Logic Programming and PROLOG

Marc Bezem
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

We discuss the foundations of the programming language PROLOG in logic programming and demonstrate its use by examples. We explain unification (with and without occur check), SLD-resolution, depth-first search procedure, cut and negation as failure.

1. Introduction

Logic programming, in the broad sense of the term, is a style of programming which is based on mathematical logic. The underlying idea is that deduction and computation are strongly interrelated. Already in the seventeenth century Leibniz formulated his ideal of reducing all deduction to computation (by a 'calculus ratiocinator'). Leibniz' ideal was elaborated by Frege and Hilbert, until, by the work of Gödel, Church and Turing, a fundamental limitation became apparent. In general, it is impossible to decide on the truth of a proposition by a finite computation. The converse of Leibniz' ideal turned out to be less problematic. After suitable codification, every computation may be viewed as a deduction (in any reasonably strong deductive system). We refer the interested reader to [5] for an overview of deduction and computation.

In the narrow sense of the term, logic programming is a style of programming based on viewing computation as deduction in first-order logic (thus excluding other types of mathematical logic, such as higher-order logic, type theory etc.). The idea of using (a subset of) predicate logic as a programming language is formulated by Kowalski in [6]. A programming language based on this idea is PROLOG, due to Colmerauer, actually preceding [6].

The name of the programming language PROLOG stands for PROgramming in LOGic. It has the following prominent features: simple, concise syntax based on first-order logic; backtracking as a built-in control mechanism; unification as a powerful computation primitive. Being particularly suited for symbolic processing, applications of PROLOG can be found in the following fields: deductive databases, expert systems, theorem proving, computer algebra, natural language processing and compiler construction.

The goal of this paper is to provide a short introduction to logic programming and PROLOG, paying special attention to the relation between the two. There are four sections, including the introduction. Section 2 contains some necessary preliminaries, such as the syntax and the semantics of first-order logic, the syntax of logic programming, as well as the notion of unification. Section 3 deals with logic programming; we explain the execution of logic programs, SLD-resolution and SLD-trees. In Section 4 we discuss the foundations of PROLOG in logic programming and demonstrate its use by examples. Good introductory texts on logic programming are [1] and [7]. For PROLOG we recommend [3] and [9]. Our exposition of logic programming is based mainly on [1].

2. Preliminary definitions and notation; unification

2.1. Syntax of first-order logic

The syntax of logic programming languages such as PROLOG is based on first-order logic (not to be confused with the propositional logic from the article of van Emde Boas). The *alphabet* of first-order logic consists of constants a,b,c, variables x,y,z, function symbols f,g,h, predicate symbols p,q,r, propositional connectives ($\neg$ for negation, $\vee$ for disjunction, $\wedge$ for conjunction, $\rightarrow$ for implication), and quantifiers (universal quantifier $\forall$, existential quantifier $\exists$).

The set of *terms* contains all variables and constants, and is closed under function application. For example $f(g(a),x)$ is a term. We use s,t as syntactical variables for terms.

Atoms, or *atomic formulas*, are obtained by applying predicates to terms, for example $p(f(g(a),x),y,h(x))$ is an atom. We use A,B as syntactical variables for atoms.

Formulas are constructed from atomic formulas with the help of propositional connectives and quantifiers.

To all denotations we add primes and subscripts when necessary.

2.2. Semantics of first-order logic

Throughout this paper we use $\models$ to denote the Tarskian deduction relation. More precisely, by $P \models F$ we express that a formula F is a (*logical*) *consequence* of a set of formulas P in the sense of the Tarski semantics of first-order logic (see any textbook on mathematical logic, such as [4]). Two formulas are called (*logically*) *equivalent* if they are consequences of each other.

