^2_{(S,C)}$ on colors with respect to S and C such that both equivalence relations simultaneously satisfy the following conditions:

- If (c_1, c_2) is in $R^2_{(S,C)}$ and s_1 and s_2 are states in S such that $c_1 \in color(s_1)$ and $c_2 \in color(s_2)$, then

$$(pos(s_1), pos(s_2)) \in R^1_{(S,C)}.$$

- If both c_1 and c_2 are in C and are in the set $color(s)$ with some state $s \in S$, then $(c_1, c_2) \in R^2_{(S,C)}$.

Note that, since $\mathcal{M}$ is normalized, a color c only occurs at skip states of $\mathcal{M}$. Thus, for all pairs (p,q) in $R^1_{(S,C)}$, p and q are positions for skip states.

We define a equivalence relation R^t on positions with respect to a ground n-tuple t to be the set $\{(p,q) \mid t \diamond p = t \diamond q\}$. We claim the following facts.

Lemma 10.1.2 *Suppose $\mathcal{M}'$ is an unwinding of stratified and normalized automaton $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$. Then a ground n-tuple t is accepted by $\mathcal{M}'$ if and only if $C_{\mathcal{M}',t}(\varepsilon)$ is a terminal state S in $\mathcal{M}'$ and there exists at least one set $C \in \tau(least(S))$ such that $R^1_{(S,C)} \subseteq R^t$.*

Proof. Suppose ground n-tuple t is accepted by $\mathcal{M}'$. Then $C_{\mathcal{M}',t}(\varepsilon)$ is a terminal state S in which $s = least(S)$ is a terminal state in $\mathcal{M}$ and $t \models_{\mathcal{M}} \tau(s)$. Thus, there exists at lease one set C in $\tau(s)$ such that $t \models_{\mathcal{M}} C$. Suppose s_1 and s_2 are two arbitrary states such that $(pos(s_1), pos(s_2)) \in R^1_{(S,C)}$. By the definition of $\mathcal{M}'$, s_1

and s_2 are skip states and both $C_{\mathcal{M}',t}(pos(s_1))$ and $C_{\mathcal{M}',t}(pos(s_2))$ are defined. Thus, $p_1 = pos(s_1)$ and $p_2 = pos(s_2)$ are positions of ground n-tuple t. Since $(p_1, p_2) \in R^1_{(S,C)}$, there are two cases:

- There exists a color $c \in C$ such that $c \in color(s_1) \cap color(s_2)$. Since $c \in C$, $t \models_{\mathcal{M}} c$. Clearly, in this case, $t \diamond p_1 = t \diamond p_2$.
- There exist colors c_1 and c_2 such that:

  1. both c_1 and c_2 are in $R^2_{(S,C)}$ with $c_1 \in color(s_1)$ and $c_2 \in color(s_2)$,
  2. and both c_1 and c_2 are in the set $color(s')$ with some state $s' \in S$.

  Since c_1 and c_2 are in C, $t \models_{\mathcal{M}} c_1$ and $t \models_{\mathcal{M}} c_2$. Since c_1 occurs at s_1 and s', we have $t \diamond p_1 = t \diamond pos(s')$. Since c_2 occurs at s_2 and s', we have $t \diamond p_2 = t \diamond pos(s')$. It follows that $t \diamond p_1 = t \diamond p_2$.

In summary, (p_1, p_2) is also in R^t. It follows that $R^1_{(S,C)} \subseteq R^t$.

Conversely, suppose $C_{\mathcal{M}',t}(\varepsilon)$ is a terminal state S in $\mathcal{M}'$ and there exists at least one set $C \in \tau(least(S))$ such that $R^1_{(S,C)} \subseteq R^t$. Suppose c is an arbitrary color in C. For all the states s_1 and s_2 in S with $c \in color(s_1) \cap color(s_2)$, we have $(pos(s_1), pos(s_2)) \in R^1_{(S,C)}$. By the assumption, $(pos(s_1), pos(s_2)) \in R^t$. Thus, $t \diamond pos(s_1) = t \diamond pos(s_2)$. Clearly, we have $t \models_{\mathcal{M}} c$. It follows that $t \models_{\mathcal{M}} C$. Since C is a set in $\tau(s)$, $t \models_{\mathcal{M}} C$. Thus, t is accepted by $\mathcal{M}$ and in turn, t is accepted by $\mathcal{M}'$. $\square$

Theorem 10.1.3 *Suppose $\mathcal{M}_1$ and $\mathcal{M}_2$ are unwindings of two stratified and normalized automata. Suppose further, $\mathcal{M}$ is an automaton that is built as a construction (a disjunction) described in section 6.2 on $\mathcal{M}_1$ and $\mathcal{M}_2$. Then $\mathcal{M}_1$ is subsumed by $\mathcal{M}_2$ if and only if for all the terminal states S in $\mathcal{M}$, one of the following conditions holds:*

