Logic Programming—a Functional Approach

Borislav Agapiev
Dipl. Ing. Electrotechnics, University of Belgrade, 1984

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

April 1992

The dissertation "Logic Programming—a Functional Approach" by Borislav Agapiev has been examined and approved by the following Examination Committee:

Richard Kieburtz
Professor
Thesis Research Adviser

David Maier
Professor

James Hook
Assistant Professor

John Conery
Associate Professor

Dedication

To my mother.

Acknowledgements

This dissertation would not have been possible without the help and support of many people at the Oregon Graduate Institute. First I would like to thank my adviser, Dick Kieburtz, for all kinds of support I received during the course of my graduate research. The numerous discussions we had on many topics helped me tremendously in gaining better understanding of many aspects of my research. I thank him for all the time he spent and the patience he showed.

I would also like to thank my friend and former officemate, Bart Schaefer, for his patience in always having time to listen to my not always coherent ideas. Discussions with him and my other friends and colleagues at OGI have been very important to me. I believe they are a part of what makes OGI such a wonderful research environment.

Last, but most certainly not least, I would like to thank my best friend Ljupčo Čitkušev for being such a great friend and for the help he generously extended to me so many times. Our numerous discussions have been a part of the drive to understand the world around us, the drive which never failed to motivate me and always provided me with the strength to go on further. So many times we ventured together to the wonderful road of the search for meaning, the road that never ends but is so astonishingly beautiful. I cannot wait to go there again!

Contents

Abstract

Logic Programming—a Functional Approach

Borislav Agapiev, Ph.D.
Oregon Graduate Institute of Science & Technology, 1992

Supervising Professor: Richard Kieburtz

This dissertation presents a new way to evaluate logic programs by translation to an equivalent equational presentation. A logic program is translated to a set of equations that can be viewed as a functional program. The equations have solutions that coincide with answer substitutions that satisfy the corresponding logic program. An essential feature is that the bindings for variables are defined by sets of mutually recursive data definitions. The corresponding solutions are not *least*, in the sense of domain theory, but they can be found in a straightforward way. An immediate consequence of our strategy is that unification is replaced by pattern matching. This translation does not lead to a functional interpreter of logic programming but to a functional program equivalent to the translated logic program. The approach is inspired by attribute grammars and their implementation in lazy functional languages.

Chapter 1

Introduction

Since the earliest days of computing one of the fundamental problems has been how to make the process of creating computer programs more effective. This problem motivated research in the area of programming languages. The first means of programming were entering the machine instructions directly into the memory. It was realized very quickly that this method was very unsatisfactory way to program a computer; it was very slow and exceedingly tedious for a programmer. What was needed was a more abstract way to express programming concepts. The first answer was provided by development of assemblers; they enabled programmers to enter machine instructions in symbolic form rather than using direct machine addresses. The next important milestone was development of FORTRAN which can be viewed as the first of the new class of programming languages—high level languages. FORTRAN was a huge step forward; it revolutionized programming by freeing programmers from a maze of inessential details they had to fight previously. It provided a whole new level of abstraction in programming. Soon, there were many successors to FORTRAN—e.g. Algol60, COBOL, PL/1, and more recently, PASCAL, C, Ada, Modula-2 etc. Most of these languages—including FORTRAN are in widespread use today.

However, beginning in early seventies, it started to became increasingly clear that there were some problems with these languages. The rate of progress in the improvements of capabilities of hardware increased tremendously, but this rate was not matched by improvements in the process of producing software. As the requirements for complexity of software systems started to rapidly increase, the need for fast and reliable ways of

delivering software became more crucial. One of the answers to this problem was offered in the form of *declarative* programming.

It should be mentioned that the programming language LISP was designed at about the same time as FORTRAN. LISP can be viewed as the precursor of declarative programming languages—it was originally developed as a symbol processing language with a simple set of powerful primitives. It adopted use of λ notation for functions and strong emphasis was placed on composition of functions as means of building more complex functions from simple pieces. However, in the subsequent development of the language, more emphasis was placed on features related to imperative languages and less on declarative aspects of the language.

1.1 Declarative versus imperative programming

The main idea in declarative programming is that programs should not express *how* a result is to be computed but instead *what* is to be computed. The focus is on the properties of the result, i.e., on the question how to characterize the result of a program. This point of view is taken in mathematics. This is in sharp contrast to the usual approach taken in imperative programming, where a program can be viewed simply as a sequence of steps to be executed on a computer. Indeed, this was exactly what the very first programs in machine code were—exact descriptions of machine instructions to be executed to get a solution. Modern imperative languages still follow this approach in part—the difference is that the description of an imaginary machine on which a program in particular language operates is more abstract. For instance, machine locations are replaced by variables, machine storage configurations are described by data structures, sequences of instructions are replaced by programming constructs such as loops, etc. The emphasis is on the notions of *store* as aggregate of machine locations and *control* as means of expressing ways to sequence instructions. Imperative programs contain many details that are not connected with the result but are needed to ensure that the execution of the program is carried out correctly. As the size of a program increases, these details

quickly become overwhelming and seriously impair the ability of programmers to grasp behavior of programs.

Declarative programming offers a solution to this problem by including in programs only the information that is essential to specifying results.

1.2 The thesis question

The question this dissertation tries to answer is whether it is possible to translate logic programs directly to functional programs. The trivial answer to this question is affirmative as demonstrated by proposals for implementations of logic languages in functional languages [Fel85, Car84]. However these kinds of implementations are really not what we are after; in these approaches, as well as in the proposals for integrating functional and logic languages, an additional level of interpretation is introduced in functional solutions in order to accommodate the effects of logical variables in logic languages. Roughly speaking, one can say that a logic program is translated to an interpreter for logic programs implemented in a functional language. The consequence of this kind of approach is a completely different notion of variables in functional and logic languages. By translation of logic programs to functional programs we mean translating a logic program to a functional program without any additional interpretation. In particular we do not want to introduce interpretation of *variables*.

Another way to state the question is whether it is possible to achieve effects of logical variables in functional programs.

The main contribution of this research is to show that the answer to the question is affirmative. This objective is accomplished by exhibiting a translation with desired properties. We believe that our approach presents a new way of looking at logic languages—a point of view inspired by functional languages. The main contributions can be summarized as follows:

- We present an algorithm for translation of logic programs to sets of equations that can be viewed as functional programs; the equations produced by the translation have solutions that correspond to solutions of logic programs.
- We define a semantics for the functional programs produced by the translation; this approach defines a new semantics for logic languages.
- We show how to implement the functional language that is the target of the translation.

The rest of the dissertation is organized as follows:

- The rest of the Chapter 1 consists of an introduction to functional and logic programming and a comprehensive example.
- Chapter 2 describes the translation algorithm.
- Chapter 3 presents a semantics for programs produced by the translation.
- Chapter 4 presents an operational semantics based on traversals of cyclic graphs.
- Chapter 5 shows a code generation algorithm.
- Chapter 6 presents some new ideas on translation of logic languages to functional languages.
- Finally, Chapter 7 contains conclusion and directions for future research.

1.3 Functional and logic programming

We focus our attention on two prominent classes of declarative programming languages:

- Functional programming languages
- Logic programming languages

These two classes are based on completely different principles; they represent distinct approaches to achieving the goal of declarative programming. However, we will see in the subsequent chapters that there are also some similarities between the two approaches.

1.3.1 Logic programming

This section gives a brief introduction to logic programming. Our presentation follows closely Apt [Apt90]. The account is not intended to be comprehensive; for additional treatment on this subject the reader is referred to literature [MW88, Apt90, Llo87].

Logic programming, as its name suggests, is closely related to Mathematical Logic. Logic programs can be viewed simply as sets of logical formulae of a first order language. In order to define the formulae we first introduce several definitions.

Definition 1 (Alphabet) An *alphabet* for a first order language consists of:

- a denumerable set of *variables*
- *constructor symbols*; with each constructor symbol we associate a nonnegative integer called its *arity*
- *predicate symbols*; with each predicate symbol we associate a nonnegative integer, which is also called its arity
- the *propositional constants*, **true** and **false**
- the *connectives*:
  - *negation* $\neg$
  - *disjunction* $\vee$
  - *conjunction* $\wedge$
  - *implication* $\Rightarrow$
- *quantifiers* $\exists$ (there exists) and $\forall$ (for all)

We need to define the set of well formed formulae of the language. The formulae involve *terms*, which denote elements of the domain of discourse.

Definition 2 (Term) The set of terms is defined inductively as the least set satisfying:

- a variable is a term
- if c is a constructor symbol with arity n and $t_1, \ldots, t_n$ are terms, then $c(t_1, \ldots, t_n)$ is a term.

Constants are viewed as constructors with arity 0.

Definition 3 (Formula) The set of well formed formulae is defined as follows:

- if p is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula. These formulae are called *atoms*
- if F is a formula, so is $\neg F$
- if F and G are formulae, so are $F \vee G$, $F \wedge G$
- if x is a variable and F is a formula, $\exists x F$ and $\forall x F$ are formulae

To avoid extraneous parentheses, a precedence is imposed on connectives and quantifiers. Here is the list of binding powers, in decreasing order:

- $\neg$, $\exists$, $\forall$
- $\vee$
- $\wedge$

We assume $\vee$ and $\wedge$ associate to the right. For the sake of brevity, if F is a formula with $x_1, \ldots, x_k$ as free variables, then $\forall F$ and $\exists F$ stand for $\forall x_1 \ldots \forall x_k F$ and $\exists x_1 \ldots \exists x_k F$, respectively.

An atom will be also called a *positive literal* while a negated atom will be called *negative literal.* Atomic formulae can be understood as *propositions.* Intuitively, we can think of them as expressing either truth or falsity of some assertions. This association is reflected in the *interpretation* which assigns either truth or falsity to propositions.

Logic programs

Logic programs can be viewed as sets of formulae in a distinguished form, called *clauses.*

Definition 4 (Clauses) Let $L_1, \ldots, L_n$ be literals. A formula of the form $\forall L_1 \vee \ldots \vee L_n$ is called a *clause.* Clauses will be written in a special form. Let $M_1, \ldots, M_k$ be a list of the positive literals in the clause above and $N_1, \ldots, N_l$ be a list of the negative literals (without their negation symbols). Then the clause will be written as $M_1, \ldots, M_k \Leftarrow N_1, \ldots, N_l$. Also we will use the symbol $:-$ instead of $\Leftarrow$.

The above two ways of expressing clauses are equivalent as can be immediately seen from the tautology $p \Leftarrow q \Leftrightarrow p \vee \neg q$. Intuitively, we can think of a clause $M_1, \ldots, M_k \Leftarrow N_1, \ldots, N_l$ asserting M_1 or ...or M_k if N_1 and ...and N_l. We will focus our attention on special form of clauses, called *Horn clauses.*

Definition 5 (Horn Clauses) A clause with at most one positive literal will be called a *Horn clause.* We distinguish several kinds of Horn clauses:

- a clause with exactly one positive literal will be called a *definite clause*
- among definite clauses, clauses with no negative literals will be called *unit clauses* or *facts*
- a Horn clause with no positive literals will be called *goal clause*

The empty clause, denoted $\square$, plays an important role; it can be viewed as *contradiction.*

Unification

We need to introduce the concept of *substitution.*

Definition 6 (Substitution) A *substitution* is a finite mapping from variables to terms. Substitutions will be written as $\sigma = \{t_1/x_1, \ldots, t_n/x_n\}$. It is assumed that all x_i are distinct and $\forall i\ x_i \neq t_i$.

The substitutions can be applied to various expressions in the language. Let t be a term and σ be a substitution $\sigma = \{t_1/x_1, \ldots, t_n/x_n\}$. Then the *image* of t under σ, written as σt is defined as follows:

- if t is a variable y, then if $y = x_i$, then $\sigma t = t_i$, else y is different from all variables in σ and $\sigma t = y$
- t has the form $c(t_1, \ldots, t_m)$, then $\sigma t = c(\sigma t_1, \ldots, \sigma t_n)$

Applications of substitutions can be generalized to literals, clauses and sets of clauses. Substitutions can be composed; given substitutions $\sigma = \{t_1/x_1, \ldots, t_n/x_n\}$ and $\theta = \{r_1/y_1, \ldots, r_m/y_m\}$ the composition $\sigma\theta$ is defined as

$$\sigma\theta = \{\theta t_1/x_1, \ldots, \theta t_n/x_n, r_1/y_1, \ldots, r_m/y_m\}$$

where all pairs $x_j/\theta t_j$ such that $x_j = \theta t_j$ are removed as well as all pairs y_j/r_j such that $\exists i\ y_j = x_i$. In words, substitutions σ and θ are composed simply by applying θ to all terms t_i in σ, adding mappings for all variables in θ and making sure that the result is a valid substitution. It is easily seen that, given a term t, $\sigma\theta(t) = \theta(\sigma(t))$. We say σ is more *general* than θ if there exists a substitution η such that $\theta = \sigma\eta$. We are now ready to define the concept of *unification.*

Definition 7 (Most General Unifier) Given two terms t_1 and t_2, their *most general unifier* is a substitution σ such that $\sigma t_1 = \sigma t_2$. Furthermore, σ is the most general such substitution, i.e., for any other η such that $\eta t_1 = \eta t_2$, σ is more general than η.

The concept of unification was introduced by Robinson [Rob65]. It is the fundamental concept in the theory and practice of logic programming.

Resolution

Let $G = :-L_1, \ldots, L_n$ be a goal clause in which all literals are ground, i.e. contain no variables. Let $M = B :- D_1, \ldots, D_m$ be a ground clause such that $\exists i,\ 1 \leq i \leq n,\ B = L_i$. Then a goal clause

$$:-L_1, \ldots, L_{i-1}, D_1, \ldots, D_m, L_{i+1}, \ldots, L_n$$

obtained from G by replacing B by $D_1, \ldots, D_m$ is called the *resolvent* of G and M. B is called the *selected atom* and M is called the *input clause*. The computation step described above is called a *resolution step*. The process of computation in logic programming consists of repeated application of the resolution step until either the empty clause is derived or it is not possible to perform the resolution step any more (because there are no atoms to be selected). The sequence of resolution steps is called a *derivation*. If a derivation ends with the empty clause, it is called a *refutation*.

A refutation can be understood as a derivation of a contradiction from a logic program represented by a set of clauses and a goal. A goal $:-L_1, \ldots, L_n$ stands for $\neg L_1 \vee \ldots \vee \neg L_n$ (recall that goal is a clause with no positive literals). This form is equivalent to $\neg(L_1 \wedge \ldots \wedge L_n)$. The logic program together with the goal imply a contradiction, so the refutation can be understood as a proof of $L_1 \wedge \ldots \wedge L_n$.

The entire process of repeating resolution steps is called *resolution* theorem proving. We are going to focus our attention on particular kind of resolution, called *SLD-resolution*.

In the presence of variables, the resolution step described above can be generalized to handle clauses with variables. Assume there is a goal $G =: -L_1, \ldots, L_n$ and a clause $C = B :- D_1, \ldots, D_m$ such that there is L_i which can be unified with B, with the the unifier σ, i.e., $\sigma L_i = \sigma B$. Then the resolvent is

$$:-\sigma(L_1, \ldots, L_{i-1}, D_1, \ldots, D_m, L_{i+1}, \ldots, L_n)$$

It is obtained by substituting the body of C for L_i in G and applying σ to the rest of the goal clause. We have to be careful to avoid name clashes so C cannot contain any variables in common with G. If necessary, variables in C have to be renamed.

Given a logic program P with a goal G let $G_0, G_1, \ldots$ be a sequence of goals where $G = G_0$ and let $C_0, C_1, \ldots$ be a sequence of instances of clauses from P. We call this sequence an *SLD-derivation* if the following conditions are satisfied:

- $\forall i$ G_{i+1} is the resolvent of G_i and C_i
- $\forall i$ C_i has no variables in common with $G, C_0, C_1, \ldots, C_{i-1}$

An *SLD-derivation* in which the sequence of goals ends with the empty clause is called an *SLD-refutation.* An SLD-derivation that cannot be completed to an SLD-refutation because there are no choices for the next resolvent after the last resolution step is *failed.* The substitution $\gamma = \sigma_n \ldots \sigma_1$ obtained by composing all substitutions produced at each resolution step in an SLD-refutation is called an *answer substitution.* Finally, we need to introduce the concept of a *proof tree.* A proof tree is simply a tree that reflects the structure of an SLD-refutation since it is comprised of clauses in the refutation. The abbreviation SLD stands for Selection rule-driven Linear resolution of Definite clauses. Without going into too much detail, the *selection rule* is a rule which specifies the selected atom at the each resolution step.

Definition 8 Let t be a tree with nodes labeled by clauses and leaves labeled by facts and let $S = C_0, C_1, \ldots$ be the sequence of nodes obtained by the leftmost pre-order traversal of t (i.e. the interior nodes are traversed before their children and the children are traversed from left to right). We call t a *derivation tree* if S is the sequence of input clauses in an SLD-derivation. We call t a *proof tree* if S is the sequence of input clauses in an SLD-refutation.

Example 1 Consider the logic program

$p(X, Z) :- q(X, Y), r(Y, Z).$
$q(X, X).$
$r(X, X).$

and the goal

$:- p(X, a).$

The following sequence of goals is an SLD-refutation for the program above (the input clause used at each resolution step is shown next to a goal)

G_0 $:-p(X,a).$ C_1 $p(X_1,Z_1):-q(X_1,Y_1),r(Y_1,Z_1).$

G_1 $:-q(X_1,Y_1),r(Y_1,a).$ C_2 $q(X_2,X_2).$

G_2 $:-r(Y_1,a).$ C_3 $r(X_3,X_3).$

G_3 $\square$

The corresponding proof tree is shown in the Figure 1.1:

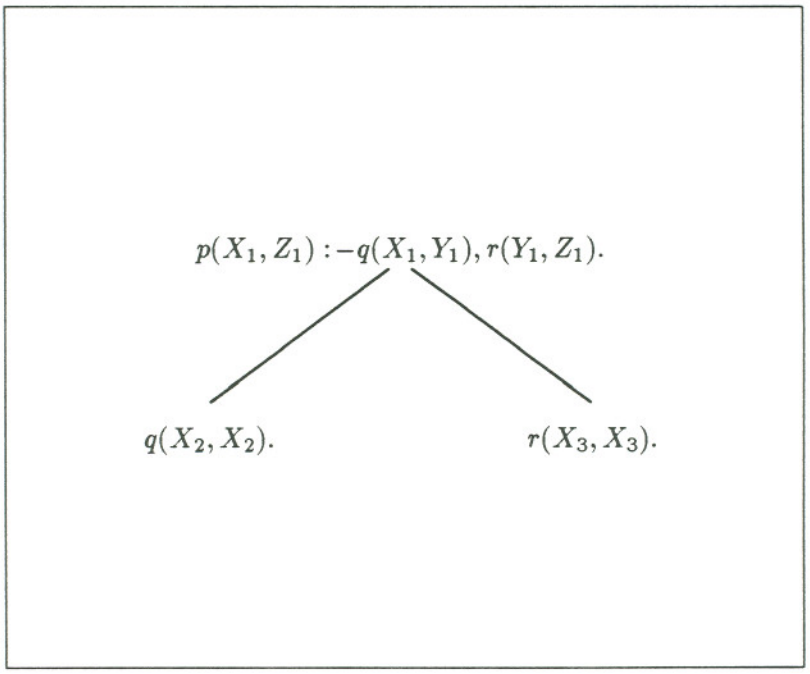

Figure 1.1: An example of a proof tree

Induction principle

We will be using extensively the principle of *structural induction* for proof trees. This principle is an instance of the general principle of *well-ordered induction.* For more information on well-orderings the reader is referred to any of standard textbooks on set theory, e.g. [Hal68].

Definition 9 (Initial Segment) Given a partially ordered set S and an element $x \in S$, the *initial segment* of x, designated $s(x)$ is the set of all elements smaller than x, i.e.,

$$s(x) = \{y \mid y \in S, y < x\}$$

Definition 10 (Well-ordering) A partially ordered set is *well-ordered* if every non-empty subset of S has a least element.

Definition 11 (Well-ordering Induction) Assume P is an inductive property of a well-ordered set S. If for an arbitrary element $x \in S$ it is always the case that $P(x)$ holds whenever $P(y)$ holds for every $y \in s(x)$ then P holds for all elements in S.

It is clear that proof trees are inductively defined, with base cases represented by facts and inductive steps by clauses. To apply the principle of well-ordered induction, we need to observe that given a proof tree t corresponding to a clause, the initial segment $s(t)$ is just the set of all proof trees corresponding to the literals in the body of the clause.

1.3.2 Functional programming

As its name suggests, functional programming is about computations with *functions*. The foundation of functional programming is provided by the λ-calculus, which is a formalism invented by Alonzo Church [Chu41].

Consider the expression $x + 5$ where x is a variable. This expression has different values depending on the value of the variable x. From this point of view we can think of it as a *function* which maps the value of the variable to the value of the whole expression. In mathematical notation, this function can be expressed as the mapping $x \mapsto x + 5$. In λ-calculus this function is expressed as

$$\lambda x.x + 5$$

So a λ-term $\lambda x.M$, where M is a expression in which x might occur, represents a function which maps a value to the result of substituting that value in place of x in M (this informal definition is slightly imprecise but it will suffice for our purposes here). A λ-term

of the form $\lambda x.M$ is called an *abstraction.* Abstractions can be applied to expressions; applying an abstraction to an expression consists simply of performing the substitution of the argument expression for all occurrences of the parameter variable in the body of the abstraction. We make these concepts more precise.

Definition 12 (λ-terms) The set of λ-terms is defined as follows:

- A variable is a λ-term. Variables will be denoted by lowercase letters such as $x, y, z \ldots$
- If x is a variable and M is a λ-term, then $\lambda x.M$ is a λ-term.
- If M and N are λ-terms then $M\ N$ is a λ-term.

We will use uppercase letters such as $M, N, P, Q, R \ldots$ for λ-terms.

Definition 13 (Free and bound variables) The set of *free variables* of a λ-term M, written $FV(M)$, is defined as follows:

- $FV(x) = \{x\}$
- $FV(\lambda x.M) = FV(M) - \{x\}$
- $FV(M\ N) = FV(M) \cup FV(N)$

A variable is *bound* in λ-term M if it has no free occurrences in M.

We will use the notation $M[N/x]$ for the λ-term obtained by substituting N for all free occurrences of x in M. We have to be careful to avoid any free variables of N becoming bound. If necessary, bound variables must be renamed. We implicitly assume $\lambda x.M$ and $\lambda y.(M[y/x])$ to be equivalent.

The process of applying an abstraction to an argument which we informally described is expressed in λ-calculus by a conversion axiom.

Definition 14 (β-conversion) The axiom of β-conversion is defined by:

$$(\lambda x.M)\ N = M[N/x]$$

This axiom is the principal axiom for conversion of terms in the λ-calculus. There is another axiom, of η-conversion:

$$\lambda x.M\ x = M$$

assuming x is not free in M. This axiom will be of less importance to us. A λ-term in the form $(\lambda x.M)\ N$ or $(\lambda x.M)\ x$ (x not free in M) is called a *redex*. We say M *reduces in one step* to N if M has a subterm which is a redex and N is obtained by performing the conversion of the redex. The reflexive and transitive closure of this relation is called *reduction*, written $M \rightarrow N$.

A λ-term is said to be in *normal form* if it contains no redexes. Not every λ-term has a normal form. A term can have several redexes; the question is which one we choose for reduction? Will the result depend on which redex is chosen? The answer to the latter question is provided by one of the fundamental theorems of the λ-calculus:

Theorem 1 (Church-Rosser) Assume there are λ-terms M, N_1, N_2 such that $M \rightarrow N_1$ and $M \rightarrow N_2$. Then there is a term P such that $N_1 \rightarrow P$ and $N_2 \rightarrow P$.

The direct corollary of the theorem is that if a λ-term has a normal form, then the normal form is unique.

The concept of reduction is the fundamental concept in functional programming. Functional programs can be viewed simply as unreduced λ-terms and the process of computation consists of reducing them to normal forms, which can be understood as the final results of programs. The emphasis is on *values*, i.e., the normal forms of expressions.

Pattern matching

Most modern functional languages (e.g. Hope [BMS80], LML [Joh87b, Aug87], SML [MTH90], CAML [Wei90], Haskell [HJW91], Miranda [Tur85] etc.) use pattern matching as a control construct. Patterns are built from values of a distinguished collection of types, which have several names in the literature—"structured types" or "algebraic types" or "free data types". These notions refer to the same concept. As an example,

consider a type of lists:

$$List\ \alpha = Nil + Cons(\alpha \times List\ \alpha)$$

This is an example of a recursive type definition; α is a *type variable* meaning it can stand for any type. In words, the above definition can be described as:

- Nil is a list of type $List\ \alpha$
- if t is a list of type $List\ \alpha$ and h is of type α, then $Cons(h, t)$ is a list of type $List\ \alpha$

This definition is an instance of a more general scheme—structured types. In general, the definition is in the form of *sum of products*. It can be represented as

$$T = c_1(T_{11} \times \ldots \times T_{1n_1}) + \ldots + c_m(T_{m1} \times \ldots \times T_{mn_m})$$

where c_i are called *constructors*. Each constructor has *arity* that specifies the number of the components in the argument tuple. The arity can be zero, in which case constructors are constants (such as Nil in the definition of $List$). T_{ij} are types, some of which can be T since the definition may be recursive. *Patterns* are simply terms with variables. Many languages put restriction on patterns by requiring them to be *linear*—meaning there are no multiple occurrences of variables.

An important example of use of patterns is in defining functions. A function F of n arguments can be defined by a set of equations with patterns:

$$\begin{array}{ll} & F\ p_{11}\ \ldots\ p_{1n} = e_1 \\ \| & F\ p_{21}\ \ldots\ p_{2n} = e_2 \\ & \quad\vdots \\ \| & F\ p_{m1}\ \ldots\ p_{mn} = e_m \end{array}$$

The function is defined by several equations. Given a sequence $v_1, \ldots v_n$ of n arguments, each equation is matched against the arguments by attempting to match every element of the argument sequence with the corresponding pattern. By matching a pattern p against

an argument v, we mean finding a substitution σ such that $\sigma p = v$, i.e., a substitution for variables in the pattern which makes the pattern equal to the value. If all matches in an argument sequence succeed for a particular equation, then the resulting substitution, which gives bindings to all variables in the patterns, is applied to the right hand side of the equation, and the result is returned as the result of the application.

It might be possible that several equations match. In that case a criterion is established as the part of the definition of the language that specifies which equations will be used. Usually, for simplicity of implementation, the equations are tried in order in which they are listed. But there are more elegant solutions, e.g., imposing restrictions on equations so at most one equation can match. It might be the case that none of the equation matches. We say that equations are *exhaustive* if there is at least one pattern matching each value of the argument type.

The choice of considering a single equation has to be made since we are considering only deterministic languages. In a non-deterministic language, the criterion of determining which equation to use is not present since semantics specifies that an arbitrary equation can be used.

The equations defined by patterns can contain *guards*. A guard is simply a condition that has to be satisfied in order for the equation in a definition to match. Guards can be in various forms; a rather general one is adopted in LML where a guard can be any boolean-valued expression. The expression is evaluated and if the result is *true*, the guard is satisfied. Guards can be used to simulate nonlinear patterns, for instance:

$$F\ x\ x \equiv F\ x\ y\ \&\ (x = y)$$

The expression after & is a guard that specifies that bindings for variables x and y should be equal.

An essential point about equations in functional programs is that they are *directed*. The usual convention is that the variables appear on the left hand side of equations; the variable then stands for the value of the term on the right hand side. For instance, the

informal meaning of a definition such as

$$\textbf{let } x = M \textbf{ in } N$$

is that all occurrences of the variable x in N are replaced by M. Other variables can appear in M; their bindings have to be provided by other definitions. It is also possible to have ground terms in the left hand side of equations, as we have seen for pattern matching. For instance, a definition such as

$$\textbf{let } 5 = x \textbf{ in } N$$

means that the value of the variable x should be evaluated and matched against the constant 5. Note that this definition does NOT mean that 5 should be substituted for all occurrences of x as we would expect in case the equation in the definition was undirected. There are also equations in which identifiers occur in both left and right hand side of an equation; in this case the values of variables are defined recursively. For instance, a definition

$$\textbf{rec } ones = 1.ones$$

defines infinite lists of ones. Directedness of functional equations is a fundamental property of functional programs. It ensures that the process of reduction is clearly defined since at each step we know the direction in which substitution for variables is performed.

Functional languages can be divided into two classes:

- *strict* functional languages
- *lazy* functional languages

The difference between the two classes is in the order of evaluation. In particular, the important issue is the order of evaluation of arguments of applications. In strict languages the arguments are evaluated before a function is invoked while in lazy languages the evaluation of arguments is delayed until the arguments are actually needed. In general, it is possible to talk about strictness or laziness of various construct in a language, e.g. constructors or pattern matching.

Example 2 Consider the expression

$$\textbf{rec}\ \mathit{ones} = 1.\mathit{ones}$$

We have seen that this expression defines an infinite list of ones. Such a definition does not make sense in a strict language since in strict languages all values are fully evaluated and an attempt to fully evaluate the expression above would result in nontermination. In contrast, in lazy languages evaluation is performed only when needed so the expression above makes perfect sense because we might be interested only in a finite part of it in which case the computation terminates.

1.4 An example

Consider an example which will help us understand better the issues involved in the relationship between functional and logic programming. This example is based on a similar example [Red86] where it was called the *address translation* problem.

Suppose we have a list of elements such that each one is either in the form DEF(a) or USE(a). These forms can be thought of as abstractions of definitions and uses of various symbols in programs, e.g., symbols for procedures, variables etc.. The goal is to translate this list to a new list such that each occurrence of USE(a) is replaced by REF(n) where n is a unique number; this number is assigned to the corresponding DEF(a) entry. We will assume that the numbers are assigned to DEF entries in the order in which the entries are listed. The numbers can be thought of as abstractions of symbol table references. The translation is to be accomplished in a single pass by a sequential evaluator. For example, the list

[DEF(a), USE(a), USE(b), DEF(b)]

should be translated to

[REF(1), REF(2)]

Numbers 1 and 2 are assigned to the occurrences of DEF(a) and DEF(b), respectively.

The problem is simple if all occurrences of USE(a) in the input list are preceded by the corresponding occurrences of DEF(a). A symbol table is introduced to record associations of symbols and numbers assigned to them. Every time an occurrence of USE(a) is encountered, the symbol table is consulted for the number assigned to the symbol a and the resulting REF(n) is put in the output list. This kind of solution can be implemented in a straightforward way either in an imperative language, logic language or functional language. But things get more complicated by having occurrences of USE(a)

precede the corresponding occurrence of DEF(a). This situation is supposed to model cases where symbols are used before they are declared. Now a functional solution does not appear to be obvious any more. The problem is in the symbol table. In order to translate the occurrences of USE(a) correctly, the corresponding entry in the symbol table has to be already present when an occurrence of USE(a) is encountered, but this implies that the translation cannot be done before the entire symbol table is constructed. It takes a complete pass just to gather this information; the translation takes an additional pass over the input list. Thus it seems that two passes are required for the functional solution.

The problem can be solved in a single pass in a logic language. Here is a solution in Prolog-like syntax by Reddy [Red86]:

translate(*Inlist*, *Outlist*) :- *map*(*Inlist*, *Outlist*, *Table*, 1).

map([ ], [ ], [ ], _).
map([def(*A*) | *Inlist*], *Outlist*, *Table*, *N*) :-
    member(assign(*A*, *N*), *Table*), *map*(*Inlist*, *Outlist*, *Table*, *N*+1).
map([use(*A*) | *Inlist*], [ref(*Addr*) | *Outlist*], *Table*, *N*) :-
    member(assign(*A*, *Addr*), *Table*), *map*(*Inlist*, *Outlist*, *Table*, *N*).

member(*A*, [*A* | *X*]).
member(*A*, [*B* | *X*]) :- *member*(*A*, *X*).