2.3. Syntax of logic programming

A *literal* is an atom or the negation of an atom (so-called *positive* and *negative* literals). A *clause* is the universal closure of a finite disjunction of literals. Thus the general form of a clause is

$$\forall x_1, \ldots, x_l (L_1 \vee \cdots \vee L_k),$$

where $x_1, \ldots, x_l$ ($l \geqslant 0$) are all the variables occurring in the literals

$L_1, \ldots, L_k$ ($k \geqslant 0$). This includes the empty clause, denoted by $\square$, as well as clauses consisting of exactly one literal. The empty clause stands for *falsum*, a false proposition (this is the natural interpretation of $\square$ according to the Tarski semantics). From now on clauses will be written in a special form, indeed called *clausal form*. The above clause written in clausal form reads

$$A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n,$$

where $A_1, \ldots, A_m$ ($m \geqslant 0$) list all positive L_i's, and $B_1, \ldots, B_n$ ($n \geqslant 0$) all negative L_i's. If we interpret the commas in the left hand side as $\vee$ and in the right hand side as $\wedge$, and $\leftarrow$ as implication from right to left, then the universal closure of the clausal form is logically equivalent to the clause itself.

A *Horn clause* is a clause containing at most one positive literal. The restriction from clauses to Horn clauses is motivated by efficiency. We distinguish between *program clauses* (or *definite clauses*), which contain exactly one positive literal, and *goal clauses*, which consist entirely of negative literals. Thus the empty clause is a goal clause. Program clauses consisting of exactly one positive literal are also called *unit clauses*. We use G as a syntactical variable for goal clauses, often simply called *goals*. A *logic program* is a finite set of program clauses.

We sum up the notational conventions concerning logic programming in Table 1. From now on the universal (respectively existential) closure of a formula F will be denoted by $\forall(F)$ (respectively $\exists(F)$).

	Horn clause	Intended meaning
program clause	$A \leftarrow B_1, \ldots, B_n$ ($n \geqslant 1$)	$\forall((B_1 \wedge \cdots \wedge B_n) \rightarrow A)$
unit (program) clause	$A \leftarrow$	$\forall(A)$
goal clause	$\leftarrow A_1, \ldots, A_m$ ($m \geqslant 1$)	$\forall(\neg(A_1 \wedge \cdots \wedge A_m))$
empty (goal) clause	$\square$	*falsum*

TABLE 1

2.4. Unification

An important notion we shall need in the sequel is that of substitution. A *substitution* is a finite set of pairs of variables and terms, denoted by $\{x_1/t_1, \ldots, x_k/t_k\}$, where all x_i's are distinct and each t_i is different from x_i. The term t_i is called a *binding* for x_i. For example $\{x/f(a), y/g(x,b), z/y\}$ is a substitution. We use θ, σ as syntactical variables for substitutions.

The *application* of a substitution σ to a syntactical expression E (a term, an atom, or a list of atoms), denoted by $E\sigma$, is the result of *simultaneous* replacement of the occurrences of variables in E by their bindings according to σ. For example, if σ is the substitution above and E is $p(x', f(x), z)$, then $E\sigma$ is $p(x', f(f(a)), y)$.

It is not difficult (see for example [1, §2.3]) to define the *composition* of two substitutions σ and θ, denoted by $\sigma\theta$, in such a way that $(E\sigma)\theta$ is identical to $E(\sigma\theta)$, for all expressions E. One can prove that composition of substitutions is associative, and so brackets in expressions like $E\theta_0 \cdots \theta_n$ are immaterial.

A *unifier* of two atoms A and A' is a substitution θ such that $A\theta$ is syntactically identical to $A'\theta$. For example $\{x/y\}$, $\{y/x\}$, $\{x/z, y/z\}$ and $\{x/f(a), y/f(a)\}$ are unifiers of $p(x)$ and $p(y)$. The atoms $p(x)$ and $p(f(x))$ do not have a unifier, they are not *unifiable*.

A substitution θ_2 is called *more general* than θ_1 if there exists θ_3 such that $\theta_1 = \theta_2\theta_3$. For example, the first three unifiers above are more general than the fourth. A unifier of two atoms is called a *most general unifier* (*mgu* for short), if it is more general than any other unifier. The first three unifiers above are mgu's of the atoms involved, the fourth is not. Another example: an mgu of $p(f(x,y),g(h(x)))$ and $p(z,g(h(a)))$ is $\{x/a, z/f(a,y)\}$. Note that the substitution of terms for variables in an atom is a specialization, which may involve a loss of information. The idea behind an mgu is that the unification is achieved while preserving as much generality as possible. We conclude this section with the following important theorem.