1. *$s = \{S_2\}$ such that S_2 is a terminal state in $\mathcal{M}_2$.*
2. *$s = \{S_1, S_2\}$ such that:*
   - *S_1 and S_2 are terminal states in $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively,*
   - *and for all the sets C_1 in the acceptance condition of S_1, there exists a set C_2 in the acceptance condition of S_2 such that*

$$R^1_{(S_2,C_2)} \subseteq R^1_{(S_1,C_1)}.$$

Proof. Suppose $\mathcal{M}_1$ is subsumed by $\mathcal{M}_2$. Suppose further, t is a ground n-tuple that is accepted by $\mathcal{M}$. Thus, $C_{\mathcal{M},t}(\varepsilon)$ is a terminal state s in $\mathcal{M}$. We consider the following cases:

- s has the form $\{S_1\}$ such that S_1 is a terminal state in $\mathcal{M}_1$. Then t is accepted by $\mathcal{M}_1$ but t is not accepted by $\mathcal{M}_2$. This is impossible since $\mathcal{M}_1$ is subsumed by $\mathcal{M}_2$.
- s has the form $\{S_2\}$ such that S_2 is a terminal state in $\mathcal{M}_2$. Then t is accepted by $\mathcal{M}_2$ but t is not accepted by $\mathcal{M}_1$. Clearly, condition (1) holds and condition (2) holds vacuously in this case.

- s has the form $\{S_1, S_2\}$ such that S_1 and S_2 are terminal states in $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively. Clearly, condition (1) holds vacuously in this case. Now consider condition (2). Suppose for the purpose of obtaining a contradiction, we assume that there exists a set C_1 in the acceptance condition of S_1 such that there does not exist a set C_2 in the acceptance condition of S_2 that possesses the following property:

  $$R^1_{(S_2,C_2)} \subseteq R^1_{(S_1,C_1)}.$$

  Suppose t' is a ground n-tuple such that $C_{\mathcal{M}_1,t'}(\varepsilon) = S_1$ and $R^{t'} = R^1_{(S_1,C_1)}$. As we mentioned before, for all pairs (p, q) in $R^1_{(S_1,C_1)}$, p and q are positions for skip states. Thus, such that ground n-tuple t' does exist. By Lemma 10.1.2, t' is accepted by $\mathcal{M}_1$. Since $\mathcal{M}_1$ is subsumed by $\mathcal{M}_2$, t' is also accepted by $\mathcal{M}_2$. However, for all the sets C_2 in the acceptance condition of S_2, $R^1_{(S_2,C_2)} \not\subseteq R^1_{(S_1,C_1)} = R^{t'}$. Thus, By Lemma 10.1.2, t' is not accepted by $\mathcal{M}_2$, which is a contradiction. It follows that condition (2) also holds in this case.

Conversely, suppose for all the terminal states s in $\mathcal{M}$, one of the stated conditions holds. Suppose further, t is a ground n-tuple that is accepted by $\mathcal{M}$. Then $C_{\mathcal{M},t}(\varepsilon)$ is a terminal state s. By the assumption, there are two cases:

1. $s = \{S_2\}$ such that S_2 is a terminal state in $\mathcal{M}_2$. In this case, t is only accepted by $\mathcal{M}_2$.

2. $s = \{S_1, S_2\}$ in which S_1 and S_2 are terminal states in $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively, and for all the sets C_1 in the acceptance condition of S_1, there exists

a set C_2 in the acceptance condition of S_2 such that

$$R^1_{(S_2,C_2)} \subseteq R^1_{(S_1,C_1)}.$$

Suppose t is accepted by $\mathcal{M}_1$. Then by Lemma 10.1.2, there exists at least one set C_1 in the acceptance condition of S_1 such that $R^1_{(S_1,C_1)} \subseteq R^t$. By the assumption, there exists a set C_2 in the acceptance condition of S_2 such that

$$R^1_{(S_2,C_2)} \subseteq R^1_{(S_1,C_1)}.$$

Clearly, $R^1_{(S_2,C_2)} \subseteq R^t$. Again, by Lemma 10.1.2, t is also accepted by $\mathcal{M}_2$.

In summary, if t is accepted by $\mathcal{M}_1$ then t is also accepted by $\mathcal{M}_2$. Thus, $\mathcal{M}_1$ is subsumed by $\mathcal{M}_2$. □

Suppose $\mathcal{M}_1$ and $\mathcal{M}_2$ are unwindings of two stratified and normalized automata, and $\mathcal{M} = (Q, s^0, \delta, \sigma, color, \tau)$ is an automaton that is constructed as described in section 6.2. Suppose further,

$$\mathcal{M}_1 = (Q_1, s_1^0, \delta_1, \sigma_1, color_1, \tau_1)$$

and

$$\mathcal{M}_2 = (Q_2, s_2^0, \delta_2, \sigma_2, color_2, \tau_2).$$

We define an automaton $\mathcal{M}' = (Q', s'^0, \delta', \sigma', color', \tau')$ as follows:

1. $s'^0 = s^0$.

2. Q' is the least subset of Q that contains s'^0 and is closed under skip and step functions, as described in the sequel.

3. For all non-terminal states s in Q', $color'(s) = color(s)$.

4. s is a step state in $\mathcal{M}'$ if s is a step state in $\mathcal{M}$. In that case, $\delta'(s,b)$ is:

   - not defined if $\delta(s,b)$ is not defined.
   - $\delta(s,b)$ if $\delta(s,b)$ is defined.

5. s is a skip state in $\mathcal{M}'$ if s is a skip state in $\mathcal{M}$. In that case, $\sigma'(s) = \sigma(s)$.