We use lowercase letters for DEF, USE, ASSIGN and REF because of Prolog convention that symbols starting with uppercase letters denote variables. The predicate *translate* represents the top level invocation. It is assumed that the variable *Inlist* is always bound to a ground list, i.e., a list with no logical variables. Further, if all symbols appearing in USE(x) elements of the input list have corresponding DEF(x) elements in the list, the output list *Outlist* will be ground as well. In order to satisfy predicate *translate*, predicate

map has to be satisfied. *Map* has two additional arguments. The third argument is the symbol table; it records the associations between symbols and numbers assigned to them. The last argument is the initial "seed" for the numbers assigned to symbols. There are three clauses for the *map* predicate. The first one specifies the translation of definitions in the input. The corresponding entry is placed in the table by the predicate *member*. It is presumed that there is no entry for this symbol in the table (i.e., there are no multiple DEFs of the same symbol). *Member* will go through the table until it reaches the end of it, which is always a logical variable. It will instantiate this variable as a new list which has ASSIGN entry as head and a new logical variable as tail. The second clause handles the translation of USE elements. When USE(a) is encountered, the symbol table is looked up by *member*. If the corresponding DEF entry has been already processed, the number associated with the symbol will be instantiated in the output list. On the other hand, if the corresponding DEF entry has not been processed yet, an ASSIGN entry will be placed in the symbol table, with a logical variable as the address for the symbol. This variable will be instantiated when DEF is encountered.

An interesting thing to note is the information about the *modes* of variables in the program above, in particular of the predicate *map*. By modes of logical variables we mean the information specifying whether a logical variable is used as:

- an *input parameter* in which case we say the variable is in the *in* mode.
- an *output parameter* in which case we say the variable is in the *out* mode.
- both an input and an output parameter; in this case the binding for the variable contains unbound logical variables. Some calls use it for input and some for output. We say the variable is in "don't know" mode.

There has been substantial amount of research on determining modes of variables in logic programs; for more information the reader is referred to literature [Deb89, DW88, Smo84, BLM83, Bru82]. In our logic program solution, variables *Inlist*, *A* and *N* are in *in* mode whereas *Outlist* is in *out* mode. *Table* is in "don't know" mode because it can be bound to a partially instantiated structure that contains unbound logical variables.

The key point is that number parts of ASSIGN entries that describe mappings of symbols are represented by unique logical variables. Each occurrence of a USE is replaced by a REF which refers to the number from the corresponding ASSIGN entry, which might be a logical variable. The logical variables in the entries for different symbols will be different even if they are not instantiated. When the corresponding DEF entry is later encountered, the logical variable is instantiated and its binding will become immediately transparent to all REF occurrences since they all share the same variable.

This point has been used as an example of an effect which can be easily achieved in logic languages but apparently cannot be achieved in functional languages [Red86]. The claim was that logic languages have more expressive power than first order functional languages because of presence of such effects.

1.4.1 A functional solution

The argument from the preceding section does have a certain intuitive appeal in that indeed it is not obvious how to obtain a functional solution which solves the problem in a single pass. However, there is a functional solution to this problem. Not very surprisingly, it involves lazy evaluation, which is known to be one of the most powerful features of functional programming. There are several possible directions from which this problem can be approached. Functional programs sometimes can contain repeated traversals of data structures that are unnecessary; this results in a loss of efficiency. There are known methods for elimination of multiple passes over data in functional programs by use of mutually recursive definitions [Bir84]. The basic idea is to define values by sets of equations so that the values define not only the output values but in effect the values obtained after intermediate passes. The essential point is that the values obtained as results of intermediate passes serve both as output, namely of the passes that produced them, and as input, namely to the subsequent passes in the computation. This means that these values will serve both as input and output which is why they have to be recursively defined. Using this strategy, it is possible to eliminate multiple passes from functional programs.

Another point of view is provided by *Attribute Grammars*. It is possible to construct an attribute grammar that solves the problem. The traversal of the input list can be viewed from this point of view as a kind of parsing pass. The result of the translation can be considered as an annotated abstract syntax tree. By annotated it is meant that this tree contains additional information about the translations of the elements in the list. Before proceeding further, we are going to give a brief introduction to attribute grammars.

Attribute grammars

We assume the reader is already familiar with *Context Free Grammars* (CFG). We will provide just a brief overview here.

Definition 15 (Context Free Grammars) A *context free grammar* $\langle N, T, P, S \rangle$ is a quadruple consisting of:

- the set of *nonterminal symbols* (nonterminals) N
- the set of *terminal symbols* (terminals) T
- the set of *productions* P
- the *starting symbol* $S \in N$

All sets in the above definition are finite. Each production $p \in P$ is of the form $p : X_0 \rightarrow X_1 \ldots X_{n_p}$ where X_0 is a nonterminal and $X_1 \ldots X_{n_p}$ is (possibly empty) sequence of terminals and nonterminals.

Definition 16 (d-trees) A *d-tree* is an ordered tree with nodes labeled with the productions of a context free grammar. If $p : X_0 \rightarrow X_1 \ldots X_{n_p}$ is a node in a d-tree, then its children are productions for nonterminals in $X_1 \ldots X_{n_p}$. Its root is labeled with a production for the starting symbol of the grammar. The leaves of a d-tree are labeled by terminals.

Each d-tree is associated with a string, i.e., a sequence of nonterminals. This sequence can be derived from productions of the grammar by listing the leaves of a d-tree in left-to-right order. The tree records how the string can be derived.

An *attribute grammar* is obtained from a context free grammar by associating with each nonterminal symbol a set of *attributes*. Attributes are simply values from some domain. There are two kinds of attributes:

- *inherited* attributes
- *synthesized* attributes

Distinct occurrences of a nonterminal symbol in a production have distinct occurrences of attributes associated with them. Distinct occurrences of attributes are called *attribute occurrences* or *attribute positions*. Given a nonterminal symbol $X \in N$, the sets of its synthesized and inherited attributes will be designated *Syn(X)* and *Inh(X)*, respectively.

With each production $p : X_0 \rightarrow X_1 \ldots X_{n_p}$ a set of rules called *semantic rules* is associated that defines the values of the set $Syn(X_0)$ of synthesized attributes of X_0 and the set $Inh(X_i)$ of inherited attributes of X_i for all nonterminal symbols X_i $1 \leq i \leq n_p$. Each rule is in the form

$$a(i) = f_{p,a,i}(a_1(i_1), \ldots, a_k(i_k))$$

where p is a production, and either $i = 0$ and $a \in Syn(X_0)$, or $1 \leq i \leq n_p$ and $a \in Inh(X_i)$. Symbols $a_j(i_j)$ are attribute positions for attributes of nonterminals in the right hand side of a production. Attribute positions are indexed by positions of nonterminal symbols in the production.

We can think of synthesized attributes as performing the bottom-up propagation of information in d-trees; inherited attributes perform the top-down propagation of information. Consider a production $p : X_0 \rightarrow X_1 \ldots X_{n_p}$. The inherited attributes of X_0 and synthesized attributes of X_i, $1 \leq i \leq n_p$ can be considered as *input* positions; inherited positions of X_0 are inputs from the top part of the d-tree, while synthesized attribute positions are inputs from the bottom of the d-tree. Dually, the synthesized attribute

positions of X_0 and inherited attribute positions of nonterminals in the right hand side can be viewed as outputs. Synthesized attributes of X_0 are outputs to the top of the d-tree and inherited positions for X_i, $1 \leq i \leq n_p$ are outputs to the bottom of the d-tree.

Definition 17 An *annotated d-tree* for an attribute grammar G is obtained from a d-tree for the underlying context free grammar by assigning values to all attribute positions of the nonterminals such that all rules are satisfied.

From now on, the term d-tree will refer to annotated d-trees. The main problem in Attribute Grammars is to compute the values of all attributes in an annotated d-tree. In order to solve this problem, a notion on *evaluation order* is introduced; this order is a total order on attribute positions specifying in what order to compute them. This problem has been heavily studied in literature on Attribute Grammars; for our purposes it suffices to assume that the evaluation order always exists. Indeed, we will see that this order is implicitly determined by the underlying functional implementation of attribute grammars.

The connection between attribute grammars and functional programming was demonstrated [Joh87a]. The motivation for our example is to show that this paradigm is related to logic programming as well.

An attribute grammar solution

The address translation problem can be solved using attribute grammars. In a way, the pass over the input list can be considered as parsing of the input list, and the resulting list can be viewed as an annotated d-tree. The translation process can be performed by propagation of attributes. The following grammar solves the problem

$Ltop \rightarrow L$:

$L{\downarrow}transl = L{\uparrow}tab$

$L{\downarrow}n = 0$

$Ltop\uparrow out\_list = L\uparrow out\_list$

$L \rightarrow \mathrm{DEF}(a).L_{rest}$:

$L\uparrow out\_list = L_{rest}\uparrow out\_list$

$L\uparrow tab = \mathrm{ASSIGN}(a, L\downarrow n).L_{rest}\uparrow tab$

$L_{rest}\downarrow transl = L\downarrow transl$

$L_{rest}\downarrow n = (L\downarrow n)+1$

$L \rightarrow \mathrm{USE}(a).L_{rest}$:

$L\uparrow out\_list = \mathrm{REF}(lookup\ a\ L\downarrow transl).L_{rest}\uparrow out\_list$

$L\uparrow tab = L_{rest}\uparrow tab$

$L_{rest}\downarrow transl = L\downarrow transl$

$L_{rest}\downarrow n = L\downarrow n$

$L \rightarrow [\,]$:

$L\uparrow out\_list = [\,]$

$L\uparrow tab = [\,]$

Regarding the notation, productions have ':' at the end; the subsequent equations are the semantic equations (rules) for the production; inherited attributes of a nonterminal are designated by $\downarrow$ and the synthesized by $\uparrow$. The symbols L_{rest} and L are used for instances of the nonterminal symbol for the list to be parsed, i.e., they can be viewed as different instances of the same symbol. The more conventional way is to distinguish different occurrences by numerical indices, but we adopt this notation for the sake of clarity and to emphasize the connection to functional style of presentation [Joh87a]. The sets of inherited and synthesized attributes for each nonterminal are shown below

$Inh(Ltop) = \{\}$

$Syn(Ltop) = \{out\_list\}$

$$Inh(L) = Inh(L_{rest}) = \{transl, n\}$$
$$Syn(L) = Syn(L_{rest}) = \{tab, out\_list\}$$

Let us see how this grammar solves the problem. The starting symbol is *Ltop*. Synthesized attribute *out_list* is the resulting list. The synthesized attribute *tab* represents the symbol table as it is being built. As we can see in the second production, a new ASSIGN entry is entered into the symbol table whenever a DEF element is encountered. If we look at the third production, we can see that this attribute is simply passed from the nonterminal at the right hand side to the nonterminal at the left hand side. This corresponds to "bottom-up" flow of information. In other words, as the list is being traversed, i.e., "parsed", the symbol table is built and passed upwards in the tree. If we examine all productions, we can notice that this passing will continue all the way up to the starting production. On the other hand, if we examine third production, we can see that the symbol table that is consulted to find the number assigned to the symbol is passed by the inherited attribute *transl*, which is passed from the nonterminal at the left hand side. Obviously, this corresponds to "top-down" flow of information. It is essential to realize that the symbol table used for lookup is obtained from a different attribute than the symbol table used for inserting of definitions. Let us call these tables the *lookup* table and the *insert* table, respectively. The lookup table is represented by the inherited attribute *transl* and the insert table is represented by the synthesized attribute *tab*. The lookup table is used for lookup because we want it to have complete information, in other words to contain *all definitions*. The lookup table should be the same as the table produced after the first pass in the two-pass functional program. In contrast to this, the insert table is passed by a different attribute. As definitions are processed, the corresponding ASSIGN entries are entered into the insert table which is being passed upwards by the synthesized attribute *tab*. If we look at the starting production, we can see that the symbol table obtained from the attribute *tab* from the inside of the tree is simply passed back by the attribute *transl*. This point is essential. At the root of the tree, we know that the entire list has been seen, i.e., no more definitions will be found.

That is why we can pass the table from the *tab* attribute to the *transl* attribute. We know that the table from *tab* is now complete! It is clear that this solution takes only a single pass, because the parsing is done in a single pass. However still it might not be clear that these attributes can be evaluated in a single pass, in other words during parsing. A good way to further clarify that this solution works is to consider a functional program implementing this attribute grammar. This program is going to be a single pass functional solution.

Here is a functional program which performs the task. This program has been created using the algorithm described in [Joh87a]. The algorithm describes how to translate an attribute grammar to a functional program. Here is the program:

```
rec
    Ltop l =
        let rec (l_out_list, l_tab) = L l l_n l_transl
        and L_out_list = l_out_list
        and l_transl = l_tab
        and l_n = 0
        in
            L_out_list
and
    L ((DEF c).rest) L_n L_transl =
        let (rest_out_list, rest_tab) =
            L rest rest_n rest_transl
        and L_out_list = rest_out_list
        and L_tab = ASSIGN(c, L_n).rest_tab
        and rest_n = L_n + 1
        and rest_transl = L_transl
        in
            (L_out_list, L_tab)
```

```
||
    L ((USE c).rest) L_n L_transl =
        let (rest_out_list, rest_tab) =
            L rest rest_n rest_transl
        and L_out_list =
            REF(lookup c L_transl).rest_out_list
        and L_tab = rest_tab
        and rest_n = L_n
        and rest_transl = L_transl
        in
            (L_out_list, L_tab)
||
    L [ ] L_n L_transl =
        ([ ],[ ])
and
    lookup c (ASSIGN(d, n).rest) =
        if c = d then n else lookup c rest end
```

We are going to consider a simple example in order to gain better understanding of the functional solution. Since there is a close correspondence between the functional solution and the attribute grammar, the example will illustrate the attribute grammar solution as well. Consider input list

USE(a).DEF(a).DEF(b).[ ]

The translation is performed by the top-level function *Ltop*. The result expression after a few simple reduction steps is shown below (the symbol ⇒ is used to indicate reduction steps)

Ltop (USE(a).DEF(a).DEF(b).[ ]) ⇒

let rec (*l_out_list*, *l_transl*) = *L* (USE(a).DEF(a).DEF(b).[ ]) 0 *l_transl* **in** *l_out_list*

⇒

let rec (*l_out_list*, *l_transl*) =

  let (*rest_out_list*, *rest_tab*) = *L* (DEF(a).DEF(b).[ ]) 0 *l_transl*

  in (REF(*lookup* a *l_transl*).*rest_out_list*), *rest_tab*

in *l_out_list*

⇒

let rec (*l_out_list*, *l_transl*) =

  let (*rest_out_list*, *rest_tab*) =

    let ($rest\_out\_list_1$, $rest\_tab_1$) = *L* (DEF(b).[ ]) 1 *l_transl*

    in ($rest\_out\_list_1$, (ASSIGN(a, 0)).$rest\_tab_1$)

  in (REF(*lookup* a *l_transl*).*rest_out_list*), *rest_tab*

in *l_out_list*

Looking at the equations we can see that *l_transl* is equal to *rest_tab* which is in turn equal to ASSIGN(a, 0).$rest\_tab_1$. The result of *lookup* a *l_transl* will be 0 because of the ASSIGN entry in the table. Note that the result will be 0 regardless of the value of $rest\_tab_1$ since the evaluation is demand driven and the demand is satisfied when the ASSIGN entry for the desired symbol is found. Finally, the output list *l_out_list* will be equal to REF(0).$rest\_out\_list_1$. Since the output is printed incrementally, the REF entry will be printed before evaluating the rest of the output list. In particular, the demand for USE(a) did not need to look at the list past DEF(a). In principle, we could have even omitted the equation for the case for the empty list which is supposed to terminate the computation. The output would still have been printed and the evaluator would have suspended waiting for further input.

There are several important points in the functional solution which should be emphasized:

- The translation process is relatively straightforward. For each nonterminal in the

grammar, there is a function corresponding to it. This function takes as arguments inherited attributes of the nonterminal, and returns as the result a tuple composed of the values of synthesized attributes. Each production is translated to an equation that defines the function associated with the nonterminal at the left hand side of the production. Clearly, there might be several equations for the same function reflecting the fact that there might be several productions with the same nonterminal at the left hand side.

- The equations defining values of attributes can be recursive. In this example, this is the case with the variables *l_transl* and *l_tab* in the equation for *Ltop*. The values of these variables have to be defined recursively because *l_transl* has to be supplied to the call to *L* as inherited attribute, but *l_transl* is obtained from *l_tab* which is in turn obtained from the call to *L*! The need for recursive definitions is not very surprising since the semantic equations in attribute grammars can be mutually recursive.

By examining the equations we can see that the program *demands* the computation of the rest of the list when the corresponding definition is not in the symbol table. The demand for a REF entry will be satisfied precisely when the corresponding DEF entry is encountered. The corresponding ASSIGN entry will be put in the symbol table. In the case when definitions are preceding uses, all lookups will find corresponding ASSIGN entries and they will return happily with the results. In the case when a use precedes the corresponding definition, the program will traverse the input list until it finds the definition for the use which is being processed. Of course, during the traversal, the other elements of the list will be processed in the same manner.

The entire example demonstrates that it is possible to have functional programs which exhibit effects similar to logical variables. The key is that the value of the symbol table is recursively defined. These effects rely on the use of lazy evaluation and mutually recursive data definitions. If we look at the example again, it is quite clear how the information about the modes is included in the attribute grammar and the corresponding

functional program. All variables which have either in or out mode in the logic program have corresponding attributes in the attribute grammar. What about variables which are neither in in or out mode? *Table* is such a variable. The essential point is that there is a pair of attributes associated with it, namely *tab* and *transl.* This is in contrast to the variables that have either input or output mode for which there is only one attribute associated with them, either inherited or synthesized depending on the mode. Indeed if we go back and look at how the functional program works, we can see that it is precisely through the interaction of the two attributes associated with *Table* that the program achieves the effect of indefinite mode. To recall briefly, the idea is to pass the value of the symbol table as it is being constructed from the DEF entries up in the d-tree and at the top of it pass it back down the tree knowing that it contains all definitions. This top-down propagation is achieved through inherited attribute *transl.*

At this point we can see the idea that is basically at the heart of this research and was the inspiration for it. In concise terms, the idea is that if it is possible to achieve in a functional program the effect of indefinite mode of logic variable *Table* that is in "don't know" mode, then why not apply the same technique to *all logical variables* in a program? This technique would be the basis for an implementation of logic programs in a functional language, via attribute grammars. In fact, attribute grammars can be considered only as a conceptual tool that help us to understand better the process of translation. It would not be necessary to use attribute grammars in order to, say, give semantics to the resulting functional program. The functional solution can be analyzed in its own light using functional programming methodology. Of course, this does not imply by any means that the attribute grammar point of view should not be used, instead it simply notes that it is not necessary to do so. It might very well be possible to use some of the results from the attribute grammars that deal with optimal evaluation of attributes to optimize the functional program. But the main focus of this research is on analysis of the translated logic programs as functional programs. This analysis has several facets; it deals with issues such semantics of resulting functional programs, methods for their execution, comparison with the more traditional logic programming point of view and

others. A concise way to express the goal of this analysis is to investigate what can be learned about logic programming from the functional point of view.

Chapter 2

Translation

This chapter presents an algorithm that translates logic programs to functional programs. The algorithm is simple and straightforward. To illustrate it, we first consider a small example.

2.1 An example

Consider the logic program:

$p(X, Z) : -q(X, Y),\ r(Y, Z).$
$q(X, X).$
$r(X, X).$

$: -p(X, a).$

There are three binary predicates p,q,r. Uppercase letters designate logical variables while lowercase letters designate constants. We want to translate this logic program to a functional program. For every predicate of arity n in the logic program, a function is created. This function takes n arguments and returns n-tuples as results. Intuitively, we can associate with every argument position of a predicate two values:

- an argument position of the function corresponding to the predicate
- a component of the output (tuple)

The argument position can be thought of as the input binding and the corresponding component of the output tuple can be thought of as output binding. At this step, no consideration has been given to repeated occurrences of variables; if a variable occurs repeatedly, there are distinct identifiers associated with every occurrence. Obviously, we would like to specify that the values corresponding to different occurrences are equal. For this reason equations are generated that correspond to the constraints imposed by the repeated occurrences of logical variables. Note that the equality in the equations is the usual (strict) equality function on the type of values the variable can assume. More precisely, the equality here is the function that checks if its arguments are equal, and if they are it returns their (common) value. Otherwise the equality test fails. In this example this means that we are interested in a single solution only. If any of the equality tests in a program fail, so does the whole program.

In our example, functions F_p, F_q, F_r are created to correspond to predicates p, q, r respectively. F_p takes two arguments, say x_{in} and z_{in}. It returns a pair of values as a result. But what should these values be? We assume that F_p gets the output values from applications of F_q and F_r. So F_p returns a pair of values corresponding to the values of logical variables X and Z, which are the arguments of predicate p. The binding for X is obtained as the first component of the pair returned by F_q. Similarly the binding for Z is obtained as the second component of the pair returned by F_r. What should be the value passed to F_q and F_r as the binding for the logical variable Y? Let us call this value y and assume it is defined for a moment. We have now a definition of F_p:

$$
\begin{array}{lll}
F_p\ x_{in}\ z_{in} = & \textbf{let rec} & x_{out}, y_{outq} = F_q\ x_{in}\ y \\
& \textbf{and} & y_{outr}, z_{out} = F_r\ y\ z_{in} \\
& \textbf{in} & x_{out}, z_{out}
\end{array}
$$

The value denoted by y has not been defined in the equation above. Furthermore, since the logical variable Y appears only in the body of the clause defining predicate p in the logic program, it is not clear how its binding should be defined. Let us look at this

problem more closely. We know that the only sources of bindings for Y are occurrences of predicates q and r. So we would expect that the bindings for Y are returned as the corresponding components of tuples returned by F_q and F_r, namely the second component of the result of F_q and the first component of the result of F_r. The bindings are defined in this way is because of the positions at which Y occurs in q and r. We expect these bindings to agree since they correspond to the same logical variable, so we impose a restriction expressing that they are equal. The function *eqc* corresponds to the usual equality predicate for non-functional types in functional languages in that it compares its arguments for equality. It is strict in both of its arguments. There is also a difference, however, since *eqc* does not return the boolean value representing the result of the test but instead in case the arguments are equal it returns their common value; otherwise it fails. Here is its definition:

$$eqc\ x\ y = \textbf{if } x = y \textbf{ then } x \textbf{ else } fail$$

We can define value y, which is undefined in the above equations, to be the result of comparison of the corresponding output components of F_q and F_r. We obtain the following set of equations:

$$\begin{array}{lllll} F_p\ x_{in}\ z_{in} = & \textbf{let rec} & x_{out}, y_{outq} & = & F_q\ x_{in}\ y \\ & \textbf{and} & y_{outr}, z_{out} & = & F_r\ y\ z_{in} \\ & \textbf{and} & y & = & eqc\ y_{outq}\ y_{outr} \\ & \textbf{in} & x_{out}, z_{out} & & \end{array}$$

We can see that all values are defined in the set of equations above. In addition, every identifier appears exactly once on the left hand side of an equation. This requirement needs to be satisfied so there are no multiple definitions of values for identifiers. It is important to note that the value y, which was undefined in the first version of the program, is now defined by a mutually recursive definition because it occurs both on the

right and left hand sides of equations.

The definitions of F_q and F_r follow the same pattern. Since they both have repeated occurrences of the same variable, the corresponding input values are equated and returned as results:

$$F_q\ x_{in1}\ x_{in2} = \textbf{let}\ x_{out} = eqc\ x_{in1}\ x_{in2}\ \textbf{in}\ x_{out}, x_{out}$$

$$F_r\ x_{in1}\ x_{in2} = \textbf{let}\ x_{out} = eqc\ x_{in1}\ x_{in2}\ \textbf{in}\ x_{out}, x_{out}$$

Finally, we are left with the definition of the goal. For the goal we have the same situation for logical variables as for the variable Y in the definition of p; namely, there are no input values to be passed as arguments to F_p. But we can adopt the same solution by creating a set of equations defining a recursive data definition. Here is the equation corresponding to the goal:

$$\textbf{let rec}\ x, a = F_p\ x\ a\ \textbf{in}\ x$$

Now we have the complete translation of our example. Note that the set of equations we have represents a syntactically valid functional program. We can submit it as a program to an evaluator for lazy functional language, such as LML.

2.1.1 Discussion

We would like to try to execute our simple program. Unfortunately, if we execute this program, it will loop. Does this mean that the set of equations specifying our program does not have solutions? The answer to this question is no. There is an obvious solution (in which all variables are equal to constant a). To see this, let us simplify by reduction the top-level expression using the reduction rules:

$$
\begin{array}{l}
\textbf{let rec } x, a = F_p\ x\ a \textbf{ in } x \\
\Rightarrow \\
\textbf{let rec } x, a = \ \textbf{let rec } x_{out}, y_{outq} = F_q\ x\ y \\
\qquad\qquad\qquad\quad \textbf{and } y_{outr}, z_{out} = F_r\ y\ a \\
\qquad\qquad\qquad\quad \textbf{and } y = eqc\ y_{outq}\ y_{outr} \\
\qquad\qquad\qquad\quad \textbf{in } x_{out}, z_{out} \\
\textbf{in } x \\
\Rightarrow \\
\textbf{let rec } x, a = \ \textbf{let rec } x_{out}, y_{outq} = eqc\ x\ y, eqc\ x\ y \\
\qquad\qquad\qquad\quad \textbf{and } y_{outr}, z_{out} = eqc\ y\ a, eqc\ y\ a \\
\qquad\qquad\qquad\quad \textbf{and } y = eqc\ y_{outq}\ y_{outr} \\
\qquad\qquad\qquad\quad \textbf{in } x_{out}, z_{out} \\
\textbf{in } x \\
\Rightarrow \\
\textbf{let rec } x, a = eqc\ x\ y, eqc\ y\ a \\
\textbf{and } y = eqc\ (eqc\ x\ y)\ (eqc\ y\ a) \\
\textbf{in } x
\end{array}
$$

It is quite clear that the above set of equations has a solution, $x = y = a$, since this substitution satisfies all equations. However, from the functional programming point of view, the solution of the above set is $\perp$ ($\perp$ denotes nontermination). The reason is that *eqc* is strict and even after the first iteration of solving the fixpoint equations the approximations for values still remain $\perp$. This situation can be also illustrated in a simpler example. Consider the equation

$$\textbf{let rec } v = eqc\ v\ 5 \textbf{ in } v$$

This equation recursively defines a value of type *Int* (the type of integers). The recursive definitions of values of non-functional type are allowed in lazy functional languages. The

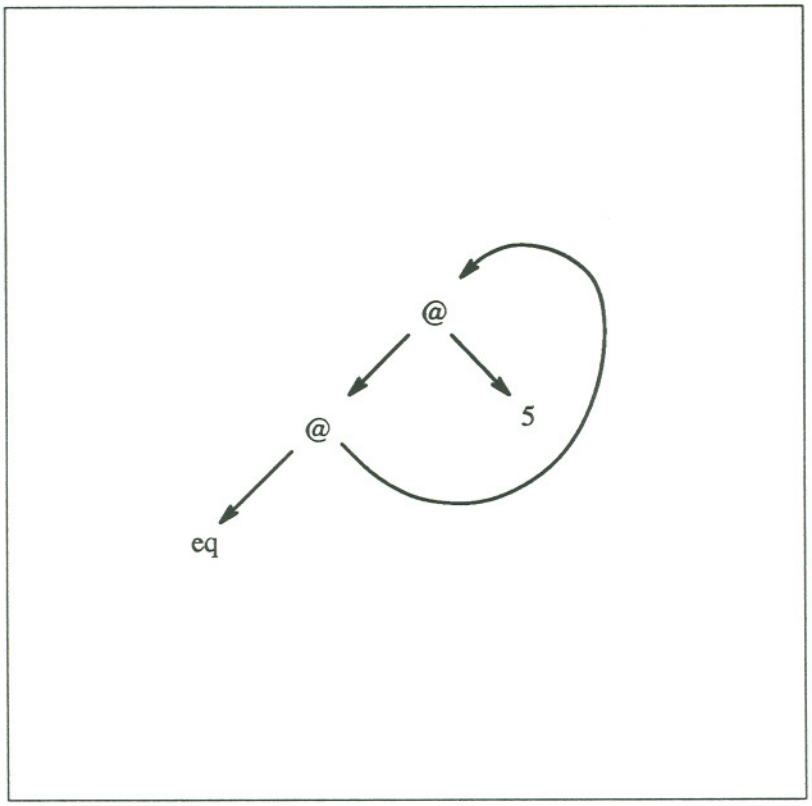

Figure 2.1: Graph representation of a recursive equation

equation above is a valid expression in LML. But the meaning of v according to the least fixpoint semantics of LML is $\perp$. Indeed, if we use $\perp$ as the initial approximation for v, then the right hand side of the definition above is equal to $eqc\ \perp\ 5 = \perp$, and the new approximation for v is still $\perp$. However, it is clear that the equation has another solution, namely $v = 5$. We would like this to be the solution to the equation above instead of $\perp$.

To get an idea how to solve this kind of equation, we can look at the operational representation of such equations in a lazy functional language. A simple way to handle recursive definitions of non-functional type is by cyclic graphs. In the example above, v is represented by a cyclic graph, shown in Figure 2.1. Application nodes are represented by @. The evaluation starts at the redex; since the graph is not evaluated the

left spine is traversed to find the function node. The arguments are stacked along the way. After the node representing *eqc* is reached, it is determined that there are enough arguments to proceed with reduction and the code for *eqc* is entered. Since *eqc* is strict, the very first thing it does is to evaluate its arguments. But its first argument is the starting redex itself and the whole process is repeated indefinitely, i.e., the program loops. Could we somehow prevent this looping? The answer to this question is positive and the solution is quite simple. It is necessary to prevent re-evaluation of a node that is currently being evaluated. In our case this node is the top redex. Every time a reduction is attempted, the redex is marked with a special tag indicating that its reduction is in progress. This way, every redex is checked before its evaluation to see if it is marked; if it is the evaluation is not performed.

2.2 The translation algorithm

As we have seen in the example from Section 2.1, the main idea is to translate a predicate to a function that accepts tuples corresponding to input bindings and returns tuples corresponding to output values. This approach is in the spirit of functional programming, where the emphasis is placed on values and equations over them. The general translation algorithm does not differ from the one outlined in the example; most of the essential features are there. In this section we present the complete translation algorithm. First we give an informal presentation with the emphasis on giving the main points behind it and then we provide a more formal definition.

2.2.1 An informal presentation of the translation algorithm

The set of terms consists of terms built from the constructors in the logic program. For simplicity, we consider only a single sort of terms. However, it is straightforward to generalize this approach to multiple sorts. There are several observations on which the algorithm relies:

- The terms in the logic program are translated to constructor terms in the functional program. The terms in the head of the clause are translated to to a pattern consisting of these terms. We assume that our functional language has pattern matching. A head of a clause such

  $p(c(\ldots),\ldots)$

  is translated to a definition of a corresponding function

  $F_p\ (C(\ldots),\ldots)$

  The argument terms of the head literal are translated as patterns.