UNIFICATION THEOREM (ROBINSON [8]). *There exists an algorithm which produces a most general unifier of two given atoms if they are unifiable, and otherwise reports that they are not unifiable.*

3. EXECUTION OF LOGIC PROGRAMS

3.1. Basic idea

Let P be a logic program and G the goal $\leftarrow A_1, \ldots, A_n$. The execution of P with respect to G aims at *refuting* G from P, i.e. showing that $\neg G$ is a consequence of P. We recall that the intended meaning of G is $\forall(\neg(A_1 \wedge \cdots \wedge A_n))$, so $\neg G$ means $\exists(A_1 \wedge \cdots \wedge A_n)$. However, it can be proved that in general

$$P \models \exists(A_1 \wedge \cdots \wedge A_n)$$

is undecidable. This hard fact of life leaves us with the following three possibilities: an answer 'yes', an answer 'no', or no answer at all (infinite computation). Successful refutation of G from P moreover yields a substitution θ such that

$$P \models \forall((A_1 \wedge \cdots \wedge A_n)\theta).$$

Such an *answer substitution* θ is very informative: it provides a set of witnesses for the existential theorem $\exists(A_1 \wedge \cdots \wedge A_n)$.

EXAMPLE. Consider the logic program consisting of program clauses $q(a)\leftarrow$, $p(g(a))\leftarrow$ and $p(f(x))\leftarrow q(a)$, and the goal $\leftarrow p(y)$. Examples of answer substitutions are $\{y/f(x)\}$ (yielding a set of witnesses for $\exists y(p(y))$, namely the set of terms of the form $f(t)$) and $\{y/g(a)\}$ (yielding the witness $g(a)$). For the goal $\leftarrow p(y), q(y)$ no answer substitution exists.

3.2. SLD-resolution

Let us take a closer look at the execution of a logic program P and a goal $\leftarrow A_1, \ldots, A_n$. The refutation procedure called SLD-resolution (Selective Linear resolution for Definite clauses) goes as follows. Select an atom, say A_i, from

$$G_0 = \leftarrow A_1, \ldots, A_i, \ldots, A_n.$$

Choose a clause $A \leftarrow B_1, \ldots, B_m$ $(m \geqslant 0)$ from P such that A_i and A are unifiable with mgu θ_0 (see remark below). Go on with

$$G_1 = \leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n)\theta_0.$$

This procedure terminates when either $G_k = \Box$, or no program clause from P applies to the selected atom of G_k. The latter case is called *failure*, the former *success*, or *successful refutation*, since the intended meaning of $\Box$ is *falsum*.

We now show that a successful refutation yields an answer substitution. Assume conditions are as in the previous paragraph. For any goal G, let the conjunction of the atoms from G be denoted by $\sim G$. For example $\sim G_0$ denotes $A_1 \wedge \cdots \wedge A_n$, and $\sim \Box$ denotes the empty conjunction, or *verum*, a true proposition. We obviously have

$$P \models ((B_1 \wedge \cdots \wedge B_m) \rightarrow A_i)\theta_0,$$

since $A_i\theta_0$ is syntactically identical to $A\theta_0$. Hence

$$P \models (A_1 \wedge \cdots \wedge A_{i-1} \wedge B_1 \wedge \cdots \wedge B_m \wedge A_{i+1} \wedge \cdots \wedge A_n)\theta_0 \rightarrow$$
$$(A_1 \wedge \cdots \wedge A_{i-1} \wedge A_i \wedge A_{i+1} \wedge \cdots \wedge A_n)\theta_0,$$

or, in other words,

$$P \models \sim G_1 \rightarrow \sim G_0\theta_0.$$

More generally, we have for all $0 \leqslant i < k$

$$P \models \sim G_{i+1} \rightarrow \sim G_i\theta_i.$$

So if $G_k = \Box$ for some k, then it follows by successive application of this argument for $i = 0, \ldots, k-1$ that

$$P \models \sim \Box \rightarrow \sim G_0\theta_0 \cdots \theta_{k-1}.$$

Hence for $\theta = \theta_0 \cdots \theta_{k-1}$ we finally obtain, as $\sim \Box$ is a true proposition,

$$P \models \forall((A_1 \wedge \cdots \wedge A_n)\theta),$$

so θ is the desired answer substitution.