6. s is a terminal state in $\mathcal{M}'$ if s is a terminal state in $\mathcal{M}$. Moreover, we have the following cases:

   - If s has the form $\{S_1\}$ such that S_1 is a terminal state in $\mathcal{M}_1$, then $\tau'(s) = \tau_1(S_1)$.
   - If s has the form $\{S_2\}$ such that S_2 is a terminal state in $\mathcal{M}_2$, then $\tau'(s) = \{\}$.
   - If s has the form $\{S_1, S_2\}$ such that S_1 and S_2 are terminal states of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively, then $\tau'(s)$ is obtained from $\tau_1(S_1)$ by eliminating all the sets $C_1 \in \tau_1(S_1)$ such that there exists a set $C_2 \in \tau_2(S_2)$ with

$$R^1_{(S_2,C_2)} \subseteq R^1_{(S_1,C_1)}.$$

Clearly, $\mathcal{M}'$ accepts a subset of ground n-tuples that are accepted by $\mathcal{M}_1$, and by Theorem 10.1.3, $\mathcal{M}'$ is not subsumed by $\mathcal{M}_2$. We call the construction $\mathcal{M}'$ a *rough-difference of automata $\mathcal{M}_1$ and $\mathcal{M}_2$* since there may exist a ground n-tuples t such that t is accepted by $\mathcal{M}_1$ and t is also accepted by $\mathcal{M}_2$.

10.2 Implementation of Top-Down Algorithm

In this section, we show the implementation of the operations used in the TOPDOWN algorithm with automata operations.

We say an automaton $\mathcal{M}$ *represents a set S of ground n-tuples* if and only if $\mathcal{M}$ accepts S and nothing more than S. Now the following operations used in TOPDOWN algorithm can be directly implemented with automata operations as follows:

1. $instance(\bar{t})$ is represented by an automaton that is for n-tuple $\bar{t}$.

2. Suppose automata $\mathcal{M}_1$ and $\mathcal{M}_2$ represents sets of ground n-tuples S_1 and S_2 respectively. Then $union(S_1, S_2)$ is represented by a disjunction of $\mathcal{M}_1$ and $\mathcal{M}_2$. Moreover, $diff(S_1, S_2)$ is represented by a rough-difference of $\mathcal{M}_1$ and $\mathcal{M}_2$. Note that, TOPDOWN algorithm does not require an exact difference operation. An exact difference operation is just an optimization.

Suppose S is a set of ground n-tuples that is an instance of an atom A and t is a ground n-tuple in S. Then we denote the sub-term of t that is corresponding to a variable x appearing in A by $t[x]$. Looking closely to the remaining operations *join_with_head_atom*, *join_with_body_atom* and *map* used in TOPDOWN algorithm, we find that two major tasks are accomplished in those operations:

1. Given a m-tuple $\overline{X}$ of distinct variables and an atom A and a set S of ground n-tuples that is an instance of A, S is transformed to a set S' of ground m-tuples that is an instance of $\overline{X}$ such that:
   - S' is the set of all ground m-tuples t' for which there exists a ground n-tuple t in S with $t'[x] = t[x]$ for all variables x that appear in both $\overline{X}$ and A.
2. Given a m-tuple $\overline{X}$ of distinct variables and an atom A and a set S of ground m-tuples that is an instance of $\overline{X}$, S is transformed to a set S' of ground n-tuples that is an instance of A such that:
   - S' is the set of all ground n-tuples t' for which there exists a ground m-tuple t in S with $t'[x] = t[x]$ for all variables x that appear in both $\overline{X}$ and A.

Now we devise two automata operations to accomplish the above tasks:

1. FROMATOM$(\mathcal{M}, A, \overline{X})$ (for task (1)): Function FROMATOM takes as arguments:
   - a atom A and a m-tuple $\overline{X}$ of distinct variables,
   - and an automaton $\mathcal{M}$ that represents a set S of ground n-tuples that is an instance of A,

   and returns an automaton $\mathcal{M}'$ that represents the set S' that is obtained from S and is an instance of $\overline{X}$. Formally, FROMATOM$(\mathcal{M}, A, \overline{X})$ is defined as follows:

- Let $\mathcal{M}_1$ be an ungrouping of $\mathcal{M}$ with respect to the linear version of A. Let $\overline{X'}$ be the k-tuple of all the variable occurrences appearing in A. Then $\mathcal{M}_1$ represents the set S_1 that is obtained from S and is an instance of $\overline{X'}$.
- Let $\mathcal{M}_2$ be an expansion of $\mathcal{M}_1$ with respect to $m+k$.
- Let $\mathcal{M}_3$ be an automaton for $(m+k)$-tuple $\overline{X}\overline{X'}$.
- Let $\mathcal{M}_4$ be a conjunction of $\mathcal{M}_2$ and $\mathcal{M}_3$.
- We apply projection operation k times on automaton $\mathcal{M}_4$ to obtain an automaton $\mathcal{M}'$.
- Return $\mathcal{M}'$.