- The head literal can contain several occurrences of the same variable. The semantics of the logic program specifies that all these occurrences have the same binding. In the case of pattern matching, nonlinear patterns are usually not allowed, although there are languages in which similar effects can be achieved by using *guards*. We assume that only linear patterns are allowed. All variables in the head literal are translated to different variables in the pattern. We have to make sure that variables corresponding to distinct occurrences of the same variable have the same binding. This check is performed using the function *eqc*, which is essentially the same as in the example in Section 2.1. A head literal with repeated occurrences of a variable such as

  $p(\ldots,Y,\ldots,Y,\ldots)$

  is translated to

$$F_p\ (\ldots\ i_{y1}\ \ldots\ i_{y2}\ldots) = \quad \textbf{let rec}\ \ldots$$
$$\vdots$$
$$\textbf{and}\ i_y = eqc\ i_{y1}\ i_{y2}$$
$$\vdots$$

The equality function *eqc* returns the common value in case its arguments are equal and fails otherwise. The equality function is binary; for the cases in which there are more than two occurrences, we use nested application of *eqc* in an obvious way (eqc_n is n-ary equality function)

$$eqc_n\ x_1\ \ldots\ x_n = eqc\ ((eqc\ (eqc\ x_1\ x_2)\ x_3)\ \ldots)\ x_n$$

- The repeated occurrences of variables in the body of the clause are treated in a similar way. The n-tuples of values returned by the applications of functions from the body are bound to distinct identifiers; the identifiers corresponding to distinct occurrences of the same variable are compared against each other using *eqc*. The resulting values are returned as final bindings. A body literal with two occurrences of the logical variable X

$$p(\ldots, X, \ldots) :- \ldots, q(\ldots, X, \ldots, X, \ldots), \ldots$$

is translated to

$$F_p\ (\ldots, i_x, \ldots) = \quad \textbf{let rec}\ \ldots$$
$$\vdots$$
$$\textbf{and}\ \ldots, s_{x1}, \ldots, s_{x2}, \ldots = F_q\ (\ldots, i_x, \ldots, i_x, \ldots)$$
$$\textbf{and}\ s_x = eqc\ s_{x1}\ s_{x2}$$
$$\vdots$$

Identifiers s_{x1} and s_{x2} are bound to the components of a tuple returned by F_q,

corresponding to two occurrences of variable X. They are compared against each other by *eqc*. The result of the comparison s_x will be returned as a component of the tuple returned by F_p. Of course, the position at which s_x will appear in the result corresponds exactly to the position of the logical variable X in the head literal p.

- The logical variables that occur only in the body of a clause play a special role. We call these variables *existential* variables. The difference in treatment of existential variables comes from the fact that there are no input bindings obtained as the arguments of the function corresponding to the translation of the predicate. The reason for the lack of bindings is because there are no occurrences of existential variables in the head. We define the values of these variable by a recursive data definition

$p(\ldots) : -q(\ldots, Y, \ldots), \ldots$

is translated to

$$
\begin{aligned}
F_p \ \ldots = \ & \textbf{let rec} \ \ldots \\
& \qquad \vdots \\
& \textbf{and} \ \ldots, y, \ldots = F_q \ (\ldots, y, \ldots) \\
& \qquad \vdots
\end{aligned}
$$

The repeated occurrences of existential variables are handled in the same way as repeated occurrences of all variables. A clause such as

$p(\ldots) : - \ldots, q(\ldots, Y, \ldots, Y, \ldots), \ldots$

is translated to

$$
\begin{array}{lll}
F_p \ \ldots = & \textbf{let rec} \ldots & \\
 & \vdots & \\
 & \textbf{and} \ \ldots, y1, \ldots, y2, \ldots = F_q \ (\ldots, y, \ldots, y, \ldots) & \\
 & \textbf{and} \ y = eqc \ y1 \ y2 & \\
 & \vdots &
\end{array}
$$

The definition of y is again recursive, as in the previous case; repeated occurrences are handled in the usual way, by *eqc*.

- Alternate clauses for a predicate are translated to alternate equations in the definition of the corresponding function. A set of clauses $R_1 \ \ldots \ R_n$ is translated to $E_1 \parallel \ \ldots \ \parallel E_n$ where E_i is the translation of clause R_i.

We should mention a minor point. We have seen that recursive definitions are used for existential variables only. For clauses in which there are no existential variables, we do not need **let rec** and **let** can be used instead. This situation is easily detectable by the translator, which can issue **let rec** or **let** accordingly. Most high quality compilers (e.g. LML [Joh87b, Aug87]) perform this transformation automatically, by replacing **let rec** with **let** whenever possible.

Putting everything together, we obtain the algorithm for translation of clauses. A definite clause

$$p(t_1, \ldots, t_n) : -q_1(t_{11}, \ldots, t_{1n_1}), \ldots, q_m(t_{m1}, \ldots, t_{mn_m}).$$

is translated to

$$
\begin{aligned}
F_p\ (it_1, \ldots, it_n) = \quad & \textbf{let rec } st_{11}, \ldots, st_{1n_1} = F_{q_1}\ (it_{11}, \ldots, it_{1n_1}) \\
& \qquad \vdots \\
& \textbf{and } st_{m1}, \ldots, st_{mn_m} = F_{q_m}\ (it_{m1}, \ldots, it_{mn_m}) \\
& \textbf{and } i_{v_1} = eqc_{k_1}\ i_{v_1 1}\ \ldots\ i_{v_1 k_1} \\
& \qquad \vdots \\
& \textbf{and } i_{v_l} = eqc_{k_l}\ i_{v_l 1}\ \ldots\ i_{v_l k_l} \\
& \textbf{and } s_{u_1} = eqc_{r_1}\ s_{u_1 1}\ \ldots\ s_{u_1 r_1} \\
& \qquad \vdots \\
& \textbf{and } s_{u_j} = eqc_{r_j}\ s_{u_j 1}\ \ldots\ s_{u_j r_j} \\
& \qquad \vdots \\
& \textbf{and } \ldots, y_{11}, \ldots, y_{1e_1}, \ldots = F_{q_l}\ (\ldots, y_1, \ldots, y_1, \ldots) \\
& \qquad \vdots \\
& \textbf{and } y_1 = eqc\ y_{11}\ \ldots\ y_{1e_1} \\
& \qquad \vdots \\
& \textbf{in} \\
& \quad st_1, \ldots, st_n
\end{aligned}
$$

The arguments $t_1, \ldots, t_n$ of the head literal p are translated to argument patterns $it_1, \ldots, it_n$ of the function F_p. The patterns are all linear, and different identifiers are used for different occurrences of a variable in a literal. In the translation above, the variables in the head literal are designated by $v_1, \ldots, v_l$. The identifiers assigned to occurrences of these variables are designated $i_{v_j k}$; $i_{v_j k}$ is a variable corresponding to the k-th occurrence of a variable v_j. All occurrences corresponding to the same variable are compared against each other by eqc, and the result of the comparison is designated i_{v_j}. These values are passed as inherited attributes to the functions F_{q_i} corresponding to the literals in the body. The tuples returned by F_{q_i} represent the output bindings and can be thought of as synthesized attributes. Pattern matching is used for binding the components of the result to identifiers corresponding to occurrences of variables in the

body. These variables are designated $u_1, \ldots, u_j$. The identifiers corresponding to different occurrences are designated $s_{u_j k}$. Again, these patterns are linear, and checks have to be performed to assure that all occurrences of the same variable are consistent. The resulting values are designated by $s_{u_1}, \ldots, s_{u_j}$. They are used in the result $st_1, \ldots, st_n$ where each occurrence of a variable v in $t_1, \ldots, t_n$ is replaced by s_v.

The translation of facts is, as we expect, a special case of the translation of clauses since facts are just clauses with empty bodies. A fact

$p(t_1, \ldots, t_n)$.

is translated to

$$
\begin{aligned}
F_p\ (it_1, \ldots, it_n) = \quad & \textbf{let } i_{v_1} = eqc_{k_1}\ i_{v_1 1}\ \ldots\ i_{v_1 k_1} \\
& \qquad\qquad \vdots \\
& \textbf{and } i_{v_l} = eqc_{k_l}\ i_{v_l 1}\ \ldots\ i_{v_l k_l} \\
& \textbf{in} \\
& \qquad st_1, \ldots, st_n
\end{aligned}
$$

Since there is no body, the values $i_{v_1}, \ldots, i_{v_l}$ obtained from inherited attributes are used to construct the result $st_1, \ldots, st_n$ by replacing every occurrence of a variable v in $t_1, \ldots, t_n$ with i_v.

To completely describe the translation algorithm, the translation of the goal clause must be specified. The goal clause can be regarded as a regular clause with an empty head. This fact has important consequences, since it means that all variables in the goal are existential. But we have seen how to handle existential variables, by recursive definitions of bindings. A goal

$:- q_1(t_{11}, \ldots, t_{1n_1}), \ldots, q_m(t_{m1}, \ldots, t_{mn_m}).$

is translated to

let rec $\ldots, y_{11}, \ldots, y_{1e_1}, \ldots = F_{q_1}\ (\ldots, y_1, \ldots, y_1, \ldots)$

$\vdots$

and $y_1 = eqc_{e_1}\ y_{11}\ \ldots\ y_{1e_1}$

$\vdots$

in

$y_1, \ldots, y_n$

We can clearly see that values of all variables are recursively defined and are used to define the resulting tuple $y_1, \ldots, y_n$.

Remark One might believe that there is significant difference in translations of clauses and goals because the translations have been defined separately. In fact, the translations of goals and clauses are practically the same, and the reason for the difference is that goals are clauses with empty heads, so they cannot be translated to function definitions. They are translated instead to expressions in which the bindings for variables are defined recursively. A good example of the relationship between translations of goals and clauses is given by considering a clause such that all variables in its body are existential. The translation of the body of such a clause is the same as if the body has been considered as a goal by itself.

Note that even in translations of clauses in which not all variables are existential, there is a very simple relationship between the translation of a clause and the translation of its body considered as a goal. The translation of the body considered as a goal is obtained from the translation of the clause by deleting the left hand side of the equation in the function definition. The right hand side is considered as an expression and for each

variable, the variables corresponding to input and output bindings are replaced by the same variable.

Example 3 Consider a very simple example; given a clause

$p(X) :- q(X)$

its translation is

$F_p\ i_x = \textbf{let}\ s_x = F_q\ i_x\ \textbf{in}\ s_x$

If we replace occurrences of s_x and i_x in the right hand side of the equation above by a new variable x, we get

$\textbf{let rec}\ x = F_q\ x\ \textbf{in}\ x$

The resulting expression is exactly the same as the translation of the body $:- q(X)$. considered as a goal.

2.2.2 A more formal presentation of the translation algorithm

We introduce some new notation that will be helpful in the presentation of the translation algorithm.

Following Huet, we define an *occurrence* to be a list of integers. Occurrences are used for positioning subparts of terms. We will be interested only in positions of variables in terms. Given an occurrence o of a subterm of a term t, the subterm is extracted by the function *subt* defined as follows:

$$subt\ []\ t = t$$
$$subt\ p.rest\ c(t_1, \ldots, t_p, \ldots, t_n) = subt\ rest\ t_p \qquad 1 \leq p \leq n$$
$$subt\ \_\ \_ = wrong$$

wrong is a special value that indicates there is no subterm of t corresponding to o since the occurrence is not proper for the term.

Example 4 Consider a term $c_0(X, c_1(X, c_2))$. There are two occurrences of variable X in this term. The first one is [1] and the second one is [2, 1].

Given a term t, the set of all variables in t will be designated $Vars(t)$. We will use the same notation for the set of variables in a clause, i.e., the set of variables of a clause C will be designated $Vars(C)$. Given a clause C

$$p(t_1, \ldots, t_n) : -q_1(t_{11}, \ldots, t_{1n_1}), \ldots, q_m(t_{m1}, \ldots, t_{mn_m}).$$

the set of variables in C is divided into two sets:

- $Hvar(C)$—the set of variables which appear in the head of C, i.e.,

$$Hvar(C) = \bigcup_{i=1}^{n} Vars(t_i)$$

- $Evar(C)$—the set of variables appearing only in the body of C. Clearly

$$Evar(C) = Vars(C) - Hvar(C)$$

  These variables will be called *existential* variables.

Given a term t let *Occurrences(t)* be the set of occurrences of all subterms of t. The set of all occurrences of variable v in t will be designated $Occ(v, t)$, i.e.,

$$Occ(v, t) = \{o \mid o \in Occurrences(t), v \in Vars(t), subt\ o\ t = v\}$$

The set of occurrences of a variable v in a clause C is defined analogously. We will need to order sets of occurrences of variables; we introduce *positions* of occurrences of

variables in clauses. An easy way to define positions is to consider the set $Occ(v, C)$ of all occurrences of a variable v in a clause C as a list; this list will be designated $Occ\_pos(v, C)$. The elements of the set $Occ(v, C)$ can be listed in any order—all we are interested in is to assign a unique number n, $n \leq |Occ(v, C)|$ to every occurrence in $Occ(v, C)$. For a set S, $|S|$ is the number of elements in S. We assume there is a function *pos* which gives the position of an occurrence in the list of occurrences. Two sets of new variables are associated with each variable $v \in Hvar(C)$ that occurs in the head in a clause C head,

- the set of *inherited position variables* $IPV(v, C)$.
- the set of *synthesized position variables* $SPV(v, C)$.

The variables in $IPV(v, C)$ and $SPV(v, C)$ are associated with occurrences of v in C, i.e., $|IPV(v, C)| = |SPV(v, C)| = |Occ(v, C)|$. There are bijections between each of $IPV(v, C)$ and $SPV(v, C)$ and $Occ(v, C)$ assigning a variable to each occurrence. We designate these mappings *ivp* and its inverse *ipv* for inherited position variables and *svp* and its inverse *spv* for synthesized position variables. All sets of (both inherited and synthesized) position variables are mutually disjoint.

We also associate with each variable $v \in Hvar(C)$ within a clause two new distinct variables called *inherited* and *synthesized variables for* v, designated $iv(v)$ and $sv(v)$, respectively. These two variables are distinct from all other (inherited and synthesized) position variables.

With each existential variable $v \in Evar(C)$ in a clause C, a set of new variables $EPV(v, C)$ called *existential position variables* is associated. The variables in this set are associated with occurrences of existential variables in C. The corresponding bijection with $Occ(v, C)$ will be designated *epv* and its inverse *evp*. Also, we associate with v a variable called *existential variable for* v written $ev(v)$. As before, these variables are distinct from all other inherited and synthesized position variables, as well as from all other inherited and synthesized variables.

Definition 18 (Renamings) Given two terms t and s, s is a *renaming* of t if there exists a substitution σ such that $t = \sigma s$ and σ has the form

$$\sigma = \{x/y \mid x, y \text{ are variables}\}$$

The renaming substitution has to be injective on variables, i.e.,

$$\forall v_1, v_2 \in \mathit{Vars}(s)\ v_1 \neq v_2 \Rightarrow \sigma(v_1) \neq \sigma(v_2)$$

Renamings are generalized to sequences of terms.

Definition 19 (Linear renamings) Given a term t, a *linear renaming* of t is a term s obtained from t as follows:

- for each variable x in t, the set of distinct occurrences of x is identified; for each of these distinct occurrences a new variable is created (we assume none of the new variables are present in t). The new variables will be designated $y_1, \ldots, y_n$ assuming there are n occurrences of x in t.
- for each variable z such that there is only a single occurrence of z in t, z is replaced by a new variable w.

It is clear that if s is a linear renaming of t, then t is a renaming of s. Occasionally, we will explicitly provide the renaming substitution σ, which will have either the form $\{x/y \mid o \in \mathit{Occ\_pos}(x, C),\ y = \mathit{ivp}(o)\}$ or $\{x/y \mid o \in \mathit{Occ\_pos}(x, C),\ y = \mathit{svp}(o)\}$.

The translation algorithm can be understood as the process of generating a set of directed equations for each clause of a logic program, including the goal clause. Given a logic program P let C_P be the set of clauses of P excluding the goal clause, and let G be the goal clause for P. A clause has the form:

$$p(t_1, \ldots, t_n) : -q_1(t_{11}, \ldots, t_{1n_1}), \ldots, q_m(t_{m1}, \ldots, t_{mn_m}).$$

For each clause $C \in C_P$, a set of equations E is generated; E is the union of three disjoint subsets—$E = E_h \cup E_b \cup E_e$. The subsets E_h, E_b and E_e are defined as follows:

- E_h consists of a single equation

$$F_p\ (it_1, \ldots, it_n) = st_1, \ldots, st_n$$

where $st_1, \ldots, st_n$ is a renaming of $t_1, \ldots, t_n$ and $it_1, \ldots, it_n$ is a linear renaming of $t_1, \ldots, t_n$. The variables used in renamings are from the corresponding inherited and synthesized position variable sets

$$\forall i\ 1 \leq i \leq n\ \forall t_i\ \forall v \in Hvar(t_i)\ \forall o \in Occ\_pos(v, t_i)$$

$$(subt\ o\ it_i = ivp(o)) \wedge (subt\ o\ st_i = sv(v)) \wedge (subt\ o\ t_i = v)$$

There is a small point regarding variables occurring only in the head of the clause. Let $HOvar(C)$ be the set of all such variables in a clause $C = P :- Q_1, \ldots, Q_n$, i.e., $HOvar(C) = Vars(P) - \bigcup_{i=1}^{n} Vars(Q_i)$. Then for every such variable v, we pick the same variable for the synthesized and inherited variable for v, i.e.,

$$\forall v \in HOvar(C)\ sv(v) \equiv iv(v)$$

Note that instead we could add equations $sv(v) = iv(v)$ to the equations generated for the clause, but we adopt this approach since this substitution can be done at compile time.

- E_b consists of equations, one per each literal $q_i, 1 \leq i \leq m$ in the body

$$st_{i1}, \ldots, st_{in_i} = F_{q_i}\ (it_{i1}, \ldots, it_{in_i})$$

where $st_{i1}, \ldots, st_{in_i}$ is a linear renaming of $t_{i1}, \ldots, t_{in_i}$ and $it_{i1}, \ldots, it_{in_i}$ is a renaming of $t_{i1}, \ldots, t_{in_i}$. The variables in the renamings are:

  - from the inherited and synthesized position variable sets in case the corresponding variable in the body appears in the head

$$\forall j,\ 1 \leq j \leq n_i\ \forall t_{ij}\ \forall v \in Hvar(t_{ij})\ \forall o \in Occ\_pos(v, t_{ij})$$

$$(subt\ o\ it_{ij} = iv(v)) \wedge (subt\ o\ st_{ij} = svp(o)) \wedge (subt\ o\ t_{ij} = v)$$

- from existential position variable sets in case the corresponding variable is existential

$$\forall j,\ 1 \leq j \leq n_i\ \forall t_{ij}\ \forall v \in Evar(t_{ij})\ \forall o \in Occ\_pos(v, t_{ij})$$

$$(subt\ o\ it_{ij} = ev(v)) \wedge (subt\ o\ st_{ij} = evp(o)) \wedge (subt\ o\ t_{ij} = v)$$

- E_e consists of three sets of equations

  - Equations generated for the variables in the head literal p that have more than one occurrence; there is one equation for each such variable v

$$i_v = eqc\ i_{v_1}\ (eqc\ i_{v_2} \ldots\ i_{v_{n_v}})$$

$$v \in Hvar(C)\ \ i_v = iv(v)\ \ n_v = |Occ\_pos(v, p)| > 1$$

$$\forall j\ 1 \leq j \leq n_v\ i_{v_j} = ivp(pos\ j\ Occ\_pos(v, p))$$

  - Equations generated for the variables in the body literals q_i that also occur in the head; there is one equation for each such variable v that has more than one occurrence in all the literals in the body. This number is designated m_v.

$$s_v = eqc\ s_{v_1}\ (eqc\ s_{v_2} \ldots\ s_{v_{m_v}})$$

$$v \in Hvar(C)\ \ s_v = sv(v)\ \ m_v = \sum_{i=1}^{m} |Occ\_pos(v, q_i)| > 1$$

$$\forall j\ 1 \leq j \leq m_v\ \ s_{v_j} = svp(pos\ j\ \bigcup_{i=1}^{m} Occ\_pos(v, q_i))$$

  - Equations generated for existential variables that have more than one occurrence in the body

$$e_v = eqc\ e_{v_1}\ (eqc\ e_{v_2} \ldots\ e_{v_{k_v}})$$

$$v \in Evar(C)\ \ e_v = ev(v)\ \ k_v = \sum_{i=1}^{m} |Occ\_pos(v, q_i)| > 1$$

$$\forall j\ 1 \leq j \leq k_v\ \ e_{v_j} = evp(pos\ j\ \bigcup_{i=1}^{m} Occ\_pos(v, q_i))$$

Since facts are clauses with empty bodies, the translation of facts is a special case of translation of clauses. Given a fact F, a set of equations E generated from it is the union of two sets— $E = E_h \cup E_e$, which are defined in the same way as for clauses. Note the absence of E_b since in the translation of clauses it corresponds to equations generated for the literals in the body, which is empty for facts. Also, E_e consists only of equations generated for variables which have multiple occurrences in the head, i.e.,

$$i_v = eqc\ i_{v_1}\ (eqc\ i_{v_2} \ldots\ i_{v_{n_v}})$$

$$v \in Hvar(F) = Vars(F)\ \ i_v = iv(v)\ \ n_v = |Occ\_pos(v,p)| > 1$$

$$\forall j\ 1 \leq j \leq n_v\ \ i_{v_j} = ivp(pos\ j\ Occ\_pos(v,p))$$

Note that all variables in a fact appear by definition only in the head since the body is empty.

A goal clause can be viewed as a clause with no head; naturally this fact has consequences for the translation algorithm. The most important one is that all variables in the goal are by definition existential. Since there is no head, the set of equations E generated for the goal clause is the union of two sets—$E = E_b \cup E_e$. Note the absence of E_h. Because there are no variables other than existential ones, there are no inherited and synthesized variable position sets. Also, E_e consists only of equations generated for existential variables with more than one occurrence in the goal, i.e.,

$$e_v = eqc\ e_{v_1}\ (eqc\ e_{v_2} \ldots\ e_{v_{k_v}})$$

$$v \in Evar(C)\ \ e_v = ev(v)\ \ k_v = \sum_{i=1}^{m} |Occ\_pos(v,q_i)| > 1$$

$$\forall j\ 1 \leq j \leq k_v\ \ e_{v_j} = evp(pos\ j\ \bigcup_{i=1}^{m} Occ\_pos(v,q_i))$$

2.3 Correctness of the translation

We have seen how to translate a logic program to a set of equations. But we certainly want the equations produced by the translation to be sensible with respect to the original

logic program. More precisely, we want the solutions of the equations to agree with the solution of the logic program. The solution of a logic program consists of the *answer* substitution, which makes all conjuncts in the goal satisfiable. The solution of a set of equations consists of a set of values which satisfies all equations. It is important to emphasize that we are looking for *values* since we are taking the functional point of view. The direct consequence is that we will be considering only *ground* answer substitutions, that is, substitutions that map variables to terms with no variables. What about answer substitutions that are not ground? The correspondence should still hold; we want an arbitrary instantiation of variables in a non-ground answer substitution to correspond to a solution of equations obtained by the translation.

Let us look at a simple example that is supposed to illustrate nature of solutions of equations we are interested in. Consider an equation

$$\textbf{rec } v = \textbf{if } v = 5 \textbf{ then } 5 \textbf{ else } \mathit{fail}$$

It defines recursively a value v. The equation has more than one fixpoint. According to the usual fixpoint semantics given to such definitions (in lazy functional languages), this equation has $\perp$ as its solution. But this solution is *least* and it is adopted as the solution in lazy functional languages. In our case we are really not interested in this (trivial) solution, but in a solution that is ground, i.e., is a proper value. In the example, the desired solution is clearly 5. It is a solution because substituting 5 for v satisfies the equation.

2.3.1 An example of reduction

There is exact correspondence between resolution and reduction steps. To illustrate this correspondence consider a simple example

$p(X) :- q(X), r(X).$
$q(X).$
$q(a).$

$r(b)$.

$r(a)$.

Let the goal be

$:-p(Y)$.

A proof tree for this program is shown in Figure 2.2. This proof tree corresponds to the

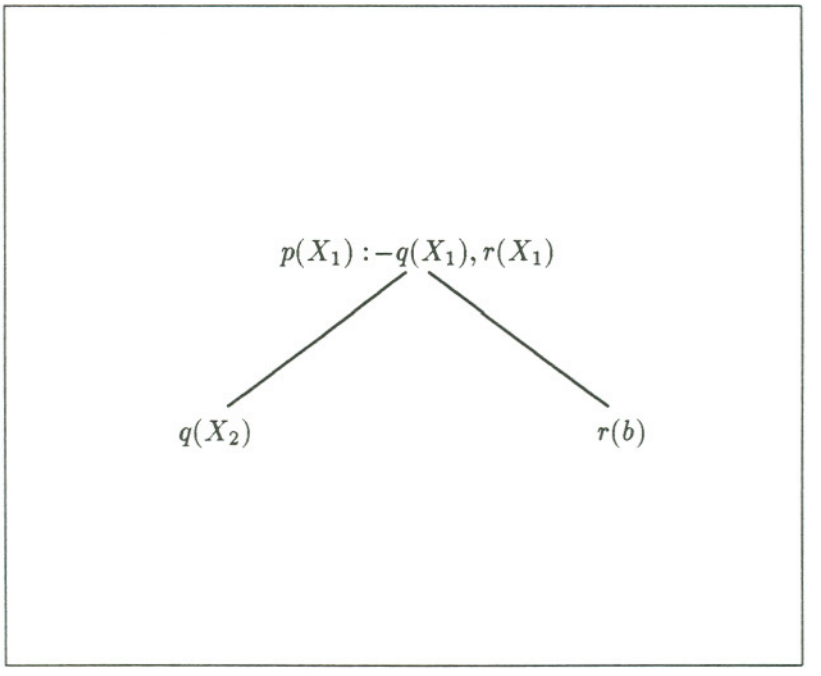

Figure 2.2: A proof tree

following SLD-derivation

$:-p(Y)$.

$:-q(Y), r(Y)$.

$:-r(Y)$.

$:-$

The answer substitution binds Y to b. The translation of the program is

$$
\begin{array}{lll}
F_p\ i_x = & \textbf{let} & s_{x1} = F_q\ i_x \\
& \textbf{and} & s_{x2} = F_r\ i_x \\
& \textbf{and} & s_x = eqc\ s_{x1}\ s_{x2} \\
& \textbf{in} & s_x
\end{array}
$$

$$
\begin{array}{ll}
 & F_q\ i_x = i_x \\
\| & F_q\ a = a
\end{array}
$$

$$
\begin{array}{ll}
 & F_r\ b = b \\
\| & F_r\ a = a
\end{array}
$$

The translation of the goal is

$$\textbf{let rec}\ y = F_p\ y$$

The equation above has a solution assigning constant b to y. To verify this solution, we need to exhibit a reduction sequence demonstrating that the equation is satisfied. Clearly, the first reduction step is the reduction of the application $F_p\ y$. Since we are demonstrating that b is the solution, we need to reduce $F_p\ b$. The reduction of the application of F_p corresponds to the resolution step with the resolvent $p(Y)$. We have seen that each clause is translated to an equation. The reduction and resolution sequences mirror each other in that at each step, the equation used in reduction is the translation of the clause used in resolution. Reducing $F_p\ b$ according to the definition of F_p, we obtain

$$
\begin{array}{ll}
\textbf{let} & s_{x1} = F_q\ b \\
\textbf{and} & s_{x2} = F_r\ b \\
\textbf{and} & s_x = eqc\ s_{x1}\ s_{x2} \\
\textbf{in} & s_x
\end{array}
$$

There are applications of F_q and F_r in the expression above. They correspond to the occurrences of literals $q(Y)$ and $r(Y)$ at the second step of the SLD-derivation. The proof tree indicates the clauses (facts) for q and r used in the derivation. The choices for facts indicate the equations used to reduce applications of F_q and F_r. In particular, we use the first equation in the definition of F_q

$$F_q\ i_x = i_x$$

to reduce $F_q\ b$. We use also the first equation in the definition of F_r

$$F_r\ b = b$$

to reduce $F_r\ b$. Now it is clear that all equations are satisfied and that the reduction sequence mirrors the sequence of resolution steps.

2.3.2 Relating solutions and answer substitutions

We want the solution of the set of equations to correspond to the answer substitution for the logic program. But how do we know that this is indeed the case? The answer is provided by the next theorem:

Theorem 2 Given a logic program P with a goal G, let $E = \mathcal{T}(P \cup G)$ be the set of equations obtained by translating $P \cup G$ using the translation algorithm. Then there is a ground answer substitution σ which is a solution of $P \cup G$ if and only if there is a solution of E. Moreover, these solutions correspond to each other, so the bindings for every variable in the goal in σ are equal to the values for corresponding identifiers in the solution vector for E.

Proof The proof in the **only if** direction is by structural induction and in the **if** direction the proof is by well-ordering induction:

only if There exists an answer substitution σ that satisfies the goal clause. We need to show that all equations in the translation are also satisfied. We use induction on the structure of the proof tree corresponding to the answer substitution.

(i) Base case—the proof tree consists of (a node labeled by) a fact. Let the goal literal G be $q(t_1, \ldots, t_n)$ and the fact F in the proof tree be $q(p_1, \ldots, p_n)$. Then since σ is an answer substitution, $\sigma(t_1, \ldots, t_n) = \sigma(p_1, \ldots, p_n)$. The proof can be decomposed into two parts:

- Consider the equations in the translation of the goal; the equation in E_b has the form

$$l_1, \ldots, l_n = F_q\ (r_1, \ldots, r_n)$$

$l_1, \ldots, l_n$ is a linear renaming of $t_1, \ldots, t_n$ in which all occurrences of variables are replaced by new variables from the set of existential position variables; $r_1, \ldots, r_n$ is a renaming of $t_1, \ldots, t_n$ in which each variable v is replaced by the corresponding existential variable $ev(v)$. It is clear that there exist assignments α_1 and α_2 of ground terms to variables such that $\alpha_1(l_1, \ldots, l_n) = \alpha_2(r_1, \ldots, r_n) = \sigma(t_1, \ldots, t_n)$. They are given by

$$\alpha_1 = \{v_1 \mapsto t \mid v \in \mathit{Vars}(G),\ v_1 = \mathit{evp}(\mathit{Occ\_pos}(v, G)),\ t = \sigma(v)\}$$

$$\alpha_2 = \{v_1 \mapsto t \mid v \in \mathit{Vars}(G),\ v_1 = \mathit{ev}(v),\ t = \sigma(v)\}$$

We use symbol $\mapsto$ to distinguish between assignments, which map identifiers in functional programs to their bindings, and substitutions, which map variables in logic programs to their bindings. These assignments satisfy all equations in E_e since for each variable in the goal, all identifiers in the existential position variable set associated with the variable get assigned the same value, i.e.,

$$\forall v \in \mathit{Vars}(G)\ \forall p \in \mathit{Occ\_pos}(v, G)\ (\alpha_1(\mathit{evp}(p)) = \alpha_2(\mathit{ev}(v)) = \sigma(v) = t)$$

We are left to show that $F_q\ (\sigma(t_1, \ldots, t_n)) = \sigma(t_1, \ldots, t_n)$.

- To demonstrate this, consider the equations in the translation of the fact; the sole equation in E_h has the form

$$F_q\ (ip_1, \ldots, ip_n) = sp_1, \ldots, sp_n$$

where $sp_1, \ldots, sp_n$ is a renaming of $p_1, \ldots, p_n$ and $ip_1, \ldots, ip_n$ is a linear renaming of $p_1, \ldots, p_n$. Since $\sigma(p_1, \ldots, p_n) = \sigma(t_1, \ldots, t_n)$, it follows that there exists an assignment β_1 of ground terms to identifiers such that $\beta_1(ip_1, \ldots, ip_n) = \sigma(t_1, \ldots, t_n)$. It is given by

$$\beta_1 = \{v_1 \mapsto t \mid v \in Vars(F),\ p \in Occ\_pos(v, F),\ v_1 = ivp(p), t = \sigma(v)\}$$

It follows that there is a match in the definition of F_q. All equations in E_e are also satisfied because for every variable in F, all identifiers in the inherited position variable set associated with the variable are assigned the same term i.e.,

$$\forall v \in Vars(F)\ \forall p \in Occ\_pos(v, F)\ (\beta_1(ivp(p)) = \sigma(v) = t)$$

It further follows that the term assigned to the inherited variable associated with each variable is equal to the term assigned to all variables in the inherited position variable set, i.e., there exists an assignment β_2 such that

$$\forall v \in Vars(F)\ \forall p \in Occ\_pos(v, F)\ (\beta_1(ivp(p)) = \beta_2(iv(v)) = \sigma(v) = t)$$

This assignmemnt is given by

$$\beta_2 = \{v_1 \mapsto t \mid v \in Vars(F),\ v_1 = iv(v),\ t = \sigma(v)\}$$

Since $sp_1, \ldots, sp_n$ is a renaming of $p_1, \ldots, p_n$ and the variables in the renaming are exactly the variables in β_2 it follows that $\beta_2(sp_1, \ldots, sp_n) = \sigma(p_1, \ldots, p_n)$ as required.