REMARK. Sometimes variables in a program clause have to be renamed. For example, the logic program consisting of just $p(f(x))\leftarrow$ should successfully refute the goal $\leftarrow p(x)$, although $p(f(x))$ and $p(x)$ are not unifiable. Renaming will be done in a systematic way by (implicitly) adding as subscripts to the variables of the program clause the number of steps taken in the refutation procedure. Note that this does not change the meaning of the clauses, because of the universal closure. In the example above the answer substitution becomes $\{x/f(x_1)\}$.

3.3. SLD-tree

The reader will have observed two elements of non-determinism in the refutation procedure described above. The first is the selection of an atom from the goal, and the second is the choice of the program clause. As to the selection of atoms from goals, one can state and prove independency results with respect to successes and answer substitutions of SLD-resolution. Since this would lead us too far, we refer the reader to [1] or [7]. Given this independency we can safely fix a so-called *selection rule*, which defines for every goal which atom is selected. Once a selection rule has been fixed we can view the refutation procedure described in the previous subsection as pursueing a path in a tree, called *SLD-tree*. The nodes of this tree are goals, the branches represent different choices of program clauses. The root of an SLD-tree is the goal with which the refutation procedure starts, and a leaf represents either a success (the empty clause), or a failure (no program clause applicable to the selected atom of the goal). An SLD-tree should be considered as a search space. Note that SLD-trees are finitely branching trees (since programs are finite), which possibly contain infinite paths (consider the logic program $\{p\leftarrow p\}$ and the goal $\leftarrow p$).

We conclude this section with two examples of SLD-trees, taken from [2]. Consider the logic program consisting of the following three program clauses.

1. $path(x,z)\leftarrow arc(x,y),path(y,z)$
2. $path(x,x)\leftarrow$
3. $arc(b,c)\leftarrow$

The first two clauses express that *path* is the transitive and reflexive closure of *arc*. The third clause defines the relation *arc*. We consider two SLD-trees with the goal $\leftarrow path(x,c)$ as root, but with different selection rules. In Figure 1 the leftmost atom of each goal is selected, and in Figure 2 the rightmost. The branches are labeled by the mgu and by the number of the program clause used. For ease in reading the selected atoms are bold-faced and the bindings for variables (if any) of the goal precede those (if any) of the program clause in the denotations of the mgu's. Note that both SLD-trees yield the answer substitutions $\{x/b, \ldots\}$ and $\{x/c, \ldots\}$.