2. TOATOM$(\mathcal{M},\overline{X},A)$ (for task (2)): Function TOATOM takes as arguments:

   - a atom A and a m-tuple $\overline{X}$ of distinct variables,
   - and an automaton $\mathcal{M}$ that represents a set S that is an instance of $\overline{X}$,

   and returns an automaton $\mathcal{M}'$ that represents the set S' that is obtained from S and is an instance of A. Formally, TOATOM$(\mathcal{M},\overline{X},A)$ is defined as follows:

   - Let $\overline{X'}$ be the k-tuple of all the variable occurrences appearing in A.
   - Let $\mathcal{M}_1$ be an expansion of $\mathcal{M}$ with respect to $k+m$.
   - Let $\mathcal{M}_2$ be an automaton for $(k+m)$-tuple $\overline{X'}\overline{X}$.
   - Let $\mathcal{M}_3$ be a conjunction of $\mathcal{M}_1$ and $\mathcal{M}_2$.
   - We apply projection operation m times on automaton $\mathcal{M}_3$ to obtain an automaton $\mathcal{M}_4$. Then $\mathcal{M}_4$ represents a set that is an instance of $\overline{X'}$.
   - Let $\mathcal{M}'$ be a grouping of $\mathcal{M}_4$ with respect to the linear version of A.

- Return $\mathcal{M}'$.

Now the remaining operations used in TOPDOWN algorithm can be implemented with automata operations as follows:

1. *join_with_head_atom*$(J, A', \overline{X})$: Suppose automaton $\mathcal{M}$ represents a set J such that J is an instance of atom A and A has the same predicate as A'. Then

   $$join\_with\_head\_atom(J, A', \overline{X})$$

   is represented by FROMATOM$(\mathcal{M}', A', \overline{X})$, where $\mathcal{M}'$ is a conjunction of $\mathcal{M}$ and $\mathcal{M}''$ in which $\mathcal{M}''$ is an automaton for atom A'.

2. *join_with_body_atom*$(\overline{X}, S, B, S')$: Suppose set S is an instance of $\overline{X}$ and set S' is an instance of atom B. Suppose further, automaton $\mathcal{M}$ and $\mathcal{M}'$ represent sets S and S' respectively. Then

   $$join\_with\_body\_atom(\overline{X}, S, B, S')$$

   is represented by a conjunction of $\mathcal{M}$ and FROMATOM$(\mathcal{M}', B, \overline{X})$.

3. *map*$(\overline{X}, S, B)$: Suppose automaton $\mathcal{M}$ represents a set S that is an instance of $\overline{X}$. Then *map*$(\overline{X}, S, B)$ is represented by TOATOM$(\mathcal{M}, \overline{X}, B)$.

Chapter 11

MTBDD Implementation of Automata

11.1 Overview of MTBDD Representation

In this chapter, we propose a variant of Multi-Terminal Decision Diagrams (MTBDDs) [21] to represent automata. Since acceptance conditions are propositional formulas and color sets can be treated as conjunctions, standard BDD techniques can be used to represent acceptance conditions and color sets. A MTBDD is a function $f : \{0,1\}^n \rightarrow D$, where D is an arbitrary range. In order to represent automata as MTBDDs, we first encode each symbol in the set Σ by K bits given the assumption that Σ is finite and has 2^K distinct symbols. Thus, a MTBDD representation for a $step/skip-$automaton is a function $f : \{0,1\}^K \rightarrow D$, where D is the set of all the BDD representations of acceptance conditions.

11.2 BDD Representation of Acceptance Conditions

We impose a total ordering $\succ$ on all the colors in the set $\mathcal{C}$ and assign each color a boolean variable. Then an acceptance condition is naturally represented by a reduced and ordered BDD. In such BDD representation, each non-terminal node corresponds to a color c, and if we take the “high” (or “then”) edge of that node, then we assume that c is satisfied, and if we take the “low” (or “else”) edge, then we assume that c is not satisfied. When evaluating a BDD representation of an acceptance condition S, if we reach terminal node **1** then S is satisfied, otherwise S is not satisfied. For example, a BDD representation of acceptance condition $\{\{r,b\},\{g,d\}\}$ (with $r \succ b \succ g \succ d$)is shown in Figure 11.1.

We also represent a set of colors with a BDD by treating a set of colors as a

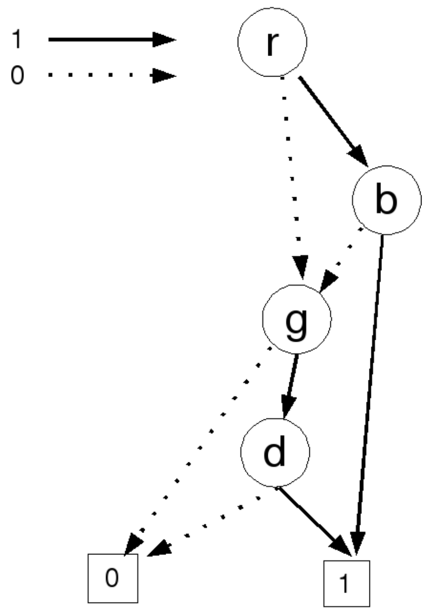

Figure 11.1: BDD representation of acceptance conditions

conjunction of atomic propositions. The boolean function operations such as $\wedge$ and $\vee$ are all the operations needed to manipulate coloring sets and acceptance conditions. Those operations are implemented by using BDD conjunction and disjunction operations respectively [11, 12].