(ii) Induction case—the root of the proof tree is labeled with a clause C. Assume C has the form

$$q(p_1, \ldots, p_n) : -r_1(p_{11}, \ldots, p_{1n_1}), \ldots, r_m(p_{m1}, \ldots, p_{mn_m})$$

Let G be the goal literal and assume it has the form $q(t_1, \ldots, t_n)$. The structure of the proof is similar to the base case; we need to show that all equations in the translations of the goal and the clause are satisfied:

- Looking at the translation of the goal, E_b consists of the equations of the form

  $$l_1, \ldots, l_n = F_q\ (r_1, \ldots, r_n)$$

  As in the base case, since $l_1, \ldots, l_n$ is a linear renaming of $t_1, \ldots, t_n$ and $r_1, \ldots, r_n$ is a renaming of $t_1, \ldots, t_n$ it is clear there are assignments α_1 and α_2 such that $\alpha_1(l_1, \ldots, l_n) = \alpha_2(r_1, \ldots, r_n) = \sigma(t_1, \ldots, t_n)$. All equations in E_e (for the goal) are satisfied because all identifiers associated with distinct occurrences of a variable in the goal get assigned the same value, namely the binding for the variable in the answer substitution. We need to show that $F_q\ (\sigma(t_1, \ldots, t_n)) = \sigma(t_1, \ldots, t_n)$.
- Consider the equation in E_h for the clause

  $$F_q\ (ip_1, \ldots, ip_n) = sp_1, \ldots, sp_n$$

  where, as before, $sp_1, \ldots, sp_n$ is a renaming of $p_1, \ldots, p_n$ and $ip_1, \ldots, ip_n$ is a linear renaming of $p_1, \ldots, p_n$. Because $\sigma(p_1, \ldots, p_n) = \sigma(t_1, \ldots, t_n)$, it follows that there exists an assignment β_1 of ground terms to identifiers such that $\beta_1(ip_1, \ldots, ip_n) = \sigma(t_1, \ldots, t_n)$. The assignment is given by

  $$\beta_I = \{v_I \mapsto t \mid v \in Hvar(C),\ p \in Occ\_pos(v, F),\ v_I = \mathit{ivp}(p), t = \sigma(v)\}$$

  It follows the pattern in the definition of F_q (i.e., the pattern in E_h) matches the argument in the application of F_q.

- Next we show that all equations in E_b in the translation of C are satisfied. The equations have the form $(1 \leq i \leq m)$

$$sp_{i1}, \ldots, sp_{in_i} = F_{r_i}\ (ip_{i1}, \ldots, ip_{in_i})$$

where $sp_{i1}, \ldots, sp_{in_i}$ and $ip_{i1}, \ldots, ip_{in_i}$ are, respectively, a linear renaming and a renaming of $p_{i1}, \ldots, p_{in_i}$. Recall that renamings are defined in the following way (R_i stands for $r_i(p_{i1}, \ldots, p_{in_i})$)

$$\begin{aligned}
&\forall v \in Hvar(R_i)\ \forall p \in Occ_pos(v, R_i) \\
&(subt\ p\ r_i(sp_{i1}, \ldots, sp_{in_i}) = svp(p)) \wedge \\
&(subt\ p\ r_i(ip_{i1}, \ldots, ip_{in_i}) = iv(v))
\end{aligned}$$

By the inductive hypothesis, for each of R_i, given an answer substitution σ, the equational translation is also satisfied with corresponding bindings. Since we are given σ by hypothesis, it also satisfies each of the R_i. The equational translation for each of R_i differs from the equations in E_b in that all variables in R_i, considered as a goal, are existential. This situation does not necessarily hold for the equations in E_b because some variables might appear in the head. These are exactly the variables in the above renaming. The consequence of this is that for each head variable v, the synthesized and inherited variable for v have equal bindings, namely $\sigma(v)$ (cf. the remark at the end of Section 2.2.1). Expressed formally

$$\forall v \in Hvar(R_i)\ (\beta_1(sv(v)) = \beta_1(iv(v)) = \sigma(v))$$

By the inductive hypothesis all equations in E_b as well as the equations in E_e are satisfied.

- Finally we observe that the resulting tuple $sp_1, \ldots, sp_n$ is a renaming of $p_1, \ldots, p_n$; also $ip_1, \ldots, ip_n$ is a linear renaming of $p_1, \ldots, p_n$. It follows that (Q stands for $q(p_1, \ldots, p_n)$) there exist assignments α_1 and α_2 such

that

$$\forall v \in Vars(Q)\ \forall p \in Occ\_pos(v, Q)$$
$$(subt\ p\ q(sp_1, \ldots, sp_n) = sv(v)) \wedge (\alpha_1(sv(v)) = subt\ p\ \sigma(p_1, \ldots, p_n))$$
$$(subt\ p\ q(ip_1, \ldots, ip_n) = ivp(p)) \wedge (\alpha_1(ivp(p)) = subt\ p\ \sigma(p_1, \ldots, p_n))$$

This concludes the proof since $F_q\ (\sigma(p_1, \ldots, p_n)) = \sigma(p_1, \ldots, p_n)$.

if The proof is by well-ordered induction on the lengths of sequences of reductions that demonstrate the satisfiability of equations. Let us call the goal literal G.

(i) Base case—the reduction sequence consists of a single reduction; the equation used in the translation must be the translation of a fact. Let us call this fact F. Let F have the form $q(p_1, \ldots, p_n)$ and let G be $q(t_1, \ldots, t_n)$. The equation in E_b in the translation of the goal G has the form

$$l_1, \ldots, l_n = F_q\ (r_1, \ldots, r_n)$$

$l_1, \ldots, l_n$ is a linear renaming of $t_1, \ldots, t_n$ with variables from sets of existential position variables; $r_1, \ldots, r_n$ is a renaming of $t_1, \ldots, t_n$ in which every variable is replaced by the corresponding existential variable. The single reduction in the reduction sequence is the reduction of $F_q\ (r_1, \ldots, r_n)$. By the hypothesis there exist assignments α_1 and α_2 of ground terms to variables such that $\alpha_1(l_1, \ldots, l_n) = \alpha_2(r_1, \ldots, r_n)$. But then it follows there also exists substitution σ_1 such that

$$\sigma_1(t_1, \ldots, t_n) = \alpha_1(l_1, \ldots, l_n) = \alpha_2(r_1, \ldots, r_n)$$

It is given by

$$\sigma_1 = \{t/v \mid v \in Vars(G), v_1 = ev(v), t = \alpha_2(v_1)\}$$

Consider the translation of F_q; the equation in E_h has the form

$$F_q\ (ip_1, \ldots, ip_n) = sp_1, \ldots, sp_n$$

Since the reduction is successful, there has to be a match between $\sigma_1(t_1, \ldots, t_n)$ (the argument of F_q) and $ip_1, \ldots, ip_n$. It follows there is a substitution

$$\omega = \{t/v_1 \mid \quad v \in Vars(F), p \in Occ\_pos(v, F), v_1 = ivp(p), \\ t = subt\ p\ (\sigma_1(q(t_1, \ldots, t_n)))\}$$

such that $\omega(ip_1, \ldots, ip_n) = \sigma_1(t_1, \ldots, t_n)$. All equations in E_e are also satisfied so the following is also true

$$\forall v \in Vars(F)\ \forall p \in Occ\_pos(v, F)\ ivp(p) = iv(v)$$

i.e., for every variable in F, the bindings for all variables in the corresponding set of inherited position variables are equal. It follows that there is a substitution σ_2

$$\sigma_2 = \{t/v \mid v \in Vars(F), p \in Occ\_pos(v, F), t = subt\ p\ (\sigma_1(q(t_1, \ldots, t_n)))\}$$

such that $\sigma_2(ip_1, \ldots, ip_n) = \sigma_1(q(t_1, \ldots, t_n))$. Finally, the answer substitution is $\sigma = \sigma_1 \cup \sigma_2$.

(ii) Induction case—E_b in the translation of the goal consists of an equation

$$l_1, \ldots, l_n = F_q\ (r_1, \ldots, r_n)$$

By the hypothesis the equation is satisfied. Reasoning similarly to the base case we conclude there has to be a substitution σ_1 such that

$$\sigma_1(t_1, \ldots, t_n) = \alpha_1(l_1, \ldots, l_n) = \alpha_2(r_1, \ldots, r_n)$$

The reduction sequence for $F_q\ (r_1, \ldots, r_n)$ contains more than one reduction step, so the equation used has to correspond to a clause. Let us call this clause C and assume it has the form

$$q(p_1, \ldots, p_n) :- r_1(p_{11}, \ldots, p_{1n_1}), \ldots, r_m(p_{m1}, \ldots, p_{mn_m})$$

There has to be a match between $\sigma_1(t_1, \ldots, t_n)$ and $ip_1, \ldots, ip_n$ in the definition of F_q. Considering the fact that all equations in E_e for variables with

multiple occurrences are satisfied, we conclude that there exists σ such that $\sigma(t_1, \ldots, t_n) = \sigma(p_1, \ldots, p_n)$. We need to show that σ also satisfies the literals in the body. By the hypothesis there has to be an assignment α_3 of ground values to identifiers in C. It is clear that

$$\forall v \in Hvar(C)\ \forall p \in Occ\_pos(v, C)$$

$$\alpha_3(iv(v)) = \alpha_3(ivp(p)) = \alpha_3(sv(v))$$

The first part of the equality $(\forall v, p\ \alpha_3(iv(v)) = \alpha_3(ivp(p)))$ is satisfied because all equations in E_e are satisfied; the second part$(\forall v, p\ \alpha_3(ivp(p)) = \alpha_3(sv(v)))$ is satisfied because

$$\alpha_3(ip_1, \ldots, ip_n) = \alpha_3(sp_1, \ldots, sp_n)$$

($ip_1, \ldots, ip_n$ and $sp_1, \ldots, sp_n$ are, respectively, a linear renaming and a renaming of $p_1, \ldots, p_n$). It follows the bindings in σ for the head variables in C are defined by

$$\forall v \in Hvar(C)\ \sigma(v) = \alpha_3(sv(v)) = \alpha_3(iv(v))$$

By the inductive hypothesis, σ satisfies all literals in the body.

∎

2.4 Matching versus unification

The process of matching and of unification are instances of the same process. There is, however, a very important difference between them. Matching is used in functional programs, in *pattern matching*. The basic idea in pattern matching is that a term is compared to a pattern; the term is a ground value, i.e., it has no variables. On the other hand, the pattern may contain variables. The process of matching consists of finding a substitution of variables in the pattern that makes the pattern equal to the term. The direction of the flow of information is clearly determined; the data flows from

the term to the pattern. In unification, we again have a term that is to be matched against a pattern. However, in this case, there is no real difference between the term and the pattern. There can be variables in both. Unification consists of finding the substitution, for both the variables in the term and the pattern, such that applying the substitution to both the term and the pattern makes them equal. This difference has very important consequences. Since in the case of matching there can be no variables in the term, there is no need to have distinguished representation of program variables. The variables in the pattern simply serve as placeholders for the actual values in the term. When matching takes place during the execution of a functional program, values are substituted for variables. In contrast, we have seen that in the case of unification there can be variables in both the term and the pattern. Since variables can be present in the term, we have to allow for variables to have a distinguished representation, in the form of logical variables. In addition, logical variables in different instances of a clause have to be renamed so they are distinct. Indeed, the presence of variables in the term seems to indicate that we cannot use matching. The most important question is: if we are to use matching instead of unification, what are the values used for variables in the term? We have seen that the idea in our approach is to define these values as aggregates of all bindings that are compared for equality against variables in the program. The aggregate values are defined by **rec** expressions. Their evaluation proceeds incrementally, and is driven by the demand propagation during the evaluations of applications of predicate functions. The evaluation of function applications can be viewed as being analogous to the traversal of derivation trees in logic programs.

Chapter 3

Semantics

The goal of semantics for a programming language is to give *meanings* to well formed phrases of the language. Meanings are elements of some mathematical domain. There can be many ways of giving a semantics to a language; semantics can focus on different aspects of a language. Logic programming is a good example for this. There are two different semantics which are widely adopted:

- *declarative* semantics given by *Herbrand interpretations*
- *operational* semantics given by *fixpoints* of certain mappings

Our approach is different from either approach above. The spirit of our research has been to provide a functional point of view of logic programming. It is not surprising that our semantics of logic programming will be based on semantics of functional languages.

An important class of models for functional languages is provided by *denotational semantics*. In denotational semantics, each phrase in a program is assigned a *denotation*, which is an element of a mathematical domain.

Our approach relies on the translation of logic programs to sets of equations; these sets of equations can be viewed as programs in a functional language. We are going to give a non-deterministic semantics to this functional language that together with the translation algorithm will give us, indirectly, semantics for logic programs.

3.1 Target language

In this section we define the target language of the translation. The language is quite similar to functional languages; as a matter of fact we can view it as a functional language with a special recursive construct. The syntax is essentially the same as in conventional functional languages and we will see that the semantics is derived by adding a distinguished value into the domain.

3.1.1 Syntax

The syntax of the language is defined by the following grammar:

Prog → Expr

Expr → Constr_id
Id
Expr , Expr
\Pat. Expr {|| \Pat.Expr}*
Expr Expr
let Eq **in** Expr

Eq → Pat = Expr
Eq **and** Eq
rec Eq
lrec Eq

Pat → Constr_id
Constr_id(Pat{,Pat}*)
Id
Pat & Eq

There are a couple of points regarding the notation. The language definition does not contain the conditional expression **if-then-else**. There is no loss in expressive power since conditionals can be represented by patterns, i.e., **if** E_1 **then** E_2 **else** E_3 can be viewed as syntactic sugar for $(\backslash\mathbf{true}.E_2 \parallel \backslash\mathbf{false}.E_3)\ E_1$. Similarly, function definitions such as $f\ p = e$ can be viewed as syntactic sugar for $f = \backslash p.e$.

A program consists of an expression. There are five kinds of expressions:

- constants
- identifiers
- λ-abstractions
- applications
- **let** expressions

The language features pattern matching; λ-abstractions can have patterns as well as left hand sides of equations in **let** expressions. Patterns are built from variables and constructors; they can contain *guards*, which are in the form of equations.

3.1.2 Domains

In this section we present the domains. They are essentially the same as in conventional functional languages [Aug87]. The only difference is the presence of a distinguished value SW.

$$\mathbf{E} = \{SW\} + \mathbf{C} + \mathbf{E} \rightarrow \mathbf{E} + \mathbf{E} \times \mathbf{E} + \{Fail\}$$
$$\mathbf{C} = \mathbf{C}_0 + \ldots + \mathbf{C}_i \underbrace{(\mathbf{E} \times \ldots \times \mathbf{E})}_{i} + \ldots$$
$$\mathbf{Env} = (\mathbf{Id} \rightarrow \mathbf{E}) + \{Fail\}$$

$\mathbf{E}$ is the domain of *values*. It is built from *constructors* $\mathbf{C}$, the function space $\mathbf{E} \rightarrow \mathbf{E}$ and two distinguished values: SW and $Fail$. Each constructor has *arity* associated

with it, which is the number of argument components; constants are viewed as nullary constructors (with arity zero). Intuitively, we can think of SW as a "match all" value, i.e., an attempt to match it against anything succeeds. *Fail* denotes a failure detected during pattern matching. **Env** is the domain of environments, which are mappings from identifiers to expressions.

3.1.3 Semantic functions

We present the semantic functions. Again we follow the spirit of functional programming in the presentation. We assume $\rightarrow$ is right associative.

$\mathcal{E} : \mathbf{Expr} \rightarrow \mathbf{Env} \rightarrow \mathbf{E}$
$\mathcal{D} : \mathbf{Eq} \rightarrow \mathbf{Env} \rightarrow \mathbf{Env}$
$\mathcal{B} : \mathbf{Pat} \rightarrow \mathbf{E} \rightarrow \mathbf{Env} \rightarrow \mathbf{Env}$

There are three semantic functions:

- $\mathcal{E}$, which gives meanings to expressions
- $\mathcal{D}$, which gives meanings to equations
- $\mathcal{B}$, which gives meanings of patterns

$\mathcal{E}$ gives meaning to expressions using environments for identifiers. $\mathcal{D}$ takes an equation and an environment, and produces an environment with bindings for identifiers in the left-hand side of the equation; a check has to be made to ensure that the meaning of the right-hand side expression conforms to the pattern in the left-hand side. If the value does not conform to the pattern, the result is a failure that is represented by *Fail* injected into environments. Checking of patterns is performed by $\mathcal{B}$. It takes a pattern, a value to be matched against the pattern and an environment; in case matching succeeds it produces an environment with the bindings for the variables in the pattern, otherwise the result is failure. In addition to the form $p = e$, equations can be formed in additional ways:

- they can be combined by **and**
- they can be recursive; there are two recursive constructs—**rec** and **lrec**

We use conventional notation for environments [Aug87]. Given a variable x, a value v and an environment ρ, $\rho[x \mapsto v]$ denotes a new environment that has the same mappings for all variables in ρ except that it maps x to v. The operator ∇ is the usual one for concatenating environments; it is *Fail*-strict i.e., $Fail \nabla \rho = \rho \nabla Fail = Fail$.

$$\mathcal{E}[\![c]\!]\rho = c$$

$$\mathcal{E}[\![x]\!]\rho = \rho\ x$$

$$\mathcal{E}[\![\lambda p_1.M_1 \parallel \ldots \parallel \lambda p_n.M_n]\!]\rho =$$
$$\lambda v.\ \mathcal{B}\ [\![p_1]\!]\ v\ \rho \neq Fail \rightarrow \mathcal{E}[\![M_1]\!](\mathcal{B}\ [\![p_1]\!]\ v\ \rho)$$
$$\vdots$$
$$\mathcal{B}\ [\![p_n]\!]\ v\ \rho \neq Fail \rightarrow \mathcal{E}[\![M_n]\!](\mathcal{B}\ [\![p_1]\!]\ v\ \rho)$$
$$Fail$$

$$\mathcal{E}[\![M\ N]\!]\rho = \mathcal{E}[\![M]\!]\rho\ (\mathcal{E}[\![N]\!]\rho)$$

$$\mathcal{E}[\![\textbf{let}\ D\ \textbf{in}\ N]\!]\rho = \mathcal{E}[\![N]\!](\mathcal{D}[\![D]\!]\rho)$$

$$\mathcal{D}[\![p = M]\!]\rho = \mathcal{B}[\![p]\!](\mathcal{E}[\![M]\!]\rho)\rho$$

$$\mathcal{D}[\![D_1\ \textbf{and}\ D_2]\!]\rho = (\mathcal{D}[\![D_1]\!]\rho)\ \nabla\ (\mathcal{D}[\![D_2]\!]\rho)$$

$$\mathcal{D}[\![\textbf{rec}\ D]\!]\rho = fix(\lambda\rho_r.\mathcal{D}[\![D]\!]\rho_r)$$

$$\mathcal{D}[\![\textbf{lrec}\ D]\!]\rho = lfix(\lambda\rho_r.\mathcal{D}[\![D]\!]\rho_r)$$

$$lfix\ F = \lim_{i\to\infty} F^i\rho_0 \quad \forall x \in \rho_0\ \rho_0(x) = SW, \text{ if the limit exists}$$
$$\perp \text{ otherwise}$$

$$\mathcal{B}[\![c_i(p_1, \ldots, p_i)]\!]\ v\ \rho =$$
$$v = \perp \rightarrow \rho\ \nabla\ (\mathcal{B}[\![p_1]\!]\ \perp\ \rho)\ \nabla \ldots \nabla\ (\mathcal{B}[\![p_i]\!]\ \perp\ \rho)$$

$v = SW \rightarrow \rho \nabla (\mathcal{B}[\![p_1]\!]\ SW\ \rho) \nabla \ldots \nabla (\mathcal{B}[\![p_i]\!]\ SW\ \rho)$

$v = c_i(v_1, \ldots, v_i) \rightarrow \rho \nabla (\mathcal{B}[\![p_1]\!]\ v_1\ \rho) \nabla \ldots \nabla (\mathcal{B}[\![p_i]\!]\ v_i\ \rho)$

Fail

$\mathcal{B}[\![x]\!]\ v\ \rho = \rho[x \mapsto v]$

$\mathcal{B}[\![p\ \&\ e]\!]\ v\ \rho = \rho \nabla (\mathcal{B}[\![p]\!]\ v\ \rho) \nabla (\mathcal{D}[\![e]\!]\rho)$

The semantic function $\mathcal{E}$ gives meaning to expressions. The meaning of a constant is the corresponding constant in the domain. The meaning of a variable is obtained by looking up the environment. Functions are defined by λ-abstractions with patterns. Function definitions can have several equations that are combined by the || operator. The meaning of such a definition is defined by non-deterministically choosing a meaning of one of constituent equations. The meanings of equations are defined as follows: the argument value is checked against the pattern in the equation using the $\mathcal{B}$ scheme; if the match is successful, the result is the body of the abstraction evaluated in the environment supplemented with the identifiers in the pattern. The meaning of a **let** expression is equal to the meaning of the body evaluated in the environment produced from the equation. Environments produced from equations joined by **and** are combined together by including identifiers in the components; it is assumed that there are no common identifiers in the component environments. The meanings of recursive equations are defined by recursive environments. There are two kinds of recursive equations:

- **rec** equations whose denotations are defined by fixpoints starting from $\perp$
- **lrec** equations whose denotations are defined similarly by fixpoints; the fixpoints are not least and the initial environments map identifiers to SW. Note that the meaning is defined by a limit, which does not always exist. But we will see in subsequent sections that the limit does exist for the translations of logic programs for which the target language is intended.

The semantics of the language is based on the semantics for functional languages [Aug87, Joh87b, PJ87]. There is an essential addition—the construct **lrec**.

We define an ordering $\preceq$ of terms as follows:

- $\forall t \ \bot \preceq t$
- $\forall t \neq \bot \ SW \preceq t$
- $c(t_1, \ldots, t_n) \preceq c(q_1, \ldots, q_n)$ *if* $t_1 \preceq q_1 \wedge \ldots \wedge t_n \preceq q_n$

SW can be thought of as denoting an "unbound" value.

We have seen in the Section 2.2 that values of existential variables were recursively defined. They are actually defined by **lrec**. The usual **rec** is used for recursive definitions of functions corresponding to predicates.

The limits defining the semantics of **lrec** expressions that are translations of logic programs always exist (it should be said that the limits are always reached after a finite number of steps, provided the corresponding derivation tree is finite). In the general case, the limits do not exist. To see what goes wrong, consider a program

let rec $F_p \ 5 = 6$
$\|$ $F_p \ 6 = 5$
in

let lrec $s = F_p \ s$ **in** s

The sequence of approximations for s is $6, 5, 6, 5, \ldots$ and the limit does not exist. However we will see that in the case we are interested in, the translation of logic programs, the limits do exist.

3.1.4 Translating logic programs to the target language

We can exhibit now the actual translation to the target language; the translation algorithm from Section 2.2 used sets of equations in functional style as the target of the translation. Now we have a language that can be used as the target of the translation.

The principal difference between the target language and equations from the translation algorithm is the presence of guards in the language. In the translation algorithm from Section 2.2 we proved that a solution of equations obtained by translation exists if and only if there is an answer substitution for the logic program. The solution was *declarative* since we did not specify an evaluation order that would determine how to search for solutions. The semantics presented in this chapter is also declarative since we are assuming that equations in function definitions are chosen non-deterministically. An actual implementation would of course need to specify a particular search strategy. We could adopt either:

- Depth-first search, which is more efficient but results in lack of completeness—a solution may not be found even if one exists.
- Breadth-first search strategy, which is complete but less efficient.

Translation of clauses

To recall from Section 2.2, a clause

$$p(t_1, \ldots, t_n) : -q_1(t_{11}, \ldots, t_{1n_1}), \ldots, q_m(t_{m1}, \ldots, t_{mn_m}).$$

was translated to

$$
\begin{aligned}
F_p\ (it_1, \ldots, it_n) = \quad & \textbf{let rec}\ (st_{11}, \ldots, st_{1n_1}) = F_{q_1}\ (it_{11}, \ldots, it_{1n_1}) \\
& \qquad \vdots \\
& \textbf{and}\ (st_{m1}, \ldots, st_{mn_m}) = F_{q_m}\ (it_{m1}, \ldots, it_{mn_m}) \\
& \textbf{and}\ i_{v_1} = eqc_{k_1}\ i_{v_1 1}\ \ldots\ i_{v_1 k_1} \\
& \qquad \vdots \\
& \textbf{and}\ i_{v_l} = eqc_{k_l}\ i_{v_l 1}\ \ldots\ i_{v_l k_l} \\
& \textbf{and}\ s_{u_1} = eqc_{r_1}\ s_{u_1 1}\ \ldots\ s_{u_1 r_1} \\
& \qquad \vdots \\
& \textbf{and}\ s_{u_j} = eqc_{r_j}\ s_{u_j 1}\ \ldots\ s_{u_j r_j} \\
& \textbf{and}\ \ldots, y_{11}, \ldots, y_{1e_1}, \ldots = F_{q_l}\ \ldots, y_1, \ldots, y_1, \ldots \\
& \textbf{and}\ y_1 = eqc\ y_{11}\ \ldots\ y_{1e_1} \\
& \qquad \vdots \\
& \textbf{in} \\
& \quad st_1, \ldots, st_n
\end{aligned}
$$

The actual translation using the language is:

$$F_p\,(it_1,\ldots,it_n)\;\&\;\left(\begin{array}{l}
\textbf{lrec}\ (st_{11},\ldots,st_{1n_1}) = F_{q_1}\ (it_{11},\ldots,it_{1n_1}) \\
\qquad\qquad\vdots \\
\textbf{and}\ (st_{m1},\ldots,st_{mn_m}) = F_{q_m}\ (it_{m1},\ldots,it_{mn_m}) \\
\textbf{and}\ i_{v_1} = eqc_{k_1}\ i_{v_1 1}\ \ldots\ i_{v_1 k_1} \\
\qquad\qquad\vdots \\
\textbf{and}\ i_{v_l} = eqc_{k_l}\ i_{v_l 1}\ \ldots\ i_{v_l k_l} \\
\textbf{and}\ s_{u_1} = eqc_{r_1}\ s_{u_1 1}\ \ldots\ s_{u_1 r_1} \\
\qquad\qquad\vdots \\
\textbf{and}\ s_{u_j} = eqc_{r_j}\ s_{u_j 1}\ \ldots\ s_{u_j r_j} \\
\textbf{and}\ \ldots,y_{11},\ldots,y_{1e_1},\ldots = F_{q_l}\ \ldots,y_1,\ldots,y_1,\ldots \\
\textbf{and}\ y_1 = eqc\ y_{11}\ \ldots\ y_{1e_1} \\
\qquad\qquad\vdots
\end{array}\right) = st_1,\ldots,st_n$$

The values of existential variables are defined by **lrec**. All equations are within a guard; in case any of the equations fails, an alternate equation is tried. If all equations are satisfied, a tuple containing the bindings for variables in the head (of the corresponding clause) is returned as the result.

Translation of goals

A goal

$:-\ q_1(t_{11},\ldots,t_{1n_1}),\ldots,q_m(t_{m1},\ldots,t_{mn_m}).$

is translated to

$$
\begin{array}{l}
\textbf{lrec } \ldots, y_{11}, \ldots, y_{1e_1}, \ldots = F_{q_1}\ (\ldots, y_1, \ldots, y_1, \ldots) \\
\qquad\qquad\qquad \vdots \\
\textbf{and } y_1 = eqc_{e_1}\ y_{11}\ \cdots\ y_{1e_1} \\
\qquad\qquad\qquad \vdots \\
\textbf{and } y_n = eqc_{e_n}\ y_{11}\ \cdots\ y_{1e_n} \\
\textbf{in} \\
\quad y_1, \ldots, y_n
\end{array}
$$

As expected, the bindings for all variables in the goal are defined recursively by **lrec**. Note that the goal expression consists of a tuple of bindings for all variables in the goal. The same information is also contained in the environment produced by evaluating all equations in the goal. In effect, what we are looking for is the environment ρ which is the solution of

$$\rho = \mathcal{D}[\![E_G]\!]\rho$$

where E_G is the translation of the goal. From now on, by evaluating the goal expression we will mean evaluating the goal environment containing bindings for all variables.

3.2 Computing solutions

In Section 2.3 we saw that (ground) answer substitutions correspond to (ground) solutions of sets of equations; we did not say anything about how to compute answer substitutions. The semantics presented in this chapter specifies that the solutions are computed as fixpoints starting from SW. We need to show that values defined by semantic equations are indeed solutions of the equations. To establish this fact, we need to look at the aggregate of all values of recursively defined variables (defined by **lrec**) in the set of equations obtained by translation. Let σ be an answer substitution for a logic program P. Let t be the corresponding proof tree. For our purposes σ will contain bindings for all existential variables in the proof tree. Looking at the translation algorithm, existential variables correspond precisely to variables defined by **lrec** in the translated

equations. The usual definitions of answer substitutions are concerned only with the bindings for the goal; this is easily obtained from our notion by restriction to variables in the goal. They are all existential by definition. The **lrec** construct defines the value of each existential variable. The semantic equations specify that the values of $y_1, \ldots, y_n$ are obtained from an environment ρ that is recursively defined. We will assume that ρ defines bindings for all existential variables in the proof tree that will be associated with an answer substitution.

Definition 20 (Evaluation trees) Given a derivation tree t, the *evaluation tree* $\epsilon(t)$ is defined by induction on the structure of t:

- Base case—the proof tree is (a node labeled by) a fact F; let E_F be the translation of F. The evaluation tree $\epsilon(t)$ consists of a node labeled by E_F.
- Induction case—the proof tree is a clause C, which has the form $P :- Q_1, \ldots, Q_n$. Let $\epsilon(t_1), \ldots, \epsilon(t_n)$ be evaluation trees for $Q_1, \ldots, Q_n$ respectively. Let E_C be the translation of C in which all identifiers corresponding to existential variables in the body of C (i.e., identifiers whose values are defined by **lrec**) are renamed to fresh distinct variables. Then the resulting evaluation tree $\epsilon(t)$ consists of a node labeled by E_C which has n descendants—the evaluation trees $\epsilon(t_1), \ldots, \epsilon(t_n)$.

Example 5 Consider the example from Section 2.1

c_0 : $p(X, Z) :- q(X, Y),\ r(Y, Z).$
c_1 : $q(X, X).$
c_2 : $r(X, X).$

$:- p(X, a).$

The clauses are labeled c_0, c_1, c_2. We have seen that the translation is

$$e_0: \quad F_p\ (i_x, i_z)\ \&\ \left(\begin{array}{lrcl} \textbf{lrec} & s_x, e_{y1} & = & F_q\ (i_x, e_y) \\ \textbf{and} & e_{y2}, s_z & = & F_r\ (e_y, i_z) \\ \textbf{and} & e_y & = & eqc\ e_{y1}\ e_{y2} \end{array}\right) = (s_x, s_z)$$

$$e_1: \ F_q\ (i_{x1}, i_{x2})\ \&\ (s_x = eqc\ i_{x1}\ i_{x2}) = (s_x, s_x)$$

$$e_2: \ F_r\ (i_{x1}, i_{x2})\ \&\ (s_x = eqc\ i_{x1}\ i_{x2}) = (s_x, s_x)$$

the translation of the goal is

$$\textbf{lrec}\ (e_x, a) = F_p\ (e_x, a)$$

A proof tree for this program is shown in Figure 3.1 (a), and the corresponding evaluation tree is shown in Figure 3.1 (b).

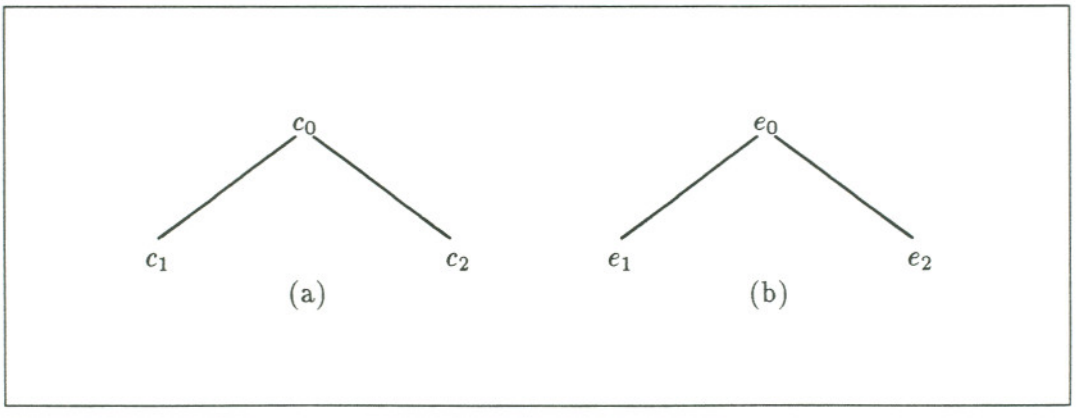

Figure 3.1: Proof and evaluation trees

It is important to emphasize that all identifiers in an evaluation tree $\epsilon(t)$ corresponding to existential variables are distinct. Given a derivation tree t, let $Ex(t)$ be the set of all existential variables in t (more precisely, all existential variables in labels of nodes of t). For each variable $v \in Ex(t)$ define two new distinct variables v_i and v_f called

pre-variable for v and *post-variable for* v respectively. We assume there are functions *pre* and *post* such that $v_i = pre(v)$ and $v_f = post(v)$. Now we define the concept of an *approximation tree*. Given a derivation tree t, the approximation tree $\alpha(t)$ is obtained from the evaluation tree $\epsilon(t)$ by replacing the label of every node in $\epsilon(t)$ by a new label defined as follows: let E be the the equation that labels the node in $\epsilon(t)$; the new equation E' is defined by

$$E' = \{l_1 = r_1 \mid e \equiv (l = r),\ e \in E,\ l_1 = \tau_l(l),\ r_1 = \tau_r(r)\}$$

$$\tau_l = \{post(v)/v \mid v \in Ex(t)\} \quad \tau_r = \{pre(v)/v \mid v \in Ex(t)\}$$

The same transformation is also applied to the goal. Intuitively, this transformations changes **lrec** equations to non-recursive equations; given an equation **lrec** $p = M$, it is changed to $\tau_l(p) = \tau_r(M)$.

Example 6 Consider the previous example. The translation of the clause c_0 is changed to

$$a_0 : \quad F_p\ (i_x, i_z)\ \&\ \left(\begin{array}{lrcl} \textbf{lrec} & s_x, e_{y1} & = & F_q\ (i_x, y_{pre}) \\ \textbf{and} & e_{y2}, s_z & = & F_r\ (y_{pre}, i_z) \\ \textbf{and} & y_{post} & = & eqc\ e_{y1}\ e_{y2} \end{array}\right) = (s_x, s_z)$$

The approximation tree is obtained from the evaluation tree in Figure 3.1 (b) by replacing the root node e_0 with a node labeled a_0. The remaining nodes are unchanged since there are no existential variables in the labeled equations. The translation of the goal is changed to

$$(x_{post}, a) = F_p\ (x_{pre}, a)$$

We can see that the occurrences of identifiers e_y and e_x, whose bindings are recursively defined, have been changed to occurrences of y_{post}, y_{pre}, x_{post}, x_{pre}. More precisely,

the occurrences of e_y and e_x on the left hand side (of equations) have been changed to occurrences of y_{post} and x_{post}, respectively. The occurrences of e_y and e_x on the right hand side have been changed to occurrences of y_{pre} and x_{pre}, respectively.

Let ρ be an environment containing bindings for pre-existential variables. We will call such environments *pre-environments.* Similarly, we will call environments containing bindings for post-existential variables *post-environments.* Given an answer substitution σ together with a proof tree t, the corresponding *answer environment* is defined as follows

$$\alpha = \{x \mapsto t \mid v \in Ex(t),\ x = ev(v),\ t = \phi(\sigma(v))\}$$

where ϕ is a function that replaces unbound variables with SW

$\phi\ Var(x) = SW$
$\phi\ c(t_1, \ldots, t_n) = c(\phi(t_1), \ldots, \phi(t_n))$

So an answer environment is obtained from an answer substitution by replacing unbound variables with SW.

Given a logic program P with a goal G, let σ be an answer substitution for $P \cup G$ and t the corresponding proof tree. Let E'_G be the translation of the goal in which occurrences of existential variables on the right hand sides of equations are replaced by corresponding pre-variables and occurrences on the left hand side by post-variables. All pre-existential variables are free in the translation; given a pre-environment ρ_0 one can evaluate the translation of the goal to obtain a post-environment ρ_f, i.e., $\rho_f = \mathcal{D}[\![E'_G]\!]\rho_0$. The computation of the meaning of the goal expression follows the structure of the approximation tree since the predicate functions are in general non-deterministic. At each step of evaluation of an application of a predicate function, we use the equation which is the label of the corresponding node of the approximation tree. Pre- and post-environments are used for computation of approximations of recursively defined environments comprising

values of identifiers defined by **lrec**.

Example 7 Consider Example 6. Let $\rho_0 = \{x_{pre} \mapsto SW,\ y_{pre} \mapsto SW\}$. We can compute $\mathcal{D}[\![(x_{post}, a) = F_p\ (x_{pre}, a)]\!]\rho_0$ giving us the bindings for x_{post} and y_{post}. We get $\rho_f = \{x_{post} \mapsto SW,\ y_{post} \mapsto a\}$.

Lemma 3 Assume there is an answer substitution σ for a logic program P with a goal G. Let α be the corresponding answer environment and ρ_0 be a pre-environment such that $\rho_0 \preceq \alpha$. Then $\rho_f \preceq \alpha$ where ρ_f is the corresponding post-environment. In particular, $\rho_f \neq \mathit{Fail}$.

Proof The proof is by induction on the structure of the proof tree.

- Base case—the proof tree consists of a fact $q(t_1, \ldots, t_{n_q})$. Let F_q be the translation of the fact. Since $\rho_0 \preceq \alpha$, some of the bindings for pre-existential variables in ρ_0 might be SW when they have ground bindings in α. Since the values in α satisfy all matches, so will the values in ρ_0 because $\rho_0 \preceq \alpha$. It follows that the result is not *Fail*. As a matter of fact, in this case $\rho_f = \alpha$ since if a binding for a variable is bound during unification of the fact q with the literal in the goal, this will be reflected in the reduction sequence by a binding occurring in the output of the application of F_q in the translation of the goal.

- Induction case—the proof tree corresponds to a clause. Let the goal G be $q(p_1, \ldots, p_{n_q})$. Let the clause be in the form

$$q(t_1, \ldots, t_{n_q}) :- r_1(t_{11}, \ldots, t_{1n_{r_1}}), \ldots, r_m(t_{m1}, \ldots, t_{mn_{r_m}}).$$

  Its translation is

$$F_q\ (it_1, \ldots, it_{n_q})\ \&\ \left(\begin{array}{l}
\textbf{lrec}\ (st_{11}, \ldots, st_{1n_{r_1}}) = F_{r_1}\ (it_{11}, \ldots, it_{1n_{r_1}}) \\
\qquad\qquad \vdots \\
\textbf{and}\ (st_{m1}, \ldots, st_{mn_{r_m}}) = F_{r_m}\ (it_{m1}, \ldots, it_{mn_{r_m}}) \\
\textbf{and}\ i_{v_1} = eqc_{k_1}\ i_{v_1 1}\ \ldots\ i_{v_1 k_1} \\
\qquad\qquad \vdots \\
\textbf{and}\ i_{v_l} = eqc_{k_l}\ i_{v_l 1}\ \ldots\ i_{v_l k_l} \\
\textbf{and}\ s_{u_1} = eqc_{r_1}\ s_{u_1 1}\ \ldots\ s_{u_1 r_1} \\
\qquad\qquad \vdots \\
\textbf{and}\ s_{u_j} = eqc_{r_j}\ s_{u_j 1}\ \ldots\ s_{u_j r_j} \\
\textbf{and}\ \ldots, y_{11}, \ldots, y_{1e_1}, \ldots = F_{r_{n_1}}\ \ldots, y_{i1}, \ldots, y_{ie_1}, \ldots \\
\textbf{and}\ y_{f1} = eqc\ y_{11}\ \ldots\ y_{1e_1} \\
\qquad\qquad \vdots
\end{array}\right)$$

$$= st_1, \ldots, st_{n_q}$$

Let $F_q(p_1, \ldots, p_{n_q})$ be the application of F_q in the translation of the goal. Since $\rho_0 \preceq \alpha$, it follows that $\rho_0((p_1, \ldots, p_{n_q})) \preceq \alpha((p_1, \ldots, p_{n_q}))$. The match in the application of F_q will succeed and $\forall i\ 1 \leq i \leq l,\ \rho_0(i_{v_i}) \preceq \alpha(i_{v_i})$, where $v_i,\ 1 \leq i \leq l$, are the variables in the head $q(t_1, \ldots, t_{n_q})$ and $i_{v_j},\ 1 \leq j \leq l$, are corresponding inherited variables. The resulting bindings for the variables i_{v_j} under ρ_0 are not greater than their bindings under α. This fact is also true for pre-existential variables $\forall j\ \rho_0(y_{ij}) \preceq \alpha(y_{ij})$. It follows

$$\forall j\ 1 \leq j \leq m,\ \rho_0((it_1, \ldots, it_{n_{r_j}})) \preceq \alpha((it_1, \ldots, it_{n_{r_j}}))$$

By the inductive hypothesis,

$$\forall j\ 1 \leq j \leq m,\ \rho_f((st_1, \ldots, st_{n_{r_j}})) \preceq \alpha((st_1, \ldots, st_{n_{r_j}}))$$

Finally,

$$\forall j\ 1 \leq j \leq l,\ \rho_f(s_{v_j}) \preceq \alpha(s_{v_j})$$

and $\rho_f \preceq \alpha$.

Intuitively, if we start the computation from an environment below the answer, there have to be some variables that are bound in the answer, but have value SW in the starting environment. The process of reduction mirrors the structure of the proof tree. For each node in the proof tree, which labels a clause used in the derivation, the corresponding equation is chosen in the reduction sequence. This equation will match since the values for variables are below the answer (for which the match still succeeds). After all applications of functions corresponding to predicates have been reduced (this point in the reduction sequence corresponds to a traversal of the entire proof tree), the result is ρ_f. It reflects new bindings found for some of the identifiers. Note that these bindings are not propagated and shared since in evaluating them we are still using the initial environment. In essence what we have done is evaluated right hand sides of recursive equations to get a new approximation for the recursive environment. Because of possible lack of sharing, ρ_f might be below σ since bindings for some identifiers might be SW in ρ_f when they actually have some ground bindings in σ. However ρ_f can never be above σ.

Example 7 provides a nice illustration. The final environment ρ_f binds variable y_{post} to a. Note that in the equation for F_p in the approximation tree (Example 6), y_{pre} occurs on the right hand sides of equations. The binding for y_{pre} is SW, regardless of the fact that the value computed for y_{post} is a. The binding for y_{pre} is used for computation of expressions on the right hand side and not the binding for y_{post}. In the translation, the binding for y is recursively defined and we have the same identifier appearing in place of y_{post} and y_{pre}.

Lemma 4 Let P be a logic program with a goal G. Let σ be an answer substitution for it and let α be the corresponding answer environment. Let ρ_0 be a pre-environment such that $\rho_0 \prec \alpha$ and let ρ_f be the corresponding post-environment obtained by evaluation of values of post-existential variables. Then $\rho_0 \prec \rho_f$.

Proof The proof is by induction on the structure of the proof tree.

- Base case—the proof tree consists of a fact $q(t_1, \ldots, t_n)$. Since $\rho_0 \prec \alpha$, there has to be an existential variable y such that $\rho_0(y_i) \prec \alpha(y)$, $y_i = pre(y)$. Looking at the proof of the logic program, the binding for y had to be obtained from one of the occurrences of y in $q(t_1, \ldots, t_n)$. But the consequence is that the binding will be reflected in the value of the post-variable y_f for y; it follows $\rho_0(y_i) \prec \alpha(y_f)$.

- Induction case—the root of the proof tree is labeled by a clause

$$q(t_1, \ldots, t_{n_q}) :- r_1(t_{11}, \ldots, t_{1n_{r_1}}), \ldots, r_m(t_{m1}, \ldots, t_{mn_{r_m}}).$$

  Consider the conjuncts r_j in the body of the clause as goals by themselves; let σ_j be the answer substitutions for them and let α_j be the corresponding answer environments. Because each literal q_j is considered as a goal, if we consider their translations the inherited variables play the role of pre-existential variables and synthesized the role of post-variables. Let ρ_{0j} be the pre-environments for each conjunct r_j. There are two cases:

  - $\exists k\ \rho_{0k} \prec \alpha_k$. By inductive hypothesis, $\rho_{0k} \prec \rho_{fk}$ and since it is always the case that $\rho_{0i} \preceq \rho_{fi}$ the claim follows.
  - $\forall k\ \rho_{0k} = \alpha_k$. In words, all variables in the translations of literals in the body have the same binding as in the answer environment. There has to be a variable v in the goal such that $\rho_0(v) \prec \alpha(v)$ and the binding for this variable had to be obtained from a match with one of t_i, $1 \leq n_q$. The corresponding situation on the logic programming side is that there exists a variable in the goal bound during the unification with the head of the clause and not during the resolution of literals in the body.

∎

Intuitively, if we start computation from an environment strictly below the answer, the final environment in the reduction sequence will be strictly above the starting environment, reflecting the fact that even though shared bindings might not be propagated, *some* identifiers will be bound.

Theorem 5 Let P be a logic program with a goal G; let σ be an answer substitution for it and let α be the answer environment. Let E_G and E_P be the translations of the goal and the program, respectively. Then $\mathcal{D}\ [\![E_G]\!]\ \epsilon = \alpha$ where ϵ is the empty environment.

Proof There is a sequence of environments $\rho_0, \rho_1, \ldots, \rho_n, \ldots$ such that every environment in the sequence is the post-environment for the preceding member considered as a pre-environment. The initial environment ρ_0 maps all identifiers to SW. By Lemma 3, $\forall j\ \rho_j \preceq \alpha$. By Lemma 4, it follows that $\rho_0, \ldots, \rho_j$ is a strictly increasing sequence as long as $\rho_j \prec \alpha$. It follows that for a finite j, $\rho_j = \alpha$ since there can be no infinite strictly increasing sequence bounded above by α (α is finite). ∎

Example 8 Consider again Examples 6 and 7. The sequence of approximating environments is

$$\rho_0 = \{x_{pre} \mapsto SW,\ y_{pre} \mapsto SW\},$$
$$\rho_1 = \{x_{pre} \mapsto SW,\ y_{pre} \mapsto a\},$$
$$\rho_2 = \{x_{pre} \mapsto a,\ y_{pre} \mapsto a\}$$

The fixpoint ρ_2 is reached after two iterations. We have seen in the Example 7 that the value of variable x is equal to the value of y. After the first iteration, the value of y, i.e., the value of y_{post}, is a, but the value of x_{post} has been computed using the initial approximation SW for y_{pre}. The consequence is that after the first iteration the value of x_{post} is SW. It is only after the second iteration that the binding for y_{pre}, i.e., constant a is propagated to x_{post}.

3.3 The equality function

Let us consider now the equality function *eqc* in more detail. We repeat its definition from Section 2.1.

$eqc\ x\ y =$ **if** $x = y$ **then** x **else** *Fail*

This definition is in essence syntactic. In this chapter we are interested in the (denotational) semantics for the translation. We need to give a semantic definition of *eqc* that accounts for all domain elements. First, it is clear that if both arguments of *eqc* are $\perp$, then the result is also $\perp$. This information gives us the first equation in its definition:

$$eqc\ \perp\ \perp = \perp$$

The next equation specifies that *eqc* is strict in its first argument, since it has to be evaluated in order to perform the matching:

$$eqc\ \perp\ x = \perp$$

We might expect a similar equation for the second argument but there is a difference in this case. Clearly, we need to evaluate the second argument as well as first in order to perform the matching, except in one case, when a failure has been detected during the evaluation of the first argument. Obviously, it is pointless to evaluate the second argument in this case since the final result will be failure. We have the following equations:

$$\begin{aligned} &eqc\ Fail\ x = Fail \\ &eqc\ x\ \perp = \perp && x \neq Fail \\ &eqc\ x\ Fail = Fail && x \neq \perp \end{aligned}$$

Note that we could have chosen to evaluate arguments in parallel, by adding the equation $eqc\ \perp\ Fail = Fail$. However, we would need a mechanism for evaluating the arguments in parallel, similar to the well known example of "parallel OR". The implementation of such functions can be problematic and we choose not to adopt it. The next two equations specify the result in case any of the arguments are unbound values:

$$eqc\ SW\ x = x$$
$$eqc\ x\ SW = x$$

Clearly, if any of the arguments is an unbound value, the result is equal to the other argument. Finally, we need to specify the result in case the arguments are ground:

$$\begin{array}{l} eqc\ c(x_1,\ldots,x_n)\ d(y_1,\ldots,y_n) = \\ \quad (c = d) \rightarrow \quad eqc\ x_1\ y_1 = \bot \rightarrow \bot \\ \qquad\qquad\quad\ \ eqc\ x_1\ y_1 = Fail \rightarrow Fail \\ \qquad\qquad\quad\ \ \vdots \\ \qquad\qquad\quad\ \ eqc\ x_n\ y_n = \bot \rightarrow \bot \\ \qquad\qquad\quad\ \ eqc\ x_n\ y_n = Fail \rightarrow Fail \\ \qquad\qquad\quad\ \ c(eqc\ x_1\ y_1,\ldots,eqc\ x_n\ y_n) \\ \quad Fail \end{array}$$

Here is the complete definition of *eqc*:

$$\begin{array}{ll} eqc\ \bot\ x = \bot & \\ eqc\ Fail\ x = Fail & \\ eqc\ x\ \bot = \bot & x \neq Fail \\ eqc\ x\ Fail = Fail & x \neq \bot \\ eqc\ SW\ x = x & \\ eqc\ x\ SW = x & \end{array}$$

$$
\begin{array}{l}
eqc\ c(x_1,\ldots,x_n)\ d(y_1,\ldots,y_n) = \\
\quad (c = d) \rightarrow \quad eqc\ x_1\ y_1 = \bot \rightarrow \bot \\
\qquad\qquad\qquad\quad eqc\ x_1\ y_1 = Fail \rightarrow Fail \\
\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\quad eqc\ x_n\ y_n = \bot \rightarrow \bot \\
\qquad\qquad\qquad\quad eqc\ x_n\ y_n = Fail \rightarrow Fail \\
\qquad\qquad\qquad\quad c(eqc\ x_1\ y_1,\ldots,eqc\ x_n\ y_n) \\
Fail
\end{array}
$$

3.4 Interpretation versus compilation

There have been numerous implementations of logic languages. Among the many proposals, the implementations of logic languages in functional languages [Car84] or functional-like languages [Fel85] or treatments of operational and denotational semantics [JM84] are particularly interesting in their relationship to our approach. The common point in all these approaches is that variables are *interpreted*, i.e., there are distinguished run-time representations of variables in programs. Of course, the particular representations adopted vary:

- bindings for variables can be recorded in association lists and different occurrences of a variable in different instances of a clause are distinguished by numerical indices [Car84].
- variables can be represented by assignable **ref** cells [Fel85].
- representations of variables can be defined in the definition of a data type of terms as unary constructors and numerical indices are used for renaming of variables to prevent name clashes [JM84].

A closer analysis of the treatment of variables in these (and to author's knowledge, all the other approaches) reveals that distinguished representations of variables are needed in order to prevent name clashes between instances of the same variable in different copies of

the same clause used during resolution. The problem with clashes becomes even clearer if we consider the definition of resolution of Horn clauses with variables [Apt90]. At every step of resolution we have to make sure that a different instance of a clause is used in order to prevent name clashes. Apt calls different instances of clauses *variants*. This definition of resolution shows that the requirement for renaming of different instances of variables in logic programs is present at the declarative level and is not merely the byproduct of a particular implementation.

The treatment of variables in logic programs presented here should be contrasted with the treatment of variables in functional programs. The problem of name clashes is also present in λ-calculus. Renaming in λ-calculus is performed by α-conversion. The essential point is that in functional programs there is no need for distinguished representations of variables—they merely serve as placeholders for terms that can be viewed as *values*, i.e., terms with no variables. The occurrences of variables in functional programs are handled during compilation; the compiler produces code that substitutes a term for each occurrence of a variable in the body of a function definition. The term is synthesized from arguments of a particular application of a function. When there are several applications of the same function, the arguments of these applications are in general different and the occurrences of the same variable in the function definition are replaced by (different) argument terms.

Unification and pattern matching are two instances of a single problem—finding the (most general) substitution that makes two terms equal. Let us emphasize again that the difference between unification and pattern matching is that in unification both terms can have variables, while in pattern matching only one of them (the *pattern*) can. One might try to employ the strategy for handling variables in functional languages directly to logic programs. However, there is a problem with this approach in that both the argument term in the clause that is acting as the pattern and the target can have variables. The consequence is that information can flow in the other direction, from the pattern to the target term meaning some variables in the pattern might get identified with variables in the target term. In particular, different instances of the same variable in different

instances of a clause might inadvertently get identified. It is because of this fact that variables in each new instance of a clause have to be renamed.

In our approach, we have seen that the functional strategy has been adopted for all variables. The reason we have been able to do this is the fact that bindings for (existential) variables are defined recursively. The binding for any other variable in a logic program is either a constant or it depends on the binding of some existential variable. Note that it is crucial that these bindings are *values* that are passed as arguments to the functions corresponding to predicates.

The treatment of variables is one of the principal reasons we believe our approach is inherently different from previous attempts to implement logic languages in functional languages. In all of the other approaches, a logic program is translated to an *interpreter* in a functional language. The degree of interpretation in the resulting functional program can vary. For instance, the search for clauses matching a predicate can be replaced by calls to an appropriate function. It is possible to replace some calls to the unification routine by pattern matching since the shape of arguments of clauses is always known. In this case, the result of translation is a kind of "customized interpreter" that reflects the structure of the clauses in the logic program. But the end result is nonetheless always an *interpreter* because it interprets the variables in logic programs and does not treat them like variables in functional programs. In our approach, variables in logic programs are translated to variables in functional programs directly. This treatment of variables is the main reason we do not consider the result of translating a logic program to a functional program to be an interpreter but a functional program *equivalent* to the original logic program. They are equivalent in the sense that the solutions of the functional program and answer substitutions for the logic program correspond to each other.

3.5 Least fixpoints are not enough

The essential feature of the translation algorithm is that bindings for existential variables are defined recursively. We have seen that the bindings can be found as solutions to sets

of recursive equations and that the solutions are iterations to fixpoints, starting from SW. We believe that the equations produced by the translation present a nice example of equations for which we are not looking for least fixpoints (which are trivial). One might argue that in a way, fixpoints starting from SW are least since there is an ordering of terms $\preceq$ with respect to which SW is below all proper values. We needed this ordering in Section 3.2 to establish that the process of iteratively computing approximations for existential environment eventually terminates producing the result corresponding to the answer substitution. In other words, we might want to consider the set of values excluding $\perp$ as a domain itself. The crucial point is that $\perp$ cannot be excluded from the domain and we need *both* kinds of fixpoints:

- The least ones (starting from $\perp$) to give meanings to recursive functions corresponding to predicates that can be mutually recursive and need not terminate.
- Fixpoints starting from SW to define bindings for existential variables. They are expressed by the recursive construct **lrec**. In essence, least fixpoints are needed to handle nontermination; fixpoints from SW are needed for propagation of bindings for variables.

The solutions to the recursive equations produced by the translation cannot be found in the usual domain with only $\perp$ and proper values. The reason is because the only choice for starting the iterative computation for the fixpoint is $\perp$ and that does not work since matching is strict and each approximation is equal to $\perp$. Of course there are are infinitely many other choices to start the iteration, namely the other elements of the domain. Moreover, one of them will trivially produce the right answer since if we start from the answer the recursive equation is immediately satisfied. But this information does not tell us anything since we do not know the answer in advance.

Chapter 4

Operational semantics

In Section 3.2 we have seen that the solutions of the equations obtained by translation of logic programs can be computed as certain fixpoints. We can implement computation of these fixpoints in a straightforward way, by iterative computation. In essence, the approach is to find a solution of the equation

$$\rho_e = F\ \rho_e$$

where ρ_e is the existential environment consisting of all existential variables and F is the appropriate function obtained from the translation. The initial approximation ρ_0 for ρ_e binds all variables to SW. Assuming a solution σ exists, i.e., the computation does not diverge nor ends in failure, we can obtain σ by iterating F on ρ_0, i.e., $\sigma = \lim_{i\to\infty} F^i\rho_0$. However, even though perfectly valid, this strategy would not be an efficient way to compute solutions because of a large amount of recomputation. We want a better way to find the answer. The subsequent sections are intended to show how to compute solutions efficiently.

4.1 Graph reduction

Graph reduction is a well known technique for implementation of lazy functional languages [Joh87b, PJ87]. It was first proposed by Wadsworth [Wad71] for implementing λ-calculus. The main idea behind graph reduction is that program expressions are represented by (directed) *graphs*. The advantage of using graphs, as opposed to trees, is that it is possible to implement *sharing*.

A graph is a pair $\langle V, E \rangle$ where

- V is the set of *vertices* or *nodes*
- E is the set of directed *edges*; each edge is a pair (n_0, n_1) where n_0 is the *source* node and n_1 is the *destination* node

To each expression in the language a graph is assigned. We also define an auxiliary function r that assigns a designated node, called the *root* node, to each graph produced by translating an expression. Nodes in graphs belong to one of following classes:

- CONSTR c—a node corresponding to a constructor c
- APP—nodes corresponding to function applications
- PAIR_k—a node corresponding to a k-tuple; it has k children representing the components of the tuple
- FUN f—a node corresponding to a function; in the rest of the presentation FUN will be omitted since it will be always possible to determine from the graph that a node is a FUN node
- EQC—a node corresponding to equality test

We simultaneously define two functions: $\mathcal{G}$, that assigns graphs to expressions; r, assigning nodes to graphs.

$\mathcal{G}\ [\![x]\!]\ \rho_e = \langle \rho_e(x), \{\} \rangle$

$r(G) = \rho_e(x)$

$\mathcal{G}\ [\![c(x_1, \ldots, x_k)]\!]\ \rho_e = \langle \{m : \text{CONSTR } c\} \cup V_1 \cup \ldots \cup V_k, E_{x1} \cup \ldots \cup E_{xk} \rangle$ where

$\forall i\ 1 \leq i \leq k \quad X_i = \langle V_i, E_i \rangle = \mathcal{G}\ [\![x_i]\!]\ \rho_e,\ E_{xi} = E_i \cup \{(m, r(X_i))\}$

$r(G) = m$

$\mathcal{G}\ [\![(x_1, \ldots, x_k)]\!]\ \rho_e = \langle \{m : \text{PAIR}_k\ c\} \cup V_1 \cup \ldots \cup V_k, E_{x1} \cup \ldots \cup E_{xk} \rangle$ where

$\forall i\ 1 \leq i \leq k\ \ X_i = \langle V_i, E_i \rangle = \mathcal{G}\ [\![x_i]\!]\ \rho_e,\ \ E_{xi} = E_i \cup \{(m, r(X_i))\}$

$r(G) = m$

$\mathcal{G}\ [\![eqc\ x_1\ x_2]\!]\ \rho_e = \langle \{n : \text{EQC}\} \cup V_1 \cup V_2, E_{x1} \cup E_{x2} \rangle$ where

$\forall i\ 1 \leq i \leq 2\ X_i = \langle V_i, E_i \rangle = \mathcal{G}\ [\![x_i]\!]\ \rho_e,\ \ E_{xi} = E_i \cup \{(n, r(X_i))\}$

$r(G) = n$

$\mathcal{G}\ [\![f\ x]\!]\ \rho_e = \langle \{n : \text{APP}, n_f : \text{FUN}\ f\} \cup V_x, E_x \cup \{(n, n_f), (n, r(G_x))\}$ where

$G_x = \langle V_x, E_x \rangle = \mathcal{G}\ [\![x]\!]\ \rho_e$

$r(G) = n$

$\mathcal{G}\ [\![\textbf{lrec}\ x = E]\!]\ \rho_e = \mathcal{G}\ [\![E]\!]\ \rho_e[x \mapsto n]$

$r(G) = r(E) = n$

In the definition above, $r(G) = n$ means that n is the root node of the graph produced by $\mathcal{G}$ by translating the argument expression. Also, all nodes that are explicitly mentioned are assumed to be new nodes distinct from other nodes. The translation function $\mathcal{G}$ takes an environment argument that is used for keeping track of nodes assigned to identifiers. Note that applications of *eqc* are represented by EQC nodes. In essence EQC is a binary constructor. There is no equation for **let** expressions; it is not necessary since programs with **let** expressions can be transformed to programs without them

$$\textbf{let}\ x = M\ \textbf{in}\ N \equiv (\lambda x.N)\ M$$

The pattern matching on pairs in **let** expressions can also be removed

$$\textbf{let}\ x_1, \ldots, x_n = M\ \textbf{in}\ N \equiv \textbf{let}\ p = M\ \textbf{in}\ N_1$$

where $N_1 = \sigma(N)$, $\sigma = \{\pi_1\ p/x_1, \ldots, \pi_n\ p/x_n\}$. Function π_i is the i-th projection, i.e.,

$$\pi_i\ (x_1, \ldots, x_i, \ldots, x_n) = x_i$$

It is important to note that the graphs produced from expressions can be cyclic; in particular the translation of an **lrec** expression introduces a cycle into the graph. The process of reduction is represented by transformation of graphs. The most important rule of reduction is the β-reduction rule; given a redex $(\lambda x.M)\ N$, the graph for the result is constructed by creating a copy of the graph for M in which all (free) occurrences of x are replaced by the graph for N. Translations of constructors, tuples, *eqc*-checks and applications are illustrated in Figure 4.1 (a), (b), (c) and (d), respectively.