11.3 MTBDD representation of Automata

Suppose each symbol in the set Σ is encoded by K bits. Now we give the definition of MTBDD representation of automata. A Multi-Terminal Binary Decision

Diagram MD for automaton $\mathcal{M}$ is defined to be a tuple

$$(N, T, S, F, r, high, low, color, value)$$

such that:

1. r is the *root node* that corresponds to the initial state of $\mathcal{M}$.
2. N is a finite set of *step nodes* that correspond to the step states of $\mathcal{M}$.
3. T is a finite set of *auxiliary nodes*. In the sequel, we will show that auxiliary nodes are only related to step nodes.
4. S is a finite set of *skip nodes* that correspond to the skip states of $\mathcal{M}$.
5. F is a finite set of *terminal nodes* that correspond to the terminal states of $\mathcal{M}$. We say a node u is a non-terminal node if $u \in N \cup T \cup S$.
6. N, T, S and F are pairwise disjoint sets. This is directly from the fact that the sets of step states, skip states and terminal states of $\mathcal{M}$ are pairwise disjoint.
7. *color* is a labeling function that maps a non-terminal node to a BDD representation of a set of colors.
8. *value* is a labeling function that maps a terminal node to a BDD representation of an acceptance condition.

9. *high* is a function that maps a non-terminal node u to its *high child*, and *low* is a function that maps a non-terminal node u to its *low child*. We also call the directed edge from u to $high(u)$ a "high" (or "then") edge, and the directed edge from u to $low(u)$ a "low" (or "else") edge. We say u is *the parent node* of $high(u)$ and $low(u)$.

In normal "reduced" BDDs, a non-terminal node with identical high and low children is skipped and reduced to either one of its children. In the MTBDD representation of automata, this kind of reduction does not happen (this is called "quasi-reduced"). Formally, a MTBDD representation of an automaton also satisfies the following conditions:

1. A skip node must have identical high and low children.

2. The high and low children of a step node must be auxiliary nodes.

3. Suppose n is an auxiliary node. Then $color(n)$ must be a BDD that represents empty set of colors.

4. Suppose $n_1 \dots n_m$ is a sequence of nodes such that for all $i \in [1, m-1]$, there is a "high" edge or "low" edge from n_i to n_{i+1}. Suppose further, n_1 is a step node and n_m is a step or skip node, and for all $i \in [1, m-1]$, n_i is an auxiliary node. Then m must be $K+1$. In this case, the K directed edges from n_1 to n_m is a binary encoding of some symbol in Σ ("high" edge for bit 1 and "low" edge for bit 0).

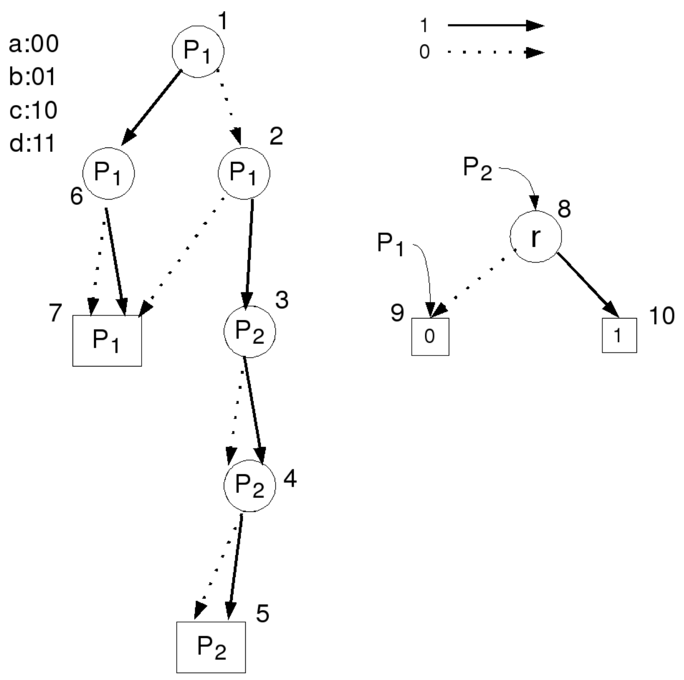

Figure 11.2: A MTBDD that represents n-tuple *bXX*

5. If there exist two non-terminal nodes u and v such that u and v are in the same set (N, T, or S) and $color(u) = color(v)$ and $high(u) = high(v)$ and $low(u) = low(v)$, then u must be identical to v.

6. If there exist two terminal nodes u and v such that $value(u) = value(v)$, then u must be identical to v.

We say a MTBDD that satisfies the above conditions is a *quasi-reduced ordered* MTBDD (QRO-MTBDD), which is a variant of ordinary MTBDD. We show an example of QRO-MTBDD that represents an n-tuple *bXX* in Figure 11.2.

In this example, we suppose $\Sigma = \{a, b, c, d\}$ such that a,b,c and d are all constants. The encoding of the constants is also shown in Figure 11.2. In that figure, we use circles to represent non-terminal nodes and use squares to represent terminal nodes. The numbers outside of the nodes are the indices of those nodes. For all the nodes n of MTBDDs, n contains a pointer that points to a BDD that represents either a set of colors or an acceptance condition. For all the non-terminal nodes n of BDDs, n contains a boolean variable that stands for a color, and there are only two terminal nodes 1 and 0 in a BDD. In Figure 11.2, root node (1) is a step node and it contains a pointer that points to terminal node (9) that represents an empty set of colors. Node (2) is an auxiliary node. Nodes (3) and (4) are skip nodes and both of them contain a pointer that points to a BDD that represents coloring set $\{r\}$. Node (5) is a terminal node and it contains a pointer that points to a BDD that represents acceptance condition $\{\{r\}\}$. Node (6) is an auxiliary node and both children of node (6) lead to a terminal node (7) that contains a pointer that points to a BDD that represents acceptance condition $\emptyset$.