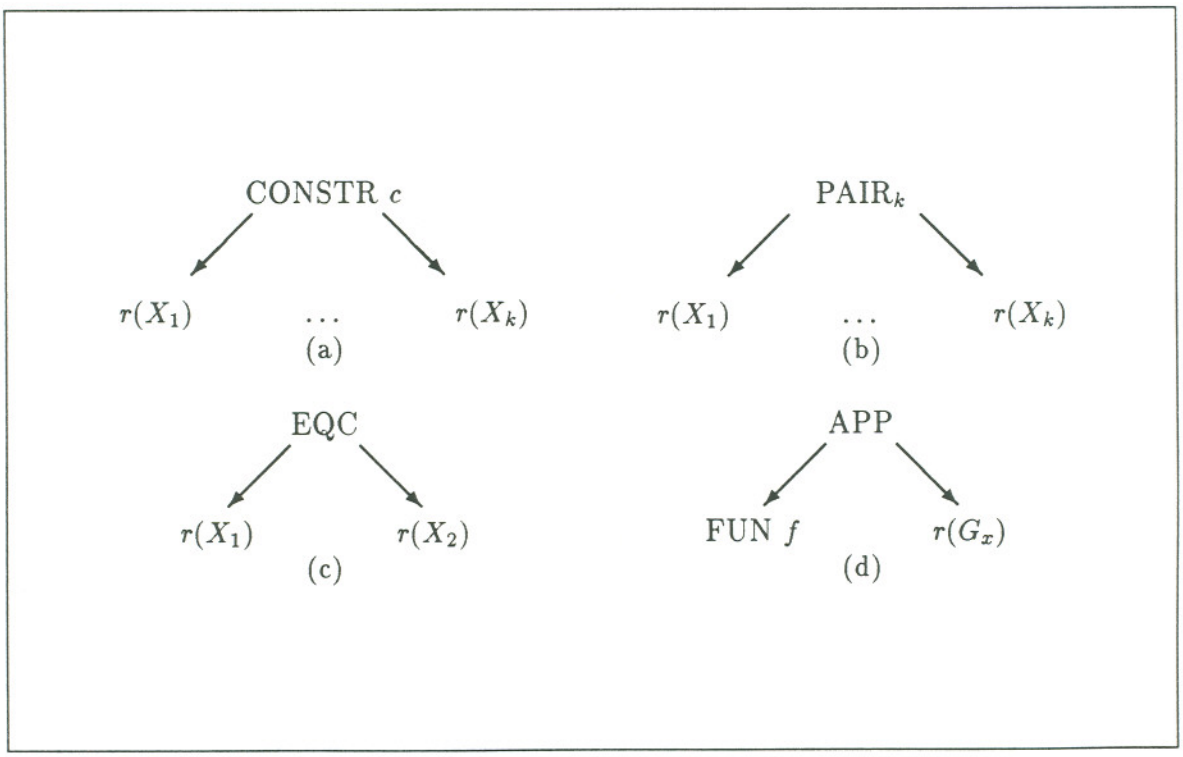

Figure 4.1: Graph translations

Example 9 Consider the expression

$$\textbf{lrec}\ x = eqc\ c_0\ (F_p\ x)\ \textbf{in}\ x$$

The corresponding graph is shown in Figure 4.2. The root of the graph is the EQC node.

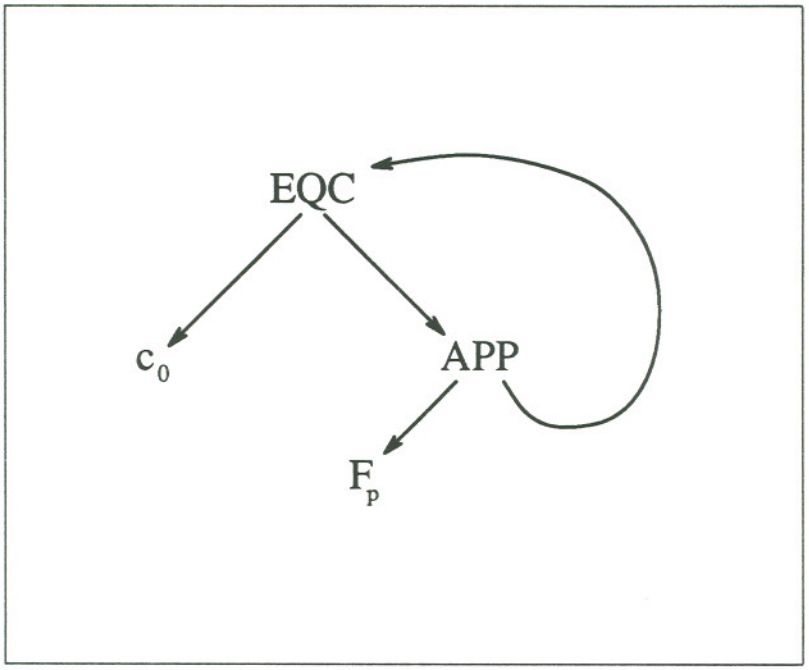

Figure 4.2: An example of a graph

4.2 Traversal of cyclic graphs

We consider first a subset of logic programs, called Datalog programs [MW88]. The restriction is that the only constructors allowed are nullary, i.e., constants. We pay particular attention to *eqc*-nodes, since it is possible to reduce expressions in the translation to a form in which there are only applications of the equality function *eqc*. The translation algorithm is slightly changed. Since we are considering only Datalog terms, each term is either a variable or a constant. Variables are handled the same way as before; for constants, pattern matching is replaced by applications of *eqc*. For instance if a constant

a appears in the head of a clause

$$p(\ldots, a, \ldots) :- \ldots$$

its translation is

$$F_p\ (\ldots, x, \ldots) \ldots\ =\ (\ldots, eqc\ x\ a, \ldots)$$

If a constant appears in the body of a clause, it is simply passed as an argument in the corresponding applications of F_q where q is the literal in the body in which the constant appears. We can view this in an informal way as a kind of "lazy" pattern matching. Consider a goal

$$:- q_1(t_{11}, \ldots, t_{1n_1}),\ \ldots,\ q_m(t_{m1}, \ldots, t_{mn_m}).$$

We have seen in Section 3.1.4 that its translation is

lrec $\ldots, y_{11}, \ldots, y_{1e_1}, \ldots = F_{q_1}\ (\ldots, y_1, \ldots, y_1, \ldots)$

$\vdots$

and $y_1 = eqc_{e_1}\ y_{11}\ \ldots\ y_{1e_1}$

$\vdots$

and $y_n = eqc_{e_n}\ y_{11}\ \ldots\ y_{1e_n}$

in

$y_1, \ldots, y_n$

We can perform reductions of applications of the F_{q_i}. For any derivation tree the goal expression can be reduced to a form in which there are no applications of predicate functions F_r. We are going to call expressions in this form to be in *eqc-normal form.*

Proposition 6 (*eqc*-normal forms) Assume a logic program P with a goal G. Let E be their translation using the algorithm for translation of Datalog programs. Then for any finite derivation tree, either successful or failed, the corresponding goal expression can be reduced to *eqc*-normal form.

Proof The proof is by induction on the structure of the derivation. Since the derivation is finite, we can always reduce applications of F_p for every literal p in the derivation. ∎

Example 10 Consider a goal $:-p(Y, b, Y)$ and a fact $p(a, X, X)$. The translation of the goal is

$$\begin{array}{l}\textbf{lrec } y_1, b, y_2 = F_p\ (y, b, y) \\ \textbf{and } y = eqc\ y_1\ y_2 \\ \textbf{in } y\end{array}$$

The translation of the fact is

$$\begin{array}{lll} F_p\ (in_1, in_2, in_3) = & \textbf{let} & x = eqc\ in_2\ in_3\ \textbf{in} \\ & & (eqc\ a\ in_1, x, x)\end{array}$$

After reduction of the application of F_p in the goal we get

$$\begin{array}{l}\textbf{lrec } y_1, b, y_2 = \textbf{let } x = eqc\ b\ y\ \textbf{in } (eqc\ a\ y, x, x) \\ \textbf{and } y = eqc\ y_1\ y_2 \\ \textbf{in } y\end{array}$$

This expression is in *eqc*-normal form. Intuitively, the resulting expression corresponds to an aggregate of all equality tests which have to be satisfied in the derivation. The bindings for existential variables are defined recursively, by **lrec**.

4.3 Traversal algorithm

The solutions of equations in *eqc*-normal form are of course the solutions of original equations. The solutions consist of bindings for existential variables; the fact that the equations are in *eqc*-normal form emphasizes that the only possible (non trivial) bindings are manifest constants appearing in expressions defining the bindings. In other words,

suppose we have an **lrec** expression in *eqc*-normal form, **lrec** $x = N$. Then if any subexpression of N is a constant, the solution x is either that constant or failure, assuming the computation does not diverge. Failure occurs when different constants appear in N. This observation forms a basis for an algorithm that searches for a solution by traversing the graph representations of expressions and looking for constants. Since the graph is, in general, cyclic, we need to employ a mechanism for traversing cyclic structures. A simple marking of nodes will suffice. The traversal algorithm is defined as follows:

```
procedure traverse(root);
if marked(root) then return;
if root = const(a) then
    if result = SW then result := const(a)
    else  if result = const(b) then
              if a = b then return else Fail
else if root = EQC(g1,g2) then mark(root); traverse(g1); traverse(g2);

procedure tr(graph);
    result := SW;
    traverse(graph);
    return  result;
```

The algorithm is imperative; *result* is a global variable which is assigned SW initially. Intuitively, SW indicates that no constant has been found during the traversal. Whenever a constant is found, *result* is assigned the value of the constant if its previous value was SW. If *result* is already set to some constant, then a check is performed between the value of *result* and the constant upon consideration. If they are different, the result of the whole traversal is failure; otherwise the constants are equal and the computation proceeds. If an EQC node is encountered during the traversal, it is first marked; then its descendants are traversed. The very first thing in the algorithm is to check if the node

under consideration is marked. In case it is, we return without performing anything, to prevent infinite looping due to cycles in the graph.

4.4 Correctness of the traversal algorithm

The expressions in *eqc*-normal forms can be viewed in two ways:

- As sets of recursive equations; EQC terms are interpreted by equality functions.
- As cyclic graphs; EQC terms are interpreted as nodes in a graph.

The solutions of recursive equations can be computed as fixpoints in iterative way. On the other hand, viewing the expressions as graphs, solutions can be found in a single pass, by traversal of the graph. Note that the reduction to *eqc*-normal form can be done during the traversal, so a separate pass is not needed. Intuitively, the traversal of the graph agrees well with our understanding of unification; by traversing a graph corresponding to an existential variable, we expect all constants or other variables which are unified with it during the process of computation to be "reachable" from the root. The structure of the graph reflects unifications with constants as well as sharing of unified variables. It should be mentioned that whenever the term "unification" is used, it refers to unifications in the logic program under consideration.

This relationship will be made more precise by the next theorem. First let us introduce some notation to clarify the presentation.

Assume we are given a logic program P with a goal G and an answer substitution σ for it. Let E_G be the translation of G and let Γ_G be the graph assigned to (Section 4.1) the *eqc*-normal form of E_G. For any variable v in G, there is a corresponding variable y_v in E_G; we have seen in Section 2.2.2 that $y_v = ev(v)$. Let us identify the node in Γ_G corresponding to y_v and call it the *designated node* for y_v. We will use notation $dn(y_v)$ for the designated node for v (i.e., for y_v, but there is no confusion since there is one-to-one correspondence between them). More precisely, recall that variables in the

goal are translated to **lrec** expressions. Let

$$y_v = E$$

be the definition of the binding for y_v in E_G. The designated node $dn(v)$ is defined to be $r(\mathcal{G}(E))$, i.e., the root node of the (cyclical) graph representing the binding for y_v.

Theorem 7 Given a logic program P with a goal G let σ be an answer substitution for it. Then for every variable v in the goal the following is true:

- if v has a ground binding c, then $tr(dn(v)) = c$
- if v is unbound in σ, then $tr(dn(v)) = SW$

Proof The proof is by induction on the structure of the proof tree. There are two cases:

- The proof tree is a fact. Let the goal literal G be $q(t_1, \ldots, t_n)$ and its translation

$$\begin{aligned} y_1, \ldots, y_n = \quad & \textbf{lrec } \ldots, y_{11}, \ldots, y_{1e_1}, \ldots = F_q\ (\ldots, y_1, \ldots, y_1, \ldots) \\ & \textbf{and } y_1 = eqc_{e_1}\ y_{11}\ \ldots\ y_{1e_1} \\ & \qquad\qquad \vdots \\ & \textbf{and } y_n = eqc_{e_n}\ y_{11}\ \ldots\ y_{1e_n} \\ & \textbf{in} \\ & \quad y_1, \ldots, y_n \end{aligned}$$

Consider the fact in the proof tree; it has the form

$$q(p_1, \ldots, p_n)$$

Its translation is

$$F_q\ (ip_1, \ldots, ip_n)\ \&\ \begin{pmatrix} s_{u_1} = eqc_{r_1}\ i_{u_1 1}\ \ldots\ i_{u_1 r_1} \\ \vdots \\ s_{u_j} = eqc_{r_j}\ i_{u_j 1}\ \ldots\ i_{u_j r_j} \end{pmatrix} = sp_1, \ldots, sp_n$$

Since we are dealing with translation of Datalog terms, the argument terms ip_j are (distinct) variables, and each result term sp_j is either

- s_{u_l}, if variable u_l is in the corresponding position
- *eqc* c_i ip_j, if a constant c_i appears in the corresponding position

Now consider an (existential) variable y and a term it is unified with during the unification of literal in the goal and the corresponding fact. There are two possibilities:

- The variable is unified with a constant c; then in the *eqc*-normal form of the translation y will be defined by

  $$\textbf{lrec}\ y = eqc\ y\ c$$

  It is clear that $tr(y) = c$ since the corresponding graph is cyclic and the traversal routine will mark the redex.

- The variable is unified with another variable, say w. Then it is clear that $tr(y) = tr(w)$. Whenever a variable is unified with another variable, the corresponding graph for this variable will be traversed as well. Clearly, this process cannot proceed indefinitely since there are only a finite number of variables in the goal. Eventually, if there is a constant binding for any of the variables that are unified together, it will be found; otherwise the value of *result* will be SW. The multiple occurrences of variables either in the fact or in the goal literal are covered by traversal since each argument of the corresponding EQC node will be traversed.

- The inductive case—the proof tree is a clause

  $$q(t_1, \ldots, t_{n_{q_i}}) :- r_1(t_{11}, \ldots, t_{1n_{r_1}}), \ldots, r_m(t_{m1}, \ldots, t_{mn_{r_m}}).$$

  Consider each of the conjuncts r_i in the body of the clause as goals by themselves; since there is an answer substitution for the whole program, there has to be an

answer substitution for each of r_i. The translations of r_i as goals are recursive **lrec** expressions; the corresponding graphs Gr_{r_i} are cyclic. The translation of the original goal literal q which the clause is resolved with is

$$
\begin{array}{l}
\textbf{lrec } \ldots, y_{11}, \ldots, y_{1e_1}, \ldots = F_q\ (\ldots, y_1, \ldots, y_1, \ldots) \\
\textbf{and } y_1 = eqc_{e_1}\ y_{11}\ \ldots\ y_{1e_1} \\
\qquad\qquad \vdots \\
\textbf{and } y_n = eqc_{e_n}\ y_{11}\ \ldots\ y_{1e_n} \\
\textbf{in} \\
\quad y_1, \ldots, y_n
\end{array}
$$

The translation of the clause is

$$
F_q\ (it_1, \ldots, it_{n_{q_i}})\ \&
\left(
\begin{array}{l}
\textbf{lrec } (st_{11}, \ldots, st_{1n_{r_1}}) = F_{r_1}\ (it_{11}, \ldots, it_{1n_{r_1}}) \\
\qquad\qquad \vdots \\
\textbf{and } (st_{m1}, \ldots, st_{mn_{r_m}}) = F_{r_m}\ (it_{m1}, \ldots, it_{mn_{r_m}}) \\
\textbf{and } i_{v_1} = eqc_{k_1}\ i_{v_1 1}\ \ldots\ i_{v_1 k_1} \\
\qquad\qquad \vdots \\
\textbf{and } i_{v_l} = eqc_{k_l}\ i_{v_l 1}\ \ldots\ i_{v_l k_l} \\
\textbf{and } s_{u_1} = eqc_{r_1}\ s_{u_1 1}\ \ldots\ s_{u_1 r_1} \\
\qquad\qquad \vdots \\
\textbf{and } s_{u_j} = eqc_{r_j}\ s_{u_j 1}\ \ldots\ s_{u_j r_j} \\
\textbf{and } \ldots, w_{11}, \ldots, w_{1e_1}, \ldots = F_{r_{n_1}}\ \ldots, w_1, \ldots, w_1, \ldots \\
\textbf{and } w_1 = eqc\ w_{11}\ \ldots\ w_{1e_1} \\
\qquad\qquad \vdots
\end{array}
\right)
= st_1, \ldots, st_{n_{q_i}}
$$

Note that we have to make sure that the identifiers for bindings of existential variables in the body of the clause are appropriately renamed so there are no name

clashes with identifiers for other existential variables. Reducing the application of F_q according to the translation of the clause, we get a new **lrec** expression; this expression has a corresponding graph, which we will call Gr_q. The essential point is that this graph can be built from the graphs Gr_{r_j} corresponding to conjuncts r_i viewed as subgoals. Each of Gr_{r_j} is cyclic. By the induction hypothesis, the theorem holds for each of Gr_{r_j}; we need to show that it follows that the result holds also for the composite graph Gr_q. Consider an arbitrary variable v in an arbitrary literal r_j in the body. We can identify a node corresponding to v in Gr_{r_j}; the *eqc*-normal form for the translation of r_j has the form

$$\textbf{lrec } y_v = E$$

The root node is $r(\mathcal{G}(E))$. There are edges in the graph that point back to the root—they correspond to occurrences of y_v in E. The graph is illustrated in Figure 4.3 (the root node is designated by r and a back edge to the root is designated e_v). The new graph Gr_q is composed of subgraphs Gr_{r_j}, with edges e_v to the root removed. These edges are not present in Gr_q since the binding for v is not recursively defined; it is defined by the equation

$$s_v = E\{i_v/y_v\}$$

where $E\{i_v/y_v\}$ is E with all free occurrences of y_v substituted by i_v. Recall that s_v is the synthesized variable for v and i_v is the inherited variable for v. There are several cases on how the composite graph Gr_q is built, depending on the number of occurrences of v in the head literal q:

- There is a single occurrence, say o; there are two subcases depending on what is in the goal G in the same position, i.e., what is $o(G)$ ($o(G)$ stands for *subt o G*):
  - $o(G)$ is a constant c; in this case Gr_q is equal to Gr_{r_j} in which all the back edges e_v to the root are replaced by edges to a constant node containing c. The result of traversal in this case clearly has to be c because the traversal

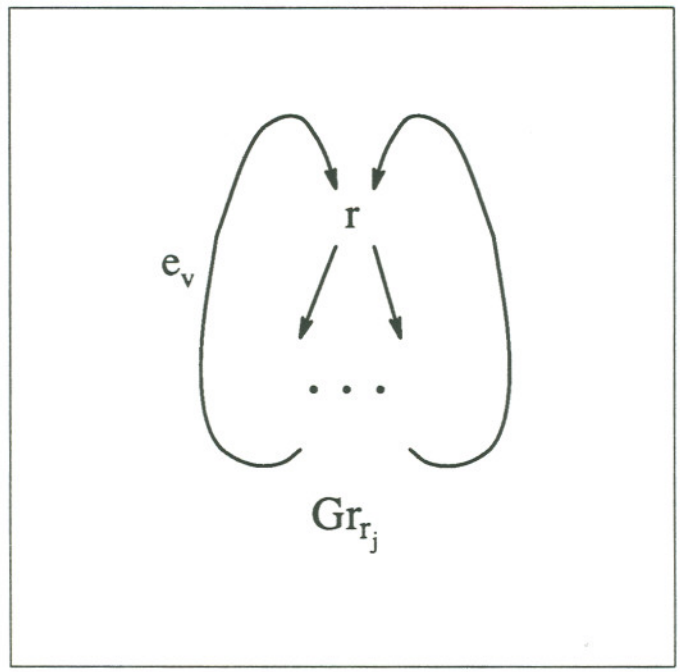

Figure 4.3: Cyclic graph for r_j

of Gr_{r_j} will produce either SW, in case v is unbound in r_j considered as a goal, or c in case v is bound to c. In both cases the traversal of the modified graph will also produce c.

* $o(G)$ is a variable w; there are two subcases depending on the number of occurrences of w in G:
  - there is a single occurrence; in this case Gr_q is identical to Gr_{r_j}
  - there is more than one occurrence; Gr_q is built from several component subgraphs corresponding to occurrences of w in G. A new EQC node is created which becomes the root of Gr_q; the edges from the root point to the roots of the component subgraphs corresponding to occurrences of w. Finally in the components graphs back edges to their roots are replaced by edges to the new root of Gr_q. Clearly, the result of traversal of the composite graph will be either a constant c, in case the traversal of one or more of component subgraphs is the same

constant , or SW, in case there are no constants in the component subgraphs. This situation is illustrated in Figure 4.4. Clearly, the

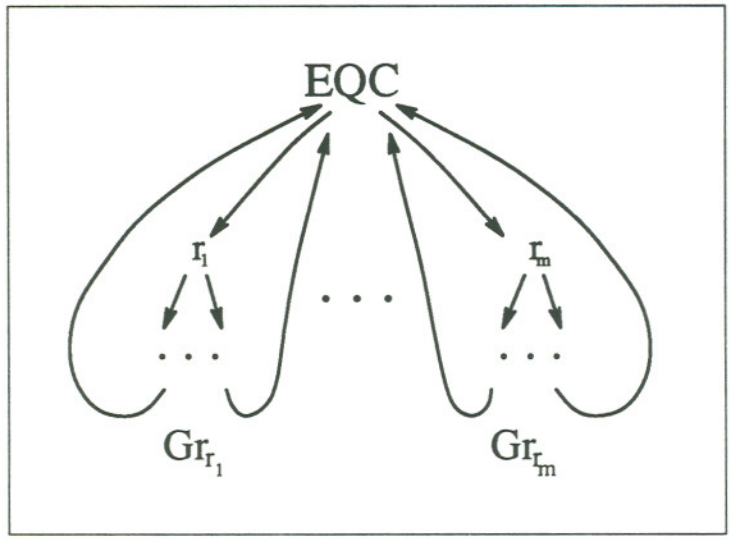

Figure 4.4: The composite graph

result of traversal of the composite graph will be either a constant c in case the traversal of one or more of the component subgraphs is the same constant or SW in case there are no constants in the component subgraphs.

– there is more than one occurrence of v in q. Then the composite graph assigned to i_v has at its root an EQC node; this node corresponds to equality checks for inherited position variables for v, i.e., variables corresponding to different occurrences of v in q. The edges from the root EQC node point to subgraphs corresponding to subgraphs (i.e., their roots) for inherited position variables. Each of the subgraphs is built essentially in the same way as in the previous case (when there is only a single occurrence of v in q)—given an occurrence o for an inherited position variable i_{vj}, we need to consider what is the term in the corresponding position in the goal, i.e., what is $o(G)$. The result follows by reasoning similarly. This situation is illustrated in Figure 4.5

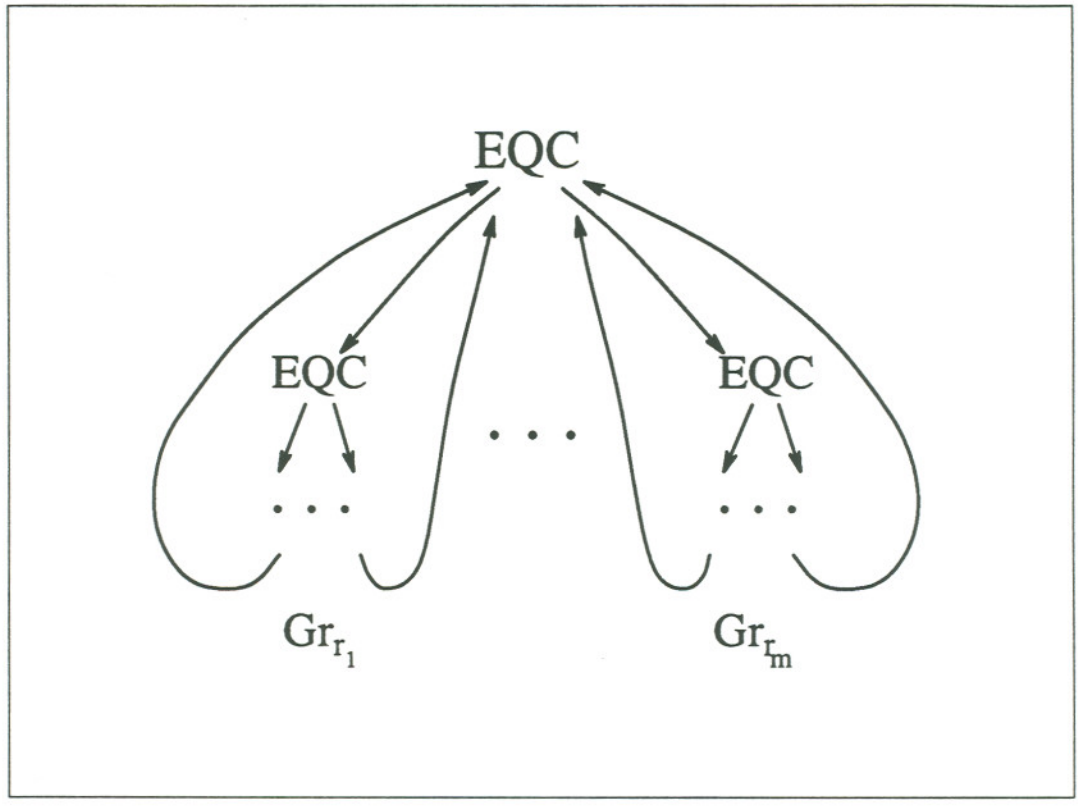

Figure 4.5: The composite graph for q

4.5 General programs

We have considered in the preceding sections Datalog programs, which are a restricted form of logic programs. The difference is that only nullary constructors, i.e., constants are included. In the general case the basic properties still hold. The principal difference is in the translation of terms, which can have constructors of arbitrary arity. In Section 4.2 we have seen that an occurrence of a constant a in the head of a clause is translated to

$$F_p\ (\ldots, x, \ldots) \ldots = (\ldots, eqc\ x\ a, \ldots)$$

A new variable x is created and it is equated to a in the resulting tuple by returning *eqc* x a as the corresponding component. In the general case, a term $c(\ldots, x_1, \ldots, x_2, \ldots)$ can appear in the argument position; assume x_1 and x_2 appear as i-th and j-th argument of c, respectively. The translation is

$$F_p\ (\ldots, x, \ldots) \ldots = (\ldots, eqc\ x\ c(\ldots, \pi_i\ x, \ldots, \pi_j\ x, \ldots), \ldots)$$

The occurrences of variables are replaced in the output term by applications of projection functions π_k to the variable corresponding to the argument term.

4.5.1 Traversing graphs with arbitrary constructors

We have seen that in the case of Datalog programs, the bindings for variables can be found by traversing cyclic graphs consisting of EQC nodes. In case there corresponding variable has a (ground) binding, there will be a constant in the graph. In case of general programs, the situation is actually the same; if there is a ground binding for an existential variable, it will be reflected in the corresponding graph. The main difference, which complicates things a little bit, is that since constructors can be of arbitrary arity, the constant nodes can have descendants as well as EQC nodes. In essence if an existential variable has a ground binding, each subcomponent has to be in the subgraph of the graph corresponding to the binding.

4.6 A functional version of the traversal algorithm

An imperative algorithm for traversing graphs representing the bindings for variables has been described in Section 4.3. In this section we specify a functional version of the traversal algorithm. This version is intended for *call-by-need* evaluation strategy. The algorithm is still not purely functional since it marks its redex with SW after initiating the evaluation. The algorithm is presented by two mutually recursive functions. They are defined equationally using pattern matching. The set of values is defined as

$$V = SW + EQC(V \times V) + C_0 + \ldots + C_i \underbrace{(V \times \ldots \times V)}_{i} + \ldots$$

The definition includes

- A distinguished value SW, which corresponds to a cyclical graph containing no ground terms.
- Terms of the form $EQC(v1, v2)$ that correspond to the result of comparing two argument terms.
- A set of ground values, which are represented as *constructors* applied to arguments.

The intuition behind the algorithm is straightforward. In essence, we want to traverse the graph, making sure that all nodes have been visited. We have to be careful to avoid possible cycles in the graph which might cause looping. This problem is handled by marking each redex before it is examined. First we present the Datalog version. The algorithm is defined as follows:

$$tr : V \to V \to V$$
$$sel_t : V \to V \to V \to V$$

rec

$$tr\ SW\ c = c$$
$$\|\quad tr\ EQC(a,b)\ c = sel_t\ a\ b\ c$$
$$\|\quad tr\ x\ \_ = x$$

and

$$sel_t\ SW\ a\ c = tr\ a\ c$$
$$\|\quad sel_t\ a\ SW\ c = tr\ a\ c$$
$$\|\quad sel_t\ EQC(a,b)\ d\ c = sel_t\ (sel_t\ a\ b\ (tr\ d\ c))\ d\ c$$
$$\|\quad sel_t\ a\ EQC(b,d)\ c = sel_t\ a\ (sel_t\ b\ d\ (tr\ a\ c))\ c$$
$$\|\quad sel_t\ a\ b\ \_ = \textbf{if}\ a = b\ \textbf{then}\ a\ \textbf{else}\ \mathit{fail}$$

The signatures of tr and sel_t are given in the top two lines. The function tr is used for traversal of expressions. We have seen that the expressions can be represented as

graphs and that the key point in traversals of graphs is to ensure that entire graphs are always traversed. Function tr takes two arguments; the first argument is the expression to be traversed and the second one can be intuitively thought of as the remaining part of the expression of which the current argument is part. The reason the rest of the expression is represented by an argument is because at any point during the traversal the computation can get stuck because there are no constants in the current subexpression. But the current redex should not be left as it is since there may be constants in the rest of the expression. Since the evaluation strategy is call-by-need, the current redex has to be updated with the final value obtained after traversing the entire expression. If the the argument of tr is SW, the traversal continues with the remaining part of the structure. Function sel_t is used for comparison of arguments of EQC nodes. It traverses its arguments and checks if they are equal; in case they are, the common value (equal to both) is returned as result and in case the arguments are unequal, the whole traversal fails. We can think of this function as selecting the argument from which the result binding is going to come, hence its name (sel_t for select-traverse).

Let us describe the algorithm in more detail. The first equation for tr specifies the case in which the part of the graph that is being traversed is marked; in this case, the traversal of the current argument is abandoned and the result will be the value obtained from the rest of the graph. If the current value is an EQC node, then sel_t is invoked on arguments of the EQC node in order to find their values and check that they are equal. If the current graph is neither a SW nor an EQC node, then it is a ground value, and it is returned as the result.

The first two equations for sel_t handle the cases in which either of the arguments is SW. In this case, the result is obtained by traversing the other argument. In case either of the arguments is an EQC node, sel_t is applied to the arguments of the EQC node as we would expect. But in addition, the third argument is equal to the other argument of the top sel_t. The reason is that if the EQC node does not denote a ground value, the result

is denoted by the other argument (and the remainder of the graph). Finally, if none of the above cases applies, the arguments must be ground terms, which are compared for equality.

In case of general terms with arbitrary constructors, the algorithm is slightly changed:

rec

$$tr\ SW\ c = c$$
$$\|\quad tr\ EQC(a,b)\ c = sel_t\ a\ b\ c$$
$$\|\quad tr\ C_0\ \_ = C_0$$
$$\|\quad tr\ C_i(x_1,\ldots,x_i)\ c = C_i(tr\ x_1\ (\pi_1\ c),\ldots,tr\ x_i\ (\pi_i\ c))$$

and

$$sel_t\ SW\ a\ c = tr\ a\ c$$
$$\|\quad sel_t\ a\ SW\ c = tr\ a\ c$$
$$\|\quad sel_t\ EQC(a,b)\ d\ c = sel_t\ (sel_t\ a\ b\ (tr\ d\ c))\ d\ c$$
$$\|\quad sel_t\ a\ EQC(b,d)\ c = sel_t\ a\ (sel_t\ b\ d\ (tr\ a\ c))\ c$$
$$\|\quad sel_t\ C_0\ C_0\ \_ = C_0$$
$$\|\quad sel_t\ C_i(x_1,\ldots,x_i)\ C_i(y_1,\ldots,y_i)\ c = C_i(sel_t\ x_1\ y_1\ (\pi_1\ c),\ldots,sel_t\ x_i\ y_i\ (\pi_i\ c))$$

Since constructors can have arguments, it is necessary to traverse the graph representing a constructor as well as the subgraphs corresponding to the arguments; the traversal of each subcomponent might fail to produce a ground binding. Therefore the corresponding component of the third argument is passed so that the traversal can proceed.