We devise various MTBDD operations to implement automata operations by directly following those constructions defined in the previous chapter.

Chapter 12

SML/NJ Implementation of the Framework

I have coded a prototype implementation of the framework proposed in this dissertation under the Standard ML of New Jersey (SML/NJ) language system. The following is a brief description of all the modules in the SML/NJ implementation:

1. Module for Horn clauses: This module contains a parser for Horn clauses, which is built upon ML-Lex and ML-Yacc programs. It provides utility functions to manipulate Horn clause programs, such as divide a program into a set of rules and a set of facts. It also provides the following interfaces:
    (a) Interface to access the statistical information of a Horn clause program, such that the total number of function symbols and constants. This information is used to determine how many bits are needed to encode all the function symbols and constants.
    (b) Interface to map a function symbol or a constant to its binary encoding, or verse versa. The former is useful when we create an automaton for an n-tuple (see Section 5.6).
    (c) Interface to map a function symbol to its arity. This interface is used to add $\#_f$ new skip states with respect to some function symbol f while we perform a branching operation on a skip state (see Section 7.2).
2. Basic BDD module: This module implements all the logical operations of BDDs and operation caches. In this module, maximally structure-shared storage of BDDs is achieved by using node tables, and operation caches are used to avoid repeating the calculation of an operation on particular

arguments after it has been done once. In order to avoid swamping the system through overly aggressive memory allocation, the implementation applies a heuristic that automatically monitors garbage-collection activity to determine when to adjust the maximal sizes of the node table and operation caches.

3. Coloring module: This module is built upon the basic BDD module to implement BDD representations for color sets and acceptance conditions. It implements the union operation for color sets. It also implements And, Or and other operations for acceptance conditions.

4. Basic MTBDD module: This module is built upon the basic BDD module to implement MTBDD representation for automata. It implements conjunction, disjunction, grouping, ungrouping, expansion, projection and rough-difference operations for automata.

5. Module for ground tuples: This module is built upon the basic MTBDD module to implement MTBDD representation for sets of ground tuples. It implements union, intersection operations for sets of ground tuples. It also implements major operations in top-down set-at-a-time algorithm such as *join_with_head_atom*, *join_with_body_atom* and *map* (see Section 10.2).

6. Horn clauses evaluator module: This module implements the top-down depth-first set-at-a-time evaluation algorithm for Horn clauses.

Chapter 13

Conclusions

13.1 Contributions

Static program analysis tools can help developers build reliable software with fewer errors and security vulnerabilities. In order to make it easier to design and develop program analysis tools, we propose a framework in this dissertation such that program analysis algorithms are expressed as Horn clauses and the evaluation of Horn clauses is implemented in MTBDDs.

Horn clauses are very useful in expressing analysis algorithms that involve complex objects. Chapter 3 shows that a typical type-based analysis can be naturally expressed by Horn clauses in which types are represented as functions.

Although, Horn clauses are simple and easy to use, the existing implementations of Horn clauses do not show impressive performances on analyzing large programs. The framework proposed in this dissertation leads to an efficient implementation of Horn clauses that can scale to large programs. First of all, the framework employs a top-down depth-first set-at-a-time evaluation strategies. Such strategy is good for answering queries, efficient to find partial answers, and can manipulate symbolic representations of terms. In this dissertation, the top-down depth-first set-at-a-time evaluation algorithm proposed by Bugaj and Nyugen is interpreted in terms of sets of ground tuples. This interpretation leads to a BDD-based implementation of top-down evaluation. Second of all, the framework represents sets of ground tuples as automata. Various operations on automata are devised to implement the top-down set-at-a-time evaluation algorithm. More importantly, such automata lend themselves readily to a symbolic representation using

MTBDDs. Finally, the framework is equipped with a MTBDD implementation of automata and a prototype implementation of the framework is coded under the Standard ML of New Jersey (SML/NJ).

13.2 Future Work

There are many improvements that could be made to the framework to produce better performance. One improvement could be made by reducing the cost of normalization operation with an idea that eliminates all the colors on the step states in one scan instead of repeatedly applying c-normalization procedures. Another improvement could be to implement the rough-difference operation without building "unwinding" constructions. Projection operation can also be optimized by using the idea of canonically colored automata to handle equivalent colors.

In developing the ideas presented in this dissertation, we have primarily focused on automata whose underlying directed graphs are acyclic. However, it now seems to us that allowing $step/skip$-automata to have cycles could make them more expressive in terms of the predicates they can represent. Such automata are more general and may be useful in the applications of theorem proving. In order to implement operations like conjunction and disjunction, though, it will be necessary to use "position counters" (relative positions used in conjunction and disjunction operations) explicitly. That is, it is possible that we do not reach the undecidability by introducing the infinity in a controlled way as an explicit "position counter" feature.

In this dissertation, we build a program analysis tool that uses Horn clauses as the specification language and implements Horn clauses with MTBDDs. It would be interesting to compare the following techniques for program analysis with our tool, which might suggest new directions for using logic programs in program analysis:

1. Visibly pushdown languages: They are a subclass of context-free languages that were introduced by Alur and Madhusudan [6]. Visibly pushdown languages are accepted by visibly pushdown automata whose stack behavior is determined by the input symbol. If the input symbol is a *call action* then the automaton must push, if it is a *return action* then the automaton must pop, otherwise it (is a *internal action*) cannot change the depth of the stack. It has been shown that the class of visibly pushdown languages is closed under intersection, union, complementation, renaming, concatenation and Kleene star [6]. Some undecidable problems for context-free languages, such as universality, language equivalence and language inclusion, become EXPTIME-complete for visibly pushdown languages. Visibly pushdown languages are suitable for formal program analysis. One example would be using visibly pushdown languages to specify and verify the correctness requirements of structured programs [5].

2. Satisfiability modulo theories (SMT) solver: SMT problem is to determine whether a logic formula, which is expressed in classical first-order logic

with the combinations of background theories, is satisfiable (has a solution). Under Satisfiability modulo theories (SMT), the interpretation of function symbols and predicates in first-order logic formulas is constrained by a background theory. For example, predicates for inequalities (e.g., $3x + 2y - z \geq 4$) are evaluated by using the rules of the theory of linear real arithmetic, and the theory of arithmetic restricts the interpretation of function symbols such as $+, \leq, 0$, and 1. The procedures for SMT problems are called SMT solvers. The origin of SMT solvers can be traced back to early 1980s. Since then several SMT solvers have been developed in academia and industry, which made an enormous progress in the scale of problems that can be solved. SMT solvers usually use a so called *lazy approach*, which tightly integrates boolean SAT solvers and theory-specific solvers (T-solvers) such that SAT solvers and T-solvers repeatedly communicate via simple APIs. The advantage of this approach is that SAT solvers take care of Boolean information and Theory solvers take care of theory information, that is, everyone does what it is good at. It is worth to mention that there is an efficient SMT solver developed at Microsoft, which is called $Z3$, that is used in several program analysis projects at Microsoft.

3. Top-down breadth-first evaluation: This evaluation strategy builds a searching tree starting the root node (the goal G) and searches all the children (produced by all the rules that define the predicate of G). Then for each child node, it searches all the children and so on, until it finds the facts.

To remove the potential bottleneck of tuple-at-a-time strategy in XSB, researchers convert the depth-first subgoal scheduling scheme in XSB to a breadth-first one in order to obtain a set-at-a-time search engine [20]. Experimental results in breadth-first XSB have shown excellent performance for queries involving disk-resident data [20].

Bibliography

[1] National institute of standards and technology, department of commerce. software errors cost u.s. economy $59.5 billion annually. *NIST News Release*, 2002-10.

[2] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

[3] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 233–242, New York, NY, USA, 2005. ACM.

[4] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free java. In *VMCAI*, pages 149–160, 2004.

[5] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Tools and Algorithms for the Construction and Analysis of*

Systems Lecture Notes in Computer Science, volume 2988, pages 467–481. Springer, 2004.

[6] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211, New York, NY, USA, 2004. ACM.

[7] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *J. Log. Program.*, 4(3):259–262, 1987.

[8] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1986. ACM Press.

[9] A. Banerjee and D. Naumann. Secure information flow and pointer confinement in a java-like language, 2002.

[10] P. N. Benton. Strictness properties of lazy algebraic datatypes. In *WSA '93: Proceedings of the Third International Workshop on Static Analysis*, pages 206–217, London, UK, 1993. Springer-Verlag.

[11] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[12] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[13] G. L. Burn. A logical framework for program analysis. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 30–42, London, UK, 1993. Springer-Verlag.

[14] G. L. Burn, C. L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In *on Programs as data objects*, pages 42–62, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[15] K. Clark. Predicate logic as a computational formalism. Technical Report DOC 79/59, Dept. of Computing, Imperial College, 1979.

[16] M. Coppo, F. Damiani, and P. Giannini. Strictness, totality, and non-standard-type inference. *Theor. Comput. Sci.*, 272(1-2):69–112, 2002.

[17] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems — a case study. In *ACM Symposium on Programming Language Design and Implementation*, pages 117–126, 1996.

[18] C. Flanagan and S. N. Freund. Type-based race detection for java. *SIGPLAN Not.*, 35(5):219–232, 2000.

[19] C. Flanagan and S. Qadeer. A type and effect system for atomicity. *SIGPLAN Not.*, 38(5):338–349, 2003.

[20] J. Freire, T. Swift, and D. S. Warren. Taking i/o seriously: Resolution reconsidered for disk. In *International Conference on Logic Programming*, pages 198–212, 1997.

[21] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient datastructure for matrix representation. *Form. Methods Syst. Des.*, 10(2-3):149–169, 1997.

[22] H. Gallaire and J. Minker, editors. *Logic and Data Bases*. Perseus Publishing, 1978.

[23] C. K. Gomard and N. D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[24] A. Gräf. Left-to-right tree pattern matching. In *RTA*, pages 323–334, 1991.