4.7 Generating derivation trees

We have seen in Sections 4.3 and 4.6 two versions of a traversal algorithm. They represent the operational semantics for evaluation of bindings for (existential) variables. However, the versions presented really deal with the case in which derivation trees and

corresponding evaluation trees remain fixed during the evaluation of bindings. This case is reflected in the proof of Theorem 7 in Section 4.4. In a particular evaluation tree, the choices for equations defining the functions corresponding to predicates are fixed. In general, a predicate may have several clauses; the corresponding functions consequently will be defined by several equations. By taking different choices for the equations we obtain different evaluation trees. What we have done is to present an algorithm for evaluation of *deterministic* logic programs in which there is only one clause for each predicate. It is clear that in the general nondeterministic case where there are multiple clauses for each predicate, we need a search mechanism. This mechanism is of course an essential part of every interpreter for logic languages. The goal is to consider multiple derivation trees in some sequence so a proof tree will be found if it exists.

4.7.1 Choice points

The concept of *choice points* has been used extensively in implementations of interpreters for logic programming and also in many other applications that involve backtracking and nondeterministic search. The basic idea is that at a point in a program at which a nondeterministic choice is to be made, an initial choice is made according to some simple criterion (e.g., first equation in a list of equations). Since nothing ensures that this particular choice is the right one, it is necessary to record somehow the information about alternate choices. The computation proceeds according to the choice taken; there are basically three possibilities for possible outcomes of the computation:

- The computation terminates successfully; in this case if we are interested in the first successful result, we are done.
- The computation terminates unsuccessfully; at that point it is necessary to examine other choices which have not been considered when the latest decision was made. The information about other choices is usually saved in a data structure—the choice point.
- The computation diverges.

This strategy ensures that a solution will be found if it exists provided the computation does not diverge; all choices will be tried. The exact format of choice points depends on the particular implementation. It is clear however that choice points should contain enough information to restore the state of computation to the same state as when (the last) nondeterministic decision was made.

4.7.2 Choice points as alternate reductions

We have seen that a goal

$$:- q_1(t_{11}, \ldots, t_{1n_1}), \ldots, q_m(t_{m1}, \ldots, t_{mn_m}).$$

is translated to

$$\begin{array}{l}
\textbf{lrec } \ldots, y_{11}, \ldots, y_{1e_1}, \ldots = F_{q_1}\ (\ldots, y_1, \ldots, y_1, \ldots) \\
\qquad\qquad\qquad \vdots \\
\textbf{and } y_1 = eqc_{e_1}\ y_{11}\ \cdots\ y_{1e_1} \\
\qquad\qquad\qquad \vdots \\
\textbf{and } y_n = eqc_{e_n}\ y_{11}\ \cdots\ y_{1e_n} \\
\textbf{in} \\
\quad y_1, \ldots, y_n
\end{array}$$

Literals in the goal q_i are translated to applications of corresponding functions F_{q_i}. At each step of resolution, a literal is selected and a clause with the same head is chosen. Naturally there can be several clauses that match. In some resolution strategies, a choice among one of them is made and the computation proceeds. In our case, in the translated programs there is an application of a function F_{q_i} for each literal q_i. The resolution step is replaced by a *reduction step* of reducing the application of F_{q_i}. The choice in a resolution step is reflected in the translation as the choice which equation to use for reduction step.

It is clear that we would like to choose an equation out of all possible choices and

perform reduction according to it. The result is the same as in the case of deterministic logic programs. However it is also clear that we want to save the information about the other available choices for equations to use in reduction in case they are needed later. A simple way to think about the choices is to consider all possible equations for a predicate function as a list of equations. We can adopt a simple criterion for choosing the next equation for reduction by always taking the equation at the head of this list; the other available choices are represented by the tail of the list. So the choice point will contain the tail of the list of available equations and the head will be used for the current reduction.

What other information do we need in a choice point? Most certainly we need the arguments to the function since they are needed in order to perform the reduction step. We are going to use *call-by-need* evaluation rule for reduction of applications of predicate functions F_{q_i}. It is not necessary to do so and we might use *call-by-name* also. The call-by-need evaluation rule specifies that the representation of an application $f\ x$ of a function f to its argument x is overwritten with the representation of the result. This evaluation rule facilitates sharing since there are no repeated computations of applications.

To summarize, a choice point consists of:

- a list of functions representing alternate equations to be tried in reduction
- the argument tuple consisting of argument terms; the function chosen for the reduction step is applied to these terms
- the redex, i.e., a representation of the application node; this application is overwritten by the result of the reduction

It should be clear from the preceding discussion that the choice points are created at the points of reduction of application nodes (more specifically applications of functions corresponding to predicates). Choice points are maintained in a data structure. A simple solution is to put them in a list. We call this list the *choice list*. The choice

list is in essence global since every reduction is supposed to have access to it. The goal expression is repeatedly traversed; during the traversal checks are performed to make sure no failure occurs. As the goal expression is being traversed, the reductions of functions corresponding to predicates are performed; at each one of these reductions a new choice point is created and added to the choice list (assuming there are new alternatives to be tried). If there was a failure encountered during the traversal of the goal expression, a check is made to ensure there are additional choices available in the choice list. If there are not, the result of the program is failure; otherwise the goal expression is updated with the information from the choice list (producing a new choice list) and the process is repeated. This process can be expressed in the following algorithm:

```
repeat_traverse goal choices =
    let  new_choices, new_goal, result = traverse_goal choices goal
    in   if result ≠ FAIL then new_goal
         else   if no_choices new_choices then fail "No more choices"
                else   let   new_goal, new_choices =
                                    update_choice new_choices new_goal
                       in repeat_traverse new_goal new_choices
```

The function *update_choice* is not a pure function since it imperatively manipulates the goal. It updates the goal in place according to the information from the list of choice points. Traversal of the goal is performed by *traverse_goal*, which simply initiates traversal of each binding for an argument of a predicate. We have seen this routine in Section 4.6. It is important to note that the traversal routine is non-destructive—it is supposed to leave its argument intact. If we use call-by-name evaluation, then the only changes made by *tr* to its argument are the marks for detecting the cycles. It is easy to make sure that *tr* leaves its argument intact by ensuring that all marks are erased after evaluation. The argument can be preserved by having *tr* collect all redices it marks in a list; after the evaluation the marks on all nodes in the list are deleted. A more elegant

way to preserve graphs after traversals will be presented in the next chapter. It relies on the fact that the points at which the cycles are introduced are known at compile time. These points are exactly the occurrences of **lrec** expressions.

The search algorithm expressed by *repeat_traverse* specifies that the first successful answer, i.e., an answer different from *FAIL*, is returned as the solution. It should be clear that we can use very similar strategy if we wanted to collect multiple solutions. The key is that the search mechanism provided by choice points can be used in gathering multiple solutions. The main idea is that, after finding the first solution, there is nothing preventing us from looking for subsequent ones, just as if we have failed looking for the previous solution. So it would be very simple to modify the search algorithm to collect multiple solutions. For instance, *repeat_traverse* can be changed so that when a solution is found, it is stored by a side effect (e.g., printed on the output or stored in a separate data structure) and the computation proceeds as if *FAIL* has been returned.

4.8 Cyclic graphs—an artifact of the implementation?

We have seen that the operational semantics for evaluation of programs is based on traversal of cyclic graphs. A simple and natural way to traverse cyclic graphs is provided by marking nodes as they are traversed. It would appear that this marking is an inherent overhead of our evaluation strategy compared to implementations of logic programming since there does not seem to be anything similar to the marking of nodes in them. However, we believe that this is not the case. To substantiate our belief we are going to consider an implementation of Prolog [MW88] that is a representative of contemporary implementations.

One of the leading principles employed in this implementation is represented by the "Procrastination principle". The idea is simple; one would like to delay as much work as possible as long as possible during the evaluation in the hope that the delayed computations will not have to be performed in case a failure is detected. An application of this principle is represented by delaying composition of substitutions. The unification

routine is called *match*. It takes as arguments terms to be unified and returns as a result (the most general) substitution. The substitution is in the form of a list of replacements for variables encountered during the traversals of argument terms. The list of replacements is not necessarily in form of a valid substitution—there is additional processing to be done. Of particular interest for us is the requirement that the list of replacements should never have a replacement of the form $X = X$ for any variable X. The reason is because the dereferencing routine *deref*, which follows the chains of bound variables, would get caught in the obvious cycle in this case. Because of the problem with cycles, the matching routine on encountering a variable always checks if the image of the variable in the current substitution is the variable itself; if it is, no action is performed. This check, we believe, is analogous to the check for markings of nodes during traversals in our evaluation strategy. The checks are not an artifact of our evaluation strategy but they also exist in other implementations, presented in a different way. We can note that in the example of logic programming implementations we considered, the checks whether a variable is replaced by itself are performed on *every* match; in contrast, the corresponding checks in our strategy are performed only when the nodes are needed, i.e., when the bindings for variables are demanded. The checks are performed for the whole aggregate graph representing all bindings for a (existential) variable. In effect this enables our strategy to delay the checks for markings as long as the other computations.

It should be mentioned that other cyclicity checks do appear in logic programming in the form of the *occurs check*. The goal of the occurs check is to establish whether a variable is unified with a term having the variable as a subterm. There is no finite solution in the case this is true. Most implementations simply omit occurs check for the sake of efficiency. The implementations that do include it usually produce a failure in case the occurs check is true. However, there have been proposals in which unification succeeds in case the occurs check is true; the solution is assumed to be an infinite term.

Chapter 5

Code generation

The principal idea behind the research presented in this dissertation has been to demonstrate that logic programs can be translated to equations which can be viewed as functional programs. In Chapter 2 we saw how to perform this translation. In Chapter 3 a semantics for logic programs was presented in an indirect way by giving semantics to the target language for the translation. We have seen that this semantics is strongly based on the semantics for functional languages, in particular LML [Aug87].. We consider it no surprise that the implementation of the evaluator is very strongly related to implementations of functional languages, and LML in particular. In fact, we will see that the code generation is a simple modification of the code generation for LML. We think the simplicity of the modification is further evidence to support the main idea that logic programs can be viewed as functional programs.

5.1 Implementation

The implementation is based on Johnsson's and Augustsson's implementation of LML [Joh87b, Aug87]. The underlying technique is called *graph reduction*. Expressions in the language are represented by graphs, and the process of reduction is performed by transforming the graphs. The transformations are performed by sequences of code for an abstract machine, called the G-machine [Joh87b]. The G-machine was originally developed by Thomas Johnsson; we use a modified version due to Kieburtz [Kie85].

5.2 The G-machine

A state of the G-machine can be represented by a sextuple $\langle C, P, V, G, E, D \rangle$ where

- C is the code sequence that is currently being executed.
- P is the stack holding pointers to the representations of graphs in the memory.
- V is the stack for holding basic values; in our case it is used only for intermediate storage of constructor numbers.
- G is the graph consisting of all graphs built during the computation.
- E is the mapping of function names to *function descriptors*, which contain addresses of the code for the functions. In the G-machine, descriptors also contain information about the arity of a function, but since all functions in translation are unary, this information is not needed in our case.
- D is the *dump* stack which contains return addresses from function calls; elements of D are pairs consisting of:
  - a (pointer to a) code sequence; it can be viewed as the return address from a function call
  - a stack of node pointers; it can be viewed as the stack before the function call

5.2.1 G-machine nodes

G-machine nodes have one of the following forms:

- BASIC n—a node representing a basic constant n. Basic constants can be integers, booleans or characters; in our case integer nodes will suffice for machine representation of constructors
- PR n_0 n_1—a node representing a pair of nodes
- PR1 v n—a node representing a constructor; v is the constructor number and n is a node representing arguments of the constructor

- APP f x—a node representing a suspended application; f is the function descriptor node (representing a function to be applied) and x is a node representing the argument of the application
- HOLE—a freshly allocated node distinct from all other nodes; this form is used for construction of cyclic graphs used for representation of recursion constructs

5.3 Transition rules

The actions of G-machine instructions are specified by the state transition rules of the abstract machine. Rules have the form

$$P \Longrightarrow N$$

where P is the machine state before the execution of a G-machine instruction and N is the state after the instruction has been executed. The current instruction, the definition of which is given by the transition rule, is at the beginning of the code sequence. The rules for instructions in the compilation schemes are given below:

$$\langle \text{COPYP } k.c, n_0.\cdots.n_k.r, V, G, E, D\rangle \Longrightarrow \langle c, n_k.n_0.\cdots.n_k.r, V, G, E, D\rangle$$

$$\langle \text{MOVEP } k.c, n_0.n_1.\cdots.n_k.n_{k+1}.r, V, G, E, D\rangle \Longrightarrow \langle c, n_1.\cdots.n_k.n_0.r, V, G, E, D\rangle$$

$$\langle \text{POP}.c, n.r, V, G, E, D\rangle \Longrightarrow \langle c, r, V, G, E, D\rangle$$

$$\langle \text{ROTP } k.c, n_0.\cdots.n_{k-1}.n_k.n_{k+1}.r, V, G, E, D\rangle \Longrightarrow \langle c, n_k.n_0.\cdots.n_{k-1}.n_{k+1}.r, V, G, E, D\rangle$$

$$\langle \text{FST}.c, n.r, V, G[n = \text{PR } n_0\ n_1], E, D\rangle \Longrightarrow \langle c, n_0.r, V, G, E, D\rangle$$

$$\langle \text{SND}.c, n.r, V, G[n = \text{PR } n_0\ n_1], E, D\rangle \Longrightarrow \langle c, n_1.r, V, G, E, D\rangle$$

$$\langle \text{MK\_APP}.c, n_0.n_1.r, V, G, E, D\rangle \Longrightarrow \langle c, n.r, V, G[n = \text{APP } n_1\ n_2], E, D\rangle$$

$$\langle \text{MK\_PR}.c, n_0.n_1.r, V, G, E, D\rangle \Longrightarrow \langle c, n.r, V, G[n = \text{PR } n_1\ n_2], E, D\rangle$$

$$\langle \text{ALLOC}.c, p, V, G, E, D\rangle \Longrightarrow \langle c, n.p, V, G[n = \text{HOLE}], E, D\rangle$$

$$\langle \text{MK\_V1\_PR}.c, n_0.r, m.v, G, E, D\rangle \Longrightarrow \langle c, n.r, v, G[n = \text{PR1 } m\ n_0], E, D\rangle$$

$$\langle \text{UPDATE } k.c, n_0.n_1.\ldots.n_k.r, V, G[n_0 = N_0, n_k = N_k], E, D\rangle$$
$$\Longrightarrow \langle c, n_1.\ldots.n_k.r, V, G[n_0 = N_0, n_k = N_0], E, D\rangle$$

$$\langle \text{PUSHCONST } n.c, P, V, G, E, D\rangle \Longrightarrow \langle c, n.P, V, G, E, D\rangle$$

$$\langle \text{GET\_BYTE } k.c, P, V, G, E, D\rangle \Longrightarrow \langle c, P, k.V, G, E, D\rangle$$

$$\langle \text{EVAL}.c, n.r, V, G[n = \text{APP } f\ x], E[f = c_f], D\rangle \Longrightarrow \langle c_f, x.n.[], V, G, E, (c, r).D\rangle$$

otherwise

$$\langle \text{EVAL}.c, P, V, G, E, D\rangle \Longrightarrow \langle c, P, V, G, E, D\rangle$$

$$\langle \text{RET}.c, n.r, V, G, E, (c_r, p).D\rangle \Longrightarrow \langle c_r, n.p, V, G, E, D\rangle$$

The most important instruction of the machine is EVAL. It is used for evaluation of nodes. The very first action of EVAL is to examine the node on the top of the stack. There are two possible cases:

- A node is an APP node representing an application of a function. In this case EVAL initiates a jump to the code for the function, with arguments on the top of the stack. The current code address and the state of the stack are saved on the dump stack.

- Otherwise, the node is already evaluated an no action is performed.

The return form the evaluation sequence is performed by RET. Of the remaining instructions, COPYP, MOVEP, ROTP, and POP are used for stack manipulations; ALLOC, MK_APP, MK_PR and MK_V1_PR are used for allocating and creating new nodes; FST and SND are projections; PUSHCONST and GET_BYTE are used for pushing pointer constants and numerical constants onto the P- and V-stacks, respectively. Finally, UPDATE is used for updating nodes.

5.4 Compilation schemes

The code sequences that transform graphs are compiled from expressions in the language. The code generation algorithm for the language consists of several compilation schemes. Each scheme has two arguments: the environment which is a mapping of variables to numbers indicating their relative depths in the stack, and a number indicating the current depth of the stack. There are three main schemes:

- $\mathcal{E}$ scheme—compiles code to evaluate an expression in graph form and leave the result on top of the stack
- $\mathcal{C}$ scheme—compiles code to construct a graph for an expression and leave a pointer to the graph on the top of the stack
- $\mathcal{F}$ scheme—compile a function definition

$\mathcal{E}$ scheme—evaluate expression:

$$
\begin{aligned}
\mathcal{E}\ [\![c]\!]\ r\ n &= \text{PUSHCONST } c \\
\mathcal{E}\ [\![f]\!]\ r\ n &= \text{PUSHCONST } f \\
\mathcal{E}\ [\![x]\!]\ r\ n &= \text{COPYP } (n - r(x)); \text{ EVAL} \\
\mathcal{E}\ [\![c(t_1, \ldots, t_k)]\!]\ r\ n &= \mathcal{C}\ [\![t_k]\!]\ r\ n; \\
&\quad \mathcal{C}\ [\![t_{k-1}]\!]\ r\ (n+1); \text{ MK\_PR}; \\
&\quad \vdots \\
&\quad \mathcal{C}\ [\![t_1]\!]\ r\ (n+1); \text{ MK\_PR}; \\
&\quad \text{GET\_BYTE } c; \text{ MK\_V1\_PR} \\
\mathcal{E}\ [\![(t_1, \ldots, t_k)]\!]\ r\ n &= \mathcal{C}\ [\![t_k]\!]\ r\ n; \\
&\quad \mathcal{C}\ [\![t_{k-1}]\!]\ r\ (n+1); \text{ MK\_PR}; \\
&\quad \vdots \\
&\quad \mathcal{C}\ [\![t_1]\!]\ r\ (n+1); \text{ MK\_PR} \\
\mathcal{E}\ [\![\textbf{let } D \textbf{ in } M]\!]\ r\ n &= \mathcal{C}_{let}\ [\![D]\!]\ r\ n;\ \mathcal{E}\ [\![M]\!]\ r'\ n'; \\
&\quad \text{MOVEP } (n' - n - 1); \text{ POP } (n' - n - 1) \\
&\quad \textit{where } r', n' = \mathcal{X}\ [\![D]\!]\ r\ n \\
\mathcal{E}\ [\![\textbf{lrec } x = D \textbf{ in } M]\!]\ r\ n &= \mathcal{C}_{lrec}\ [\![D]\!]\ r\ n;\ \mathcal{E}\ [\![M]\!]\ r'\ n'; \\
&\quad \text{MOVEP } (n' - n - 1); \text{ POP } (n' - n - 1) \\
&\quad \textit{where } r', n' = \mathcal{X}\ [\![D]\!]\ r\ n \\
\mathcal{E}\ [\![M\ N]\!]\ r\ n &= \mathcal{C}\ [\![M\ N]\!]\ r\ n; \text{ EVAL}
\end{aligned}
$$

$\mathcal{C}$ scheme—construct graph:

$$
\begin{aligned}
\mathcal{C}\,[\![c]\!]\,r\,n &= \text{PUSHCONST } c \\
\mathcal{C}\,[\![f]\!]\,r\,n &= \text{PUSHCONST } f \\
\mathcal{C}\,[\![x]\!]\,r\,n &= \text{COPYP } (n - r(x)) \\
\mathcal{C}\,[\![c(t_1,\ldots,t_k)]\!]\,r\,n &= \mathcal{C}\,[\![t_k]\!]\,r\,n; \\
&\quad\ \mathcal{C}\,[\![t_{k-1}]\!]\,r\,(n+1);\ \text{MK\_PR}; \\
&\quad\ \vdots \\
&\quad\ \mathcal{C}\,[\![t_1]\!]\,r\,(n+1);\ \text{MK\_PR}; \\
&\quad\ \text{GET\_BYTE } c;\ \text{MK\_V1\_PR} \\
\mathcal{C}\,[\![(t_1,\ldots,t_k)]\!]\,r\,n &= \mathcal{C}\,[\![t_k]\!]\,r\,n; \\
&\quad\ \mathcal{C}\,[\![t_{k-1}]\!]\,r\,(n+1);\ \text{MK\_PR}; \\
&\quad\ \vdots \\
&\quad\ \mathcal{C}\,[\![t_1]\!]\,r\,(n+1);\ \text{MK\_PR} \\
\mathcal{C}\,[\![\textbf{let } D \textbf{ in } M]\!]\,r\,n &= \mathcal{C}_{let}\,[\![D]\!]\,r\,n;\ \mathcal{C}\,[\![M]\!]\,r'\,n'; \\
&\quad\ \text{MOVEP } (n'-n-1);\ \text{POP } (n'-n-1) \\
&\quad\ \textit{where } r', n' = \mathcal{X}\,[\![D]\!]\,r\,n \\
\mathcal{C}\,[\![\textbf{lrec } x = D \textbf{ in } M]\!]\,r\,n &= \mathcal{C}_{lrec}\,[\![D]\!]\,r\,n;\ \mathcal{C}\,[\![M]\!]\,r'\,n'; \\
&\quad\ \text{MOVEP } (n'-n-1);\ \text{POP } (n'-n-1) \\
&\quad\ \textit{where } r', n' = \mathcal{X}\,[\![D]\!]\,r\,n \\
\mathcal{C}\,[\![M\ N]\!]\,r\,n &= \mathcal{C}\,[\![M]\!]\,r\,n;\ \mathcal{C}\,[\![N]\!]\,r\,(n+1);\ \text{MK\_APP}
\end{aligned}
$$

$\mathcal{F}$ scheme—function definition:

$$
\begin{aligned}
\mathcal{F}\,[\![\pi_i]\!] &= \overbrace{\text{SND};\ldots\text{SND}}^{i-1};\ \text{FST} \\
\mathcal{F}\,[\![f\ x = e_1 \parallel \ldots \parallel x = e_k]\!] &=
\end{aligned}
$$

L_1 : $\mathcal{E}\ [\![e_1]\!]\ r\ 1$; save $(@, L_2, x)$ in *choices*; UPDATE 1; RET

$\vdots$

L_k : $\mathcal{E}\ [\![e_k]\!]\ r\ 1$; save $(@, L_{k+1}, x)$ in *choices*; UPDATE 1; RET

L_{k+1} : PUSHCONST *FAIL*; UPDATE 2; POP; RET

The tuples with arity greater than two are represented by nested pairs. The code sequence "save $(@, L_i, x)$" in the $\mathcal{F}$-scheme stands for the code to create a choice point and put it in the front of the current list of choice points (called *choices*). Recall that a choice point is a triple consisting of:

- a pointer to a redex (designated by @)
- a function corresponding to an equation in a function definition; this function is represented by a code pointer L_i
- the argument

The actual sequence is

PUSHCONST *choices*; ALLOC;
PUSHCONST *choices*; UPDATE 1; - create a new node for choices

COPYP 2; PUSHCONST L_i; MK_PR;
COPYP 4; MK_PR; - create a new choice point

MK_PR; UPDATE 1; POP - update the new list and pop it

Miscellaneous schemes:

$$\mathcal{X}\ [\![x_1 = e_1\ \textbf{and}\ \ldots \textbf{and}\ x_k = e_k]\!]\ r\ n = (r[x_1 \mapsto n+1, \ldots, x_k \mapsto n+k], n+k)$$

$$\mathcal{C}_{let}\ [\![x_1 = e_1\ \textbf{and}\ \ldots \textbf{and}\ x_k = e_k]\!]\ r\ n = \mathcal{C}[\![e_1]\!]\ r\ n; \ldots, \mathcal{C}[\![e_k]\!]\ r\ (n+k-1)$$

$$\mathcal{C}_{lrec}\ [\![x_1 = e_1\ \textbf{and}\ \ldots \textbf{and}\ x_k = e_k]\!]\ r\ n = \overbrace{\text{ALLOC}; \ldots; \text{ALLOC};}^{k}$$
$$\mathcal{C}_t[\![e_1]\!]\ r\ n;\ \text{UPDATE}\ k \ldots,$$
$$\mathcal{C}_t[\![e_k]\!]\ r\ (n+k-1);\ \text{UPDATE}\ 1$$

$$\mathcal{C}_t\ [\![M]\!]\ r\ n = \text{PUSHCONST}\ \textit{mark};\ \mathcal{C}\ [\![M]\!]\ r\ (n+1);\ \text{MK\_APP}$$

Of particular interest is the code compiled for **lrec** expressions; they are handled by the auxiliary scheme $\mathcal{C}_t$. Since **lrec** is a recursion construct, the representation of the value recursively defined by this construct is a cyclic graph. This technique is the common way of handling recursion constructs in functional languages [Joh87b, PJ87]. However, there is an important difference. In our case the meaning of a value defined by **lrec** is not the least fixpoint $\mathit{fix}\ F = \sqcup F^i(\bot)$ but the fixpoint computed by starting the computation from SW: $\mathit{lfix}\ F = \lim_{i\to\infty} F^i(SW)$. The initialization of the starting value to SW is performed by the pseudo-function *mark*; it is not a function because it performs the initialization by a side effect. Consider a declaration of a recursively defined value x

$$\textbf{lrec}\ x = M$$

The code compiled is the same as the code produced for

$$\textbf{rec}\ x = \mathit{mark}\ M$$

When the code for *mark* is executed, it performs the following:

- update the redex to SW
- evaluate the argument, in this instance M
- after the argument has been evaluated, the redex is restored to its original form

Since the redex is initialized to SW, the right hand expression M will be evaluated in an environment in which x is bound to SW. This situation is exactly what we expect according to the operational semantics. After the result has been returned, the redex is restored to its original form. In case the result is a failure, the original graph is updated according to the information in the most recent choice point and the expression is reevaluated.

Definitions of pseudo-functions:

$$\begin{aligned}
\mathcal{F}\,[\![mark]\!] &= \text{PUSHCONST } SW\text{; UPDATE 2;} \\
&\quad\ \text{COPYP 0; EVAL;} \\
&\quad\ \text{ROTP 1; PUSHCONST } tr\text{; MK_APP; UPDATE 2;} \\
&\quad\ \text{MOVEP 0; RET} \\
\mathcal{F}\,[\![upd_cp]\!] &= \text{MOVEP 2; MK_APP; UPDATE 1; RET}
\end{aligned}$$

5.4.1 The basis for the code generation

We have seen in this chapter how to generate code for our language. The underlying abstract machine is a conventional machine for execution of lazy functional languages. However, the most important point is that the code generation algorithm has been obtained by a simple modification of a well known algorithm for generating code for lazy functional languages. Instead of the G-machine and LML, we could have used another algorithm and the basic principles would be the same. The evaluation of the **lrec** recursive construct is performed by initializing the redex to SW during the evaluation of **lrec** expressions. The initialization step is very simple since it consists of updating a single node with SW. We have also seen that additional code has been added for handling of choice points. This code is needed for implementing the search strategy. The strategy we have used is rather straightforward. It can be viewed as a simple adaptation of standard concepts in implementations of search procedures. It is certainly possible to adopt other alternatives. The point is that the main contribution of our approach is in

the process of computing bindings for a given derivation tree.

There is another point that should be mentioned. In Chapter 4 we saw a traversal algorithm that employs markings of nodes for detection of cycles. In the algorithm, every EQC node had to be marked during the traversal. In the code generation algorithm presented in this Chapter, the marking is performed by the pseudo-function *mark*. Its applications are inserted by the compiler precisely at the occurrences of **lrec** expressions. Furthermore, the code is not only performing the marking but also the restoring of the redexes after the evaluation. The benefit of this strategy is that there is no need for a separate structure to keep track of marked nodes; the markings and restorings of redexes are performed simultaneously with the evaluation.

In a way, it is possible to look at the process of computation from the object-oriented point of view. Each **lrec** expression can be viewed as an object that knows how to "prepare" itself for evaluation, by initializing its redex to SW. The demand-driven evaluation ensures that the objects are evaluated as needed.

Chapter 6

Propagating bindings

This chapter considers a different problem that nevertheless illustrates issues in modeling effects of logical variables in functional languages. We want to model the bidirectional flow of information in logic programs. The goal is to make sure that potential bindings for logical variables are propagated correctly without checking for their consistency. The example we considered in Section 2.1 is a good illustration:

$p(X, Z) : -q(X, Y), r(Y, Z).$
$q(X, X).$
$r(X, X).$

The translation is

$$
\begin{array}{lllll}
F_p\ x_{in}\ z_{in} = & \textbf{let rec} & x_{out}, y_{outq} & = & F_q\ x_{in}\ y \\
& \textbf{and} & y_{outr}, z_{out} & = & F_r\ y\ z_{in} \\
& \textbf{and} & y & = & eqc\ y_{outq}\ y_{outr} \\
& \textbf{in} & x_{out}, z_{out} & &
\end{array}
$$

$$F_q\ x_{in1}\ x_{in2} = \textbf{let}\ x_{out} = eqc\ x_{in1}\ x_{in2}\ \textbf{in}\ x_{out}, x_{out}$$

$$F_r\ x_{in1}\ x_{in2} = \textbf{let}\ x_{out} = eqc\ x_{in1}\ x_{in2}\ \textbf{in}\ x_{out}, x_{out}$$

Imagine that we want to try two different goals: either $p(X, a)$ or $p(a, X)$, where a is some ground constant. Now, as we have seen in our example program, all logical variables are identified and consequently we expect that the answer substitution will bind the logical variable X to the constant a in both cases. In other words, we expect that the binding for X is correctly propagated in both cases. It is important to emphasize that the set of equations corresponding to the program remains the same; the only change is in the translation of the goal. What is interesting in this case is that we can exhibit a set of equations that has a solution in the usual domains for functional languages. In other words, we can submit it to a functional evaluator and it will perform the propagation of bindings correctly. Note that there are no special functions used for traversal of cyclic graphs in this case.

The basic idea is simple. We can consider the translation of predicates q and r, which, in essence, specifies that their arguments should be unified. Consider the previous translation:

$$F_q\ x_{in1}\ x_{in2} = \textbf{let}\ x_{out} = eqc\ x_{in1}\ x_{in2}\ \textbf{in}\ x_{out}, x_{out}$$

Since we do not want to check for consistency of bindings but just want to make sure their propagation is performed correctly, the arguments can simply be exchanged:

$$F_q\ x_{in1}\ x_{in2} = x_{in2}, x_{in1}$$

The arguments can be thought of as bindings for the variables. The (input) binding of each variable is returned as the output binding of the other one. This way, if any of the variables is given a binding in the program, it will be made available to the other variable, with which it is to be unified. This "switching" is performed whenever two variables are unified. Obviously, the function *eqc* is not needed any more. The translation of the program has changed slightly:

$$
\begin{array}{lllll}
F_p\ x_{in}\ z_{in} = & \textbf{let rec} & x_{out}, y_{outq} & = & F_q\ x_{in}\ y_{outr} \\
& \textbf{and} & y_{outr}, z_{out} & = & F_r\ y_{outq}\ z_{in} \\
& \textbf{in} & x_{out}, z_{out} & &
\end{array}
$$

The *eqc* test on two occurrences of the logical variable Y is no longer present. Instead, the corresponding bindings are simply exchanged. Note that the bindings for the occurrences of Y (y_{outq} and y_{outr}) are still defined recursively. As for the goal, the translation of $p(X, a)$ is:

$$\textbf{let rec } x, a = F_p\ x\ a \textbf{ in } x$$

while $p(a, X)$ is translated as:

$$\textbf{let rec } a, x = F_p\ a\ x \textbf{ in } x$$

Again, note that bindings for all variables in the goal are recursively defined.

To see that this system of equations can be solved by the rewriting engine of a conventional lazy functional evaluator, we can follow the sequence of rewritings specified by the equations step by step. For instance if the goal is $p(X, a)$ then its translation after the reduction of F_p is:

$$
\begin{array}{l}
\textbf{let rec } x, a = \\
\quad \begin{array}{lllll}
\textbf{let rec} & x_{out}, y_{outq} & = & F_q\ x\ y_{outr} \\
\textbf{and} & y_{outr}, z_{out} & = & F_r\ y_{outq}\ a \\
\textbf{in} & x_{out}, z_{out} & &
\end{array} \\
\textbf{in } x
\end{array}
$$

The equations containing applications of F_q and F_r can be further reduced according to

their definitions. We arrive at the following set of equations:

$$
\begin{array}{l}
\textbf{let rec } x, a = \\
\quad \begin{array}{lll}
\textbf{let rec} & x_{out}, y_{outq} & = \; y_{outr}, x \\
\textbf{and} & y_{outr}, z_{out} & = \; a, y_{outq} \\
\textbf{in} & x_{out}, z_{out} &
\end{array} \\
\textbf{in } x
\end{array}
$$

This set of equations can be solved now. We are looking for the value of identifier x. From the equations above, it is clear that $x = x_{out}$. Continuing, we have $x_{out} = y_{outr}$. Finally, $y_{outr} = a$ and the solution is $x = a$. The important thing to realize here is that this set of equations can be solved by a functional evaluator. Indeed, at every step, reduction proceeds by simply replacing every identifier in the right hand side of an equation by its definition. In a similar way, we can verify that the equations can be solved for the other goal $p(a, X)$ as well. Note that the only thing which has changed is the translation of the goal; the rest of the program is the same. Here is the translation of the goal:

$$\textbf{let rec } a, x = F_p \; a \; x \textbf{ in } x$$

After the reduction of F_p the result is:

$$
\begin{array}{l}
\textbf{let rec } a, x = \\
\quad \begin{array}{lll}
\textbf{let rec} & x_{out}, y_{outq} & = \; F_q \; a \; y_{outr} \\
\textbf{and} & y_{outr}, z_{out} & = \; F_r \; y_{outq} \; x \\
\textbf{in} & x_{out}, z_{out} &
\end{array} \\
\textbf{in } x
\end{array}
$$

Proceeding with the reductions the same way as before, we reduce applications of F_q and F_r obtaining the following:

```
let rec a, x =
    let rec  x_out, y_outq  =  y_outr, a
    and      y_outr, z_out  =  x, y_outq
    in       x_out, z_out
in x
```

From these equations, we see that $x = z_{out}$. Reducing further, $z_{out} = y_{outq}$, and after one more step $y_{outq} = a$. The solution is $x = a$.

The interesting point about the example above is that it shows that it is possible to translate a logic program to a functional program that will work for different directions of the flow of information. When the goal is $p(X, a)$, it is clear that the first argument of the predicate p is in output mode, since we are looking for a binding of the logical variable X. The second argument is clearly in input mode. When the goal is $p(a, X)$, the roles are reversed. This property of logic programs, the possibility of different directions of the flow of information, is one of their essential features. Reddy and others have claimed that this effect is not achievable in functional languages because in a functional program there is a clear distinction between outputs and inputs [Red86]. However the example above serves to illustrate that such effects are possible in functional languages.

In the examples considered, all variables in the final solution have ground bindings. This situation is what we intuitively expect, since the logic program in essence unifies all variables and the constant a is essentially propagated to all positions. But what if there is no ground binding in the entire program? Looking at the equations more closely, it is clear that the functional program will loop. Basically, as the reduction proceeds, identifiers are being replaced with each other, but this process never terminates because there are no ground bindings to terminate the reduction. In other words, informally speaking, the meaning of an unbound logical variable in the final solution is divergence ($\perp$). If there is a ground binding, the equations ensure that corresponding identifiers will be reduced to it, terminating the reduction.

6.1 Checking consistency

The above example is limited in that it performs only the propagation of bindings and it does not provide for any checks of their consistency. Clearly we would like to enhance our method so it can perform the necessary checks as well. We can try our simple example again to perhaps give us some insight into how to solve this problem. Consider the goal $p(a, b)$, where a and b are some constants. Intuitively, we know that the logic program with this goal should fail since all variables are unified and an attempt to unify a and b fails because they are different constants. The functional program produced by translation does not check for consistency, but what if we run it anyway? The goal is translated to:

let $s_1, s_2 = F_p\ a\ b$ **in** s_1, s_2

If we reduce this program, the solution we obtain is $s_1 = b$, $s_2 = a$. An interesting thing happens here. We can think of this program as specifying that a and b are passed as input bindings for arguments of the predicate p. In the logic program we expect these bindings to remain the same throughout the execution. But in the functional program the bindings returned are different. In a way, the program still performed what we wanted. It propagated each binding to all other positions. At this point an idea comes naturally to mind. It can be easily checked in the translation of the goal if the bindings returned agree with the bindings passed in. Obviously they always should. If they do not, the program should fail. So the goal can be translated to:

let $s_1, s_2 = F_p\ a\ b$ **in**
    if $s_1, s_2 = a, b$ **then** a, b **else** *fail*

6.1.1 Integrating propagation and checking of bindings

We have seen that, in essence, the idea is to perform the propagation of the bindings to all positions in the program that are unified. After we are sure that all bindings have propagated correctly, then we can perform the checks. This entire process can be performed if we use *two* inherited and synthesized attributes per position instead of a single one. Intuitively, one pair of inherited and synthesized attributes are used for propagation. The second inherited attribute is used for propagation of *final* bindings. It is important to emphasize this point; the second inherited attribute will always be bound to a ground binding, provided one exists among all positions that are unified. The consequence is that it will be safe to evaluate this position, so the corresponding function will be strict in this position. The second synthesized attribute is used for the final result, i.e., the results of checking the final bindings for consistency.

6.1.2 An example

We are going to consider a simple example which illustrates the process described above. Consider a simple logic program consisting of a single fact:

$choose([X|T], X, T).$

We present a function produced from it, following the description of the translation:

$$
\begin{array}{l}
F_{ch}\ (i_{11}, i_{21})\ (i_{12}, i_{22})\ (i_{13}, i_{23}) = \\
\quad (i_{12}.i_{13}, h_1.t_1), (hd\ i_{11}, h_1), (tl\ i_{11}, t_1) \\
\quad \textbf{where}\ h_1, t_1 = \\
\qquad \textbf{case}\ i_{21}\ \textbf{in} \\
\qquad\quad [] : fail \\
\qquad \| \ i_{211}.i_{212} : \textbf{if}\ (i_{211} = i_{22})\ \&\ (i_{212} = i_{23})\ \textbf{then}\ i_{22}, i_{23}\ \textbf{else}\ fail \\
\qquad \textbf{end}
\end{array}
$$

Let us analyze this program more closely. First, the program is a definition of the function F_{ch}. The function takes three pairs as arguments. These pairs correspond to inherited attributes. It returns a triple of pairs, each pair corresponding to a pair of synthesized attributes. The first components of the pairs of inherited attributes (i_{11}, i_{12} and i_{13}) are used for propagating the bindings. Note that they are used for propagation *only*. In particular, no checks are performed on them. The propagation is performed in a way analogous to the process described in the previous section. Because there are multiple occurrences of logical variables X and T, the corresponding positions are unified. The consequence is that we need to switch the bindings between these positions. This situation is reflected in the first component of the first pair in the output triple, which is $i_{12}.i_{13}$. We can see that this value is a list, the head being obtained from the inherited attribute corresponding to the other occurrence of variable X. Its tail is defined similarly by a value from the other position of the logical variable T. It is clear now that the values obtained from the first argument are passed to the second and third. In particular, the head of the corresponding list is returned as the first synthesized attribute for the second argument and the tail is returned as the first synthesized attribute for the third attribute. Intuitively, we can think of this switching as making a binding available to another occurrence of a logical variable.

The second components of input arguments (i_{21}, i_{22} and i_{23}) correspond to the second inherited attribute of the respective positions. It is safe to check these values for consistency, since they represent the values obtained after possible bindings are propagated to all positions. In the program above, this situation is reflected in the fact that the code is strict in values i_{21}, i_{22} and i_{23}, *i.e* they are evaluated for checking. There are three checks we need to perform:

- Verify that the first argument is indeed a non-empty list. not [].

- If the first argument is a non-empty list, we need to check that its head is equal to the second argument. The reason is because the same logical variable X occurs at corresponding positions.
- Similarly, the tail of the first argument has to be equal to the third argument.

If any of these checks fail, the whole program fails. If they succeed, the corresponding (common) values are returned as final results. Now let us look at an example of a goal:

$: -choose([1|[2]], 1, X).$

Its translation is:

let rec $(s_{11}, s_{12}), (s_{21}, s_{22}), (s_{31}, s_{32}) = F_{ch}\ (1.[2], s_{11})\ (1, s_{21})\ (s_{31}, s_{31})$
in s_{12}, s_{22}, s_{32}

We see that the first and the second argument are in input mode and the third one is in output mode. The first inherited attribute for the first argument of F_{ch} is consequently $1.[2]$ and the first inherited attribute for the second argument is 1. The third argument is in output mode and the value passed as the corresponding first inherited attribute is s_{31}, which is recursively defined. The values for the second inherited position for the first and the second argument of F_{ch} are also recursively defined. Recall that the way the program is supposed to work is that all possible bindings are propagated through the first inherited and synthesized attributes. The values returned as the first synthesized attributes by F_{ch} (s_{11}, s_{21} and s_{31}) are then passed as second inherited attributes. The second inherited attributes are supposed to have values equal to final bindings (after the entire propagation). We know that at the level of the goal, the first synthesized attributes will be equal to the result of the entire propagation, and these values are simply passed in again as second inherited arguments. Finally the result of the entire program is obtained from the second synthesized attributes returned by F_{ch}.

Using this translation, we can obtain translations of various goals. For instance, if the goal is:

$: -choose(X, 1, [2]).$

The corresponding translation is:

let rec $(s_{11}, s_{12}), (s_{21}, s_{22}), (s_{31}, s_{32}) = F_{ch}\ (s_{11}, s_{11})\ (1, s_{21})\ ([2], s_{31})$
in s_{12}, s_{22}, s_{32}

It should be emphasized that these programs are purely functional programs that can be solved by a (lazy) functional evaluator. They exhibit different directions of information flow, since any of the arguments can be used in either input or output mode. The essential point is that the translation of the fact representing the program is fixed. The translations of the goal change as the goal changes.

6.1.3 Incompleteness of the translation

Unfortunately, the strategy described above does not work for all logic programs. To see what goes wrong, it is enough to consider a simple example:

$p(X, X).$
$: -p(X, X).$

The problem is that there are two unifications of the same variable. Focusing only on the aspect of propagation of bindings, we expect the translation of the fact $p(X, X)$ to be:

$F_p\ x_{in1}\ x_{in2} = x_{in2}, x_{in1}$

The translation of the goal is:

let rec $x_1, x_2 = F_p\ x_2\ x_1$ **in** x_1, x_2

After one reduction step, we get that $x_1 = x_1$ and $x_2 = x_2$. We have *two* distinct values x_1 and x_2, which is not what we expect. There should be only one value since the variables in the logic program are unified. Note that we would get correct translation if the switching of arguments is performed in either the translation of the goal or in the translation of the fact but not in both. For instance if the goal is translated as:

let rec $x_1, x_2 = F_p\ x_1\ x_2$ **in** x_1, x_2

In this case we get $x_1 = x_2$ and $x_2 = x_1$. The values are clearly identified. The problem really is in the fact that additional switching causes the break in the propagation of the recursively defined bindings. It is essential that bindings are propagated to all positions. They always will be in the case the switching is performed exactly once.

6.2 Relationship with Linear Logic

We have seen in the preceding sections that the essential problem is how to propagate bindings to all occurrences of variables that are unified. The process of propagation can be illustrated graphically. The propagation of output components is represented by arrows directed bottom-up; the propagation of arguments is represented by arrows directed top-down. Consider the following example:

$p(Z, Z).$
$:-p(X, Y).$

The process of propagation is illustrated in Figure 6.1. Note that variables X and Y in the fact are unified because of occurrences of variable Y in the goal. In fact, all

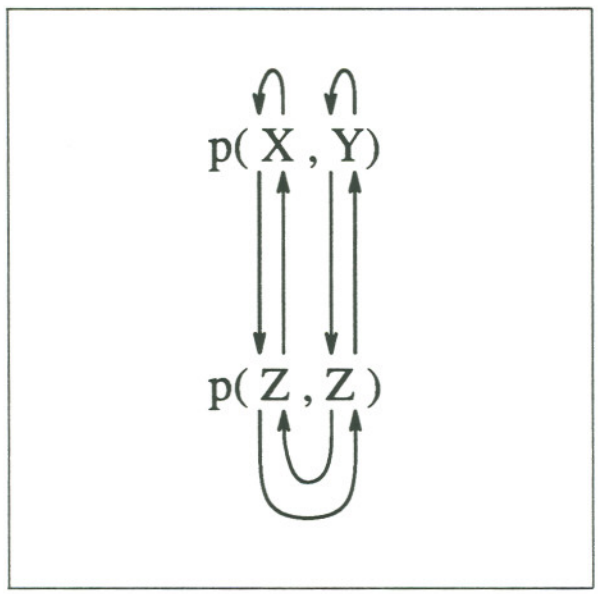

Figure 6.1: A graphical illustration of propagation

variables are unified. If we follow the arrows in the Figure, it is easy to see that the path specified by the arrows passes through every position in the tree. Had we had a constant instead of X or Y, it would have been propagated to the other position. Also if we had a fact $p(X, X)$ instead of $p(X, Y)$, the switching would have been performed twice and the path would not pass through every position in the tree. We can see that the key is that the switchings are performed so the paths created are of maximum length. The intuition about the paths of maximum length is important for understanding of possible generalizations of our scheme to programs with multiple (more than two) occurrences of variables. In all of the previous discussions we considered only cases in which there can be at most two occurrences of a variable. What about the example below?

$p(Z, Z, Z).$
$: -p(X, X, X).$

There are three occurrences of a variable, both in the fact and in the goal. The key again is the intuition about paths of maximum lengths. We can consider initially a situation

in which we view all variables as distinct, and no "switchings" are performed. Clearly, there is no single paths of maximum length, since there are three disjoint paths. If we start identifying variables, a pair at a time, the paths are going to be merged. It is clear that there always exists a combination of "switchings" such that we get a path of maximum length. The main problem is in determining which "switchings" should be performed and which should not. It is interesting to note that the requirement for the maximum lengths resembles very closely to the "long trip" condition for the "proof nets" in Linear Logic of J. Y. Girard [Gir87]. Proof nets can be viewed as a generalization of the formal notion of proof in Mathematical Logic. They are used for interpretation of proofs in Linear Logic. The relationship to proof nets seems to be a promising topic for further research.

Chapter 7

Related work and conclusion

7.1 Related work

The connection between attribute grammars and logic programming has been investigated previously [DM85]. It has been shown there is a close relationship between the two. This connection is not very surprising considering the similarities in structure of Horn clauses in logic programs and productions in attribute grammars. One of the main results is that for every logic program there is a semantically equivalent attribute grammar and *vice versa*. However it is important to note that this equivalence is established at the declarative level—the attribute grammar corresponding to a logic program has to exist but it is constructed using the information that can be obtained only by running the logic program. What is needed, in essence, is information about the direction of propagation of bindings for logical variables in logic programs; it is necessary to know which variables are in input mode and which are in output mode. This information is needed to determine which attributes are synthesized and which are inherited. The consequence of the fact that the relationship is established only at the declarative level is that that we cannot use it as a basis for *computation* of logic programs. In case some information about input-output splitting is known, it is possible to use attribute grammar evaluators to evaluate logic programs [DM85, AFZ88]. But the common point about these approaches is that attribute evaluators can be used only for a restricted class of logic programs. Our approach is fundamentally different since it demonstrates a translation of unrestricted logic programs to sets of equations that can be viewed as a

functional program.

Our approach has been inspired by the work on implementation of attribute grammars in lazy functional languages [Joh87a]. One of the motivations in that work was to present a technique for eliminating multiple traversals of data in functional programs in a systematic way—it turned out that attribute grammars presented a very nice framework for this problem. The techniques for eliminating multiple traversals from functional programs were investigated previously [Bir84]. However, those techniques were not presented in a systematic way—the transformations relied on programmer's intuition. The most important feature in both approaches was the use of recursively defined data definitions. In Johnsson's work recursive definitions were used for definitions of attribute values in top-level production. In our approach the values of existential variables are defined recursively. However, no connection to logic programming was made in Johnsson's approach. In fact, there is still a need to determine inputs and outputs in order to define synthesized and inherited arguments. The idea that both inherited and synthesized attributes should be provided for arguments which can be in arbitrary mode (either input or output) is one of the principal inspirations for this research.

Fixpoints are standard means of giving (operational) semantics of logic languages [KvE76]. The essential idea is that the set of all facts implied by a logic program is a fixpoint of the "immediate consequence" operator T_P. Each application of T_P adds to the current set of facts all facts that can be shown to be implied by a resolution with a single clause. Each application might add some new facts that are not present in the current set; the set of all facts implied by the program is obtained by iterating T_P to infinity. The result is a least fixpoint of T_P. There have been other applications of fixpoints to semantics of logic programs [GGdM89]. One way to view these approaches is by looking at fixpoints as the semantics of mutually recursive predicates. In a logic program, there are no restrictions on occurrences of predicates in bodies for their clauses and bodies of clauses for other predicates. The consequence is that predicates are in general mutually recursive. In our approach this situation is reflected in the fact that *functions* F_p corresponding to predicates can be defined recursively . In consequence, the

usual **rec** construct is needed in the target language. Our approach differs fundamentally from the others in that fixpoints are used for computation of bindings for variables. In this case, the solution is not obtained by least fixpoints but as a limit of iteration starting from SW. In essence, the fixpoint semantics is used for interpreting *sharing* of bindings between variables.

7.2 Conclusion

The main contribution of this dissertation is showing how to translate logic programs to sets of equations that can be viewed as functional programs. The programs produced by the translation do not contain any interpretation of variables—all variables are treated functionally. This feature distinguishes our approach from the previous proposals in a fundamental way. The main reason the variables have been interpreted is because it is believed that such treatment is necessary to achieve effects of logical variables in functional programs. We believe that this dissertation demonstrates that such effects are achievable in functional languages.

Semantics of SLD-resolution plays a prominent role in logic programming [Apt90]. This semantics requires *standardization apart*, which is simply a process of renaming different instances of program clauses with different variables. It is necessary to have this requirement to prevent name clashes. Since the need for fresh variables is a part of the resolution semantics for logic programs, this issue also plays a prominent role in operational semantics as well. In fact, most of the complexity involved in the implementation of logic programs is due exactly to ensuring that bindings for variables are handled correctly. For instance, there are two basic mechanisms for generating new instances of clauses [MW88]:

- **copy on use**—new instances are generated by creating new copies of clauses containing fresh variables.
- **structure sharing**—a common template of a program clause is kept and new

instances are generated by packaging this template with a substitution which renames variables in the clause to fresh ones; the benefit of this approach is that templates can be shared across different instances.

But in both of these mechanisms there is a need to create new variables for each new instance of a clause. In contrast to the treatment of variables in logic programs, the treatment of variables in functional programs is completely different. The problem of name clashes is also present in λ-calculus. There are various solutions to this problem. A particularly elegant one is provided by *combinators*, which does away with variables in programs altogether. The common point of these approaches is that in functional languages variables can be treated simply as placeholders for *values*, or, to be more precise, terms without any variables themselves. This difference becomes clearer if we compare unification and pattern matching; both are instances of the same problem. The difference between them is that in pattern matching only one of the terms can have variables. As a direct consequence, the mechanism for handling variables in functional languages is simpler and more efficient. Indeed, most optimizations of logic programs rely on replacing unification by pattern matching whenever possible. This analysis might lead one to believe that there is an inherent difference in nature of variables in logic and functional programming languages. To the author's knowledge, there have been no attempts to approach the treatment of variables in logic programs from the point of view of functional programming. In fact, there have been claims that effects achieved by logical variables in logic programs are not achievable in functional programs and that (first-order) functional programs are less expressive than logic programs [Red86]. Our work shows that this is not the case by exhibiting a translation from logic programs to sets of equations which can be viewed as functional programs. The important point is that in the equations produced by the translation there are no notions of separate representations of variables—there are only values. There is a need for a special recursive construct (**lrec**), which nevertheless is used for definitions of values. The direct consequence of the translation is that variables are treated in functional way.

The fact that the equations produced by the translation define values is the main

reason they are considered functional programs. The equations are indeed syntactically valid functional programs; they can be submitted to a lazy functional evaluator. However, we saw that the solutions we are interested in are not the least ones. They are computed by fixpoint iteration from a new value SW in the domain. The fact that the domain is modified by adding a *single value* is another reason why we view the equations as functional programs. The semantics is still defined in terms of functions over values, just like in conventional functional languages. The code generation algorithm provides further evidence of the functional nature of the equations produced by the translation. The algorithm is obtained by a very simple modification of a functional algorithm.

It should be said that the functional programs produced by the translation are first-order. Use of higher-order functions is often considered to be an essential feature of functional programming. One might ask what is the basic reason to view equations produced by translation as functional programs. We have seen that the treatment of variables is one feature that supports this view. But the fundamental reason is because the equations are *directed.* In contrast, equations in unification are undirected. The undirected nature of equations in unification is the source of its great expressive power—it is possible to have effects exhibiting bidirectional flow of information. But we have shown that these effects can also be achieved with directed equations by use of a recursive construct.

The translation essentially does not affect completeness properties of logic programs. It is well known that the completeness relies on the search strategy; there is a tradeoff between depth-first strategy, which is incomplete but efficient, and breadth-first, which is complete but less efficient. The same tradeoff exists for the functional programs produced by the translation. It is possible to define search strategies corresponding to depth-first and breadth-first.

We saw in Chapter 3 that a new distinguished value was added to the domain. There are indications that this addition is not absolutely necessary; ideas how to approach the problem in functional languages without changing the domain have been presented in Chapter 6.

Another point of interest is the recursive construct **lrec** itself; the semantics for it exhibits a natural model for which least fixpoints are not adequate. To understand why, we should realize that the least element $\perp$ denotes *nontermination.* The least element SW in the ordering of terms does not denote nontermination but simply lack of a ground binding for a variable.

7.3 Future work

There are many interesting issues raised by this research that merit further investigation:

- **Efficient implementation of logic languages**—the idea is to use techniques for efficient compilation of pattern matching from functional languages and apply them to evaluators for logic languages. From this point view, we can implement evaluators for logic languages by translating logic programs and evaluating the translated programs by modified functional evaluators.

- **Integration of logic and functional languages**—it almost goes without saying that we believe our work is relevant to languages that integrate logic and functions. The integration can be achieved by translating the logic part and viewing the result as a functional program.

- **Achieving effects of logical variables in functional programs**—in Chapter 6 we have seen how certain effects achievable by use of logical variables in logic programs can be realized in functional programs. These techniques can be generalized and we can define a methodology for translation of logic programs to functional programs. The effect of the most interest in this case is bidirectional flow of information by the propagation of bindings.

- **Relationship with Linear Logic**—it would be interesting to further investigate the connection to Linear Logic mentioned in Chapter 6.

- **Partial evaluation**—it is generally believed that, given a logic program, there are several functional programs corresponding to the uses of logic program with

different modes of logical variables. It is not known what is the relationship between these functional programs. We believe that our framework indicates a direction how to solve this problem. The idea is that the different functional versions are produced by partially evaluating the functional program produced by our translation.

- **Semantical considerations**—it would be interesting to further investigate the domains such as the one presented in Chapter 3. For instance, one interesting problem is the nature of limits involved in the definition of *lfix*. We demonstrated that they always exist for translations of logic programs but in the general case the question is open. In general, we would like to gain a better understanding of ramifications of the introduction of SW into the domain.

- **Relationship with the work on type theory**—we think it is quite clear that the work on type theory and type inference in functional languages is directly applicable to our framework. The idea is rather straightforward—one would type a logic program by translating it to a functional program that can be typed in conventional ways.

Bibliography

[AF86] Drew D. Adams and Jean Faget. Logic programming viewed functionally. In *Proceedings of the 1986 Graph Reduction Workshop*, Lecture Notes in Computer Science, Santa Fe, New Mexico, September 1986. Springer-Verlag.

[AFZ88] Isabelle Attali and Paul Franchi-Zannettacci. Unification-free execution of typol programs by semantic attribute evaluation. In *PLILP 1988*, Orleans, May 1988.

[Apt90] Krzysztof R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 10, pages 493–575. The MIT Press/Elsevier, 1990.

[Aug85] Lennart Augustsson. Compiling pattern matching. In J-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 368–381. Springer-Verlag, September 1985.

[Aug87] Lennart Augustson. *Compiling Lazy Functional Languages,Part II.* PhD thesis, Chalmers University of Technology, 1987.

[AUS86] A. V. Aho, J. D. Ullman, and R. Sethi. *Compilers: Principles, Techniques, Tools.* Addison-Wesley, Reading, Mass., 1986.

[BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Bir84] R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250, October 1984.

[BLM83] M. Belia, G. Levi, and M. Martelli. On compiling Prolog programs on demand driven architectures. In *Proceedings of Logic Programming Workshop*, Albufeira, Portugal, 1983.

[BMS80] R. M. Burstall, D. B. MacQueen, and D. T. Sanella. Hope: an experimental applicative language. Technical Report CSR-62-80, Department of Computer Science, University of Edinburgh, May 1980.

[Bru82] M. Bruynooghe. Adding redundancy to obtain more reliable and more readable Prolog programs. In *Proceedings of the First International Logic Programming Conference*, 1982.

[Car84] Mats Carlsson. On implementing Prolog in functional programming. In *Proc. 1984 International Symposium on Logic Programming*, pages 154–159, Atlantic City, February 1984. IEEE Computer Society.

[CD88] B. Courcelle and P. Deransart. Proofs of partial correctness for attribute grammars with applications to recursive procedures and logic programming. *Information and Computation*, (78):1–55, 1988.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[Deb89] Saumya Debray. Static inference of modes and data dependencies in logic programs. *ACM Trans. on Programming Languages and Systems*, 11(3):418–450, July 1989.

[DFP86] John Darlington, A. J. Field, and H. Pull. The unification of functional and logic languages. In Gary Lindstrom Doug DeGroot, editor, *Logic Programming: Functions, Relations and Equations*, pages 37–70. Prentice-Hall, 1986.

[DJL88] P. Deransart, M. Jourdan, and B. Lorho, editors. *Attribute Grammars: Main Results, Existing Systems and Bibliography*, volume 323 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1988.

[DM85] Pierre Deransart and Jan Maluszynski. Relating logic programs and attribute grammars. *Journal of Logic Programming*, (2):119–155, 1985.

[DM88] Pierre Deransart and Jan Maluszynski. A grammatical view of logic programming. Rapport de Recherche 883, INRIA, Rocquencourt, August 1988.

[DW88] Saumya Debray and David S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–229, September 1988.

[Fel85] Matthias Felleisen. Transliterating Prolog into Scheme. Technical report 182, Indiana University, Bloomington, October 1985.

[GGdM89] Georges Gardarin, Irene Guessarian, and Christophe de Mandreville. Translation of logic programs into functional fixpoint equations. *Theoretical Computer Science*, (63):253–274, 1989.

[Gir87] J. Y. Girard. Linear Logic. *Theoretical Computer Science*, (50):1–102, 1987.

[Hal68] Paul Halmos. *Naive Set Theory.* D. Van Nostrand Company, 1968.

[HJW91] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell, version 1.1. Technical report, Department of Computer Science, Yale University, 1991.

[JM84] Neil Jones and Alan Mycroft. Stepwise development of operational and denotational semantics for Prolog. In *Proc. 1984 International Symposium on Logic Programming*, pages 281–288, Atlantic City, February 1984. IEEE Computer Society.

[Joh87a] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In Gilles Kahn, editor, *Functional Programming Languages and*

Computer Architecture, volume 274 of *Lecture Notes in Computer Science*, pages 154–173, Portland, September 1987. Springer-Verlag.

[Joh87b] Thomas Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Chalmers University of Technology, 1987.

[Jou84] M. Jourdan. Strongly non-circular attribute grammars and their recursive evaluation. In *Proc. of 1984 ACM/SIGPLAN Conf. on Compiler Construction*, Montreal, June 1984.

[Kat84] Takuya Katayama. Translation of attribute grammars into procedures. *ACM Trans. on Programming Languages and Systems*, 6(3):345–369, July 1984.

[Kie85] R. B. Kieburtz. The g-machine: a fast, graph reduction evaluator. In *Proc. of IFIP Conf. on Functional Prog. Lang. and Computer Arch.*, Nancy, 1985.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Math. Systems Theory*, 2(2):127–145, 1968.

[KvE76] Robert Kowalski and M. H. van Emden. Semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, October 1976.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[LP84] Gary Lindstrom and Prakash Panangaden. Stream-based execution of logic programs. In *Proc. 1984 International Symposium on Logic Programming*, pages 168–176, Atlantic City, February 1984. IEEE Computer Society.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[MW88] David Maier and David S. Warren. *Computing with Logic.* The Benjamin/Cummings Publishing Co., 1988.

[PJ87] Simon L. Peyton-Jones. *The implementation of functional programming languages.* Prentice-Hall International, London, 1987.

[Red86] Uday S. Reddy. On the relationship between functional and logic programming. In Gary Lindstrom Doug DeGroot, editor, *Logic Programming: Functions, Relations and Equations*, pages 3–36. Prentice-Hall, 1986.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:32–41, 1965.

[Smo84] G. Smolka. Making the control and data flow of logic programs more explicit. In *ACM Symposium on LISP and Functional Programming*, 1984.

[Tur85] D. A. Turner. Miranda - a non strict functional language with polymorphic types. In J-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, September 1985.

[Wad71] C. P. Wadsworth. *Semantics and pragmatics of the λ-calculus.* PhD thesis, Oxford, 1971.

[Wei90] Pierre Weis. The CAML reference manual, version 2.6.1. Technical report, INRIA, 1990.

Biographical Note

The author was born on February 14th 1961 in Belgrade, Yugoslavia. Very soon, his interest in mathematics and natural sciences became apparent and it turned out to be the main guiding force behind his educational, professional and other interests in life.

The author graduated from Matematička Gimnazija in Belgrade (a high school specializing in mathematics) in 1979 and enrolled at Elektrotehnički Fakultet Univerziteta u Beogradu (College of Electrical Engineering at the University of Belgrade). He completed his studies in 1984 receiving the degree of Dipl. Ing. of Electrotehnics with a major in Electronics, additionally completing the requirements for equivalent degree in Computer Science. In 1985 he entered Oregon Graduate Institute (then Oregon Graduate Ceneter) to pursue his doctoral degree. He completed the requirements for a Ph.D. in Computer Science and Engineering in 1992.

His interests span several fields, including theory and practice of programming languages, functional programming, applications of logic in computer science, continuations, linear logic, category theory, computer architecture, VLSI design etc.

The author is leaving OGI to accept a position with Intel Corporation.