[25] E. Hajiyev, M. Verbaere, O. de Moor, and K. de Volder. Codequest: querying source code with datalog. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 102–103, New York, NY, USA, 2005. ACM.

[26] C. Hankin and D. L. Metayer. A type-based framework for program analysis. In *Static Analysis Symposium*, pages 380–394, 1994.

[27] R. Heldal and J. Hughes. Binding-time analysis for polymorphic types. In *PSI '02: Revised Papers from the 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 191–204, London, UK, 2001. Springer-Verlag.

[28] F. Henglein. Efficient type inference for higher-order binding-time analysis. In *Proceedings of the 5th ACM conference on Functional programming*

languages and computer architecture, pages 448–472, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[29] T. Jensen. Types in program analysis. In *The essence of computation: complexity, analysis, transformation*, pages 204–222, New York, NY, USA, 2002. Springer-Verlag New York, Inc.

[30] T.-M. Kuo and P. Mishra. Strictness analysis: a new perspective based on type inference. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 260–272, New York, NY, USA, 1989. ACM Press.

[31] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2005. ACM Press.

[32] J. W. Lloyd. *Foundations of logic programming; (2nd extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[33] E. Madalińska-Bugaj and L. A. Nguyen. Generalizing the QSQR evaluation method for Horn knowledge bases. In N. Nguyen and R. Katarzyniak, editors, *New Challenges in Applied Intelligence Technologies*, volume 134 of *Studies in Computational Intelligence*, pages 145–154. Springer, 2008.

[34] E. Madalińska-Bugaj and L. A. Nguyen. Generalizing the QSQR evaluation method for Horn knowledge bases. (Revised and extended version: `http://www.mimuw.edu.pl/~nguyen/GQSQR-revised-long.pdf`), 2010.

[35] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[36] A. Mycroft. The theory and practise of transforming call-by-need into call-by-value. In *International symposium on programming*, 1980.

[37] N. Nedjah. Minimal deterministic left-to-right pattern-matching automata. *SIGPLAN Not.*, 33(1):40–47, 1998.

[38] N. Nedjah and L. de Macedo Mourelle. Minimal adaptive pattern-matching automata for efficient term rewriting. In *CIAA '01: Revised Papers from the 6th International Conference on Implementation and Application of Automata*, pages 221–233, London, UK, 2002. Springer-Verlag.

[39] N. Nedjah, C. D. Walter, and S. E. Eldridge. Optimal left-to-right pattern-matching automata. In *ALP '97-HOA '97: Proceedings of the 6th International Joint Conference on Algebraic and Logic Programming*, pages 273–286, London, UK, 1997. Springer-Verlag.

[40] F. Nielson. The typed lambda-calculus with first-class processes. In *PARLE '89: Proceedings of the Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, pages 357–373, London, UK, 1989. Springer-Verlag.

[41] H. R. Nielson and F. Nielson. Automatic binding time analysis for a typed lambda-calculus. *Science of Computer Programming*, 10:139–176, 1988.

[42] J. Palsberg. Type-based analysis and applications. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 20–27, New York, NY, USA, 2001. ACM Press.

[43] R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[44] T. Reps. Demand interprocedural program analysis using logic databases. in applications of logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196. Kluwer Academic Publishers, Boston, MA, 1994.

[45] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *PPOPP*, pages 83–94, 2005.

[46] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *SIAM J. Comput.*, 24(6):1207–1234, 1995.

[47] K. Solberg. Annotated type systems for program analysis, 1995.

[48] K. L. Solberg. Strictness and totality analysis. In *SAS*, pages 408–422, 1994.

[49] K. L. Solberg. Strictness and totality analysis with conjunction. In *TAPSOFT*, pages 501–515, 1995.

[50] S. Sudarshan and R. Ramakrishnan. Optimizations of bottom-up evaluation with non-ground terms: extended abstract. In *ILPS '93: Proceedings of the 1993 international symposium on Logic programming*, pages 557–574, Cambridge, MA, USA, 1993. MIT Press.

[51] J. Ullman. *Principles of Databases and Knowledge-Base Systems*, volume 2 edition. Computer Science Press, Rockville, MD, 1989.

[52] J. D. Ullman. Bottom-up beats top-down for datalog. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 140–149, New York, NY, USA, 1989. ACM Press.

[53] L. Vieille. Recursive axioms in deductive databases: The query-subquery approach. In L. Kerschberg, editor, *Proc. from the First International Conference on Expert Database Systems*, pages 253–267. Addison-Wesley, Redwood City, CA, 1987.

[54] L. Vieille. Recursive query processing: the power of logic. In *Theory of Computer Science*, volume 69:1, pages 1–53, Essex, UK, 1989. Elsevier Science Publishers Ltd.

[55] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[56] D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference*

CAAP/FASE on Theory and Practice of Software Development, pages 607–621, London, UK, 1997. Springer-Verlag.

[57] D. S. Warren. Memoing for logic programs. *Commun. ACM*, 35(3):93–111, 1992.

[58] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using datalog and binary decision diagrams for program analysis. In K. Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, volume 3780 of *LNCS*. Springer-Verlag, Nov. 2005.

[59] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.

[60] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

[61] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced database systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.