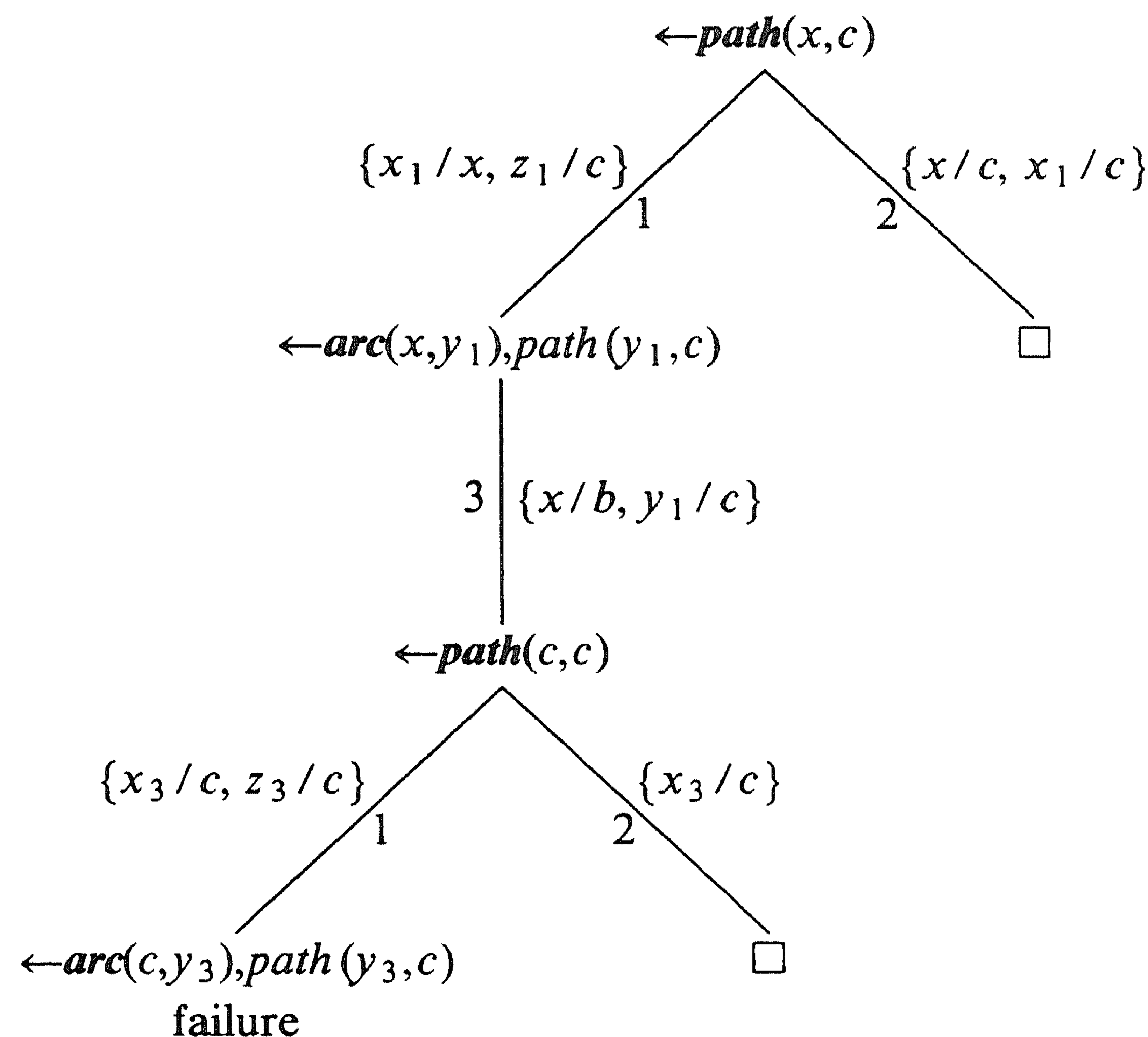

FIGURE 1

4. PROLOG

4.1. PROLOG versus logic programming

The programming language PROLOG is in the first instance based on logic programming. However, there are a number of important differences. Most differences are motivated by practical arguments such as efficiency, ease of implementation and ease of use, and are deeply regretted by the theoretical community. We shall mention some of these differences in the next paragraphs, together with short comments.

PROLOG uses a selection rule which selects of each goal the leftmost atom. As long as the independency results mentioned in the first paragraph of subsection 3.3 are valid, there is no problem. However, if one enriches logic programming with negation, then the leftmost selection rule causes problems. These problems are beyond the scope of this tutorial (see [1]).

PROLOG searches depth-first from left to right in the SLD-tree, where the direction from left to right corresponds to the textual order of the program clauses. It thus misses successes which are on the right of an infinite path. Moreover the textual order of the program clauses becomes crucial. On the other hand: from a *programming* point of view, without an emphasis on *logic*, the PROLOG way of searching in SLD-trees constitutes a flow of control, often called *backtracking*, which is very powerful (see the last example in Subsection 4.2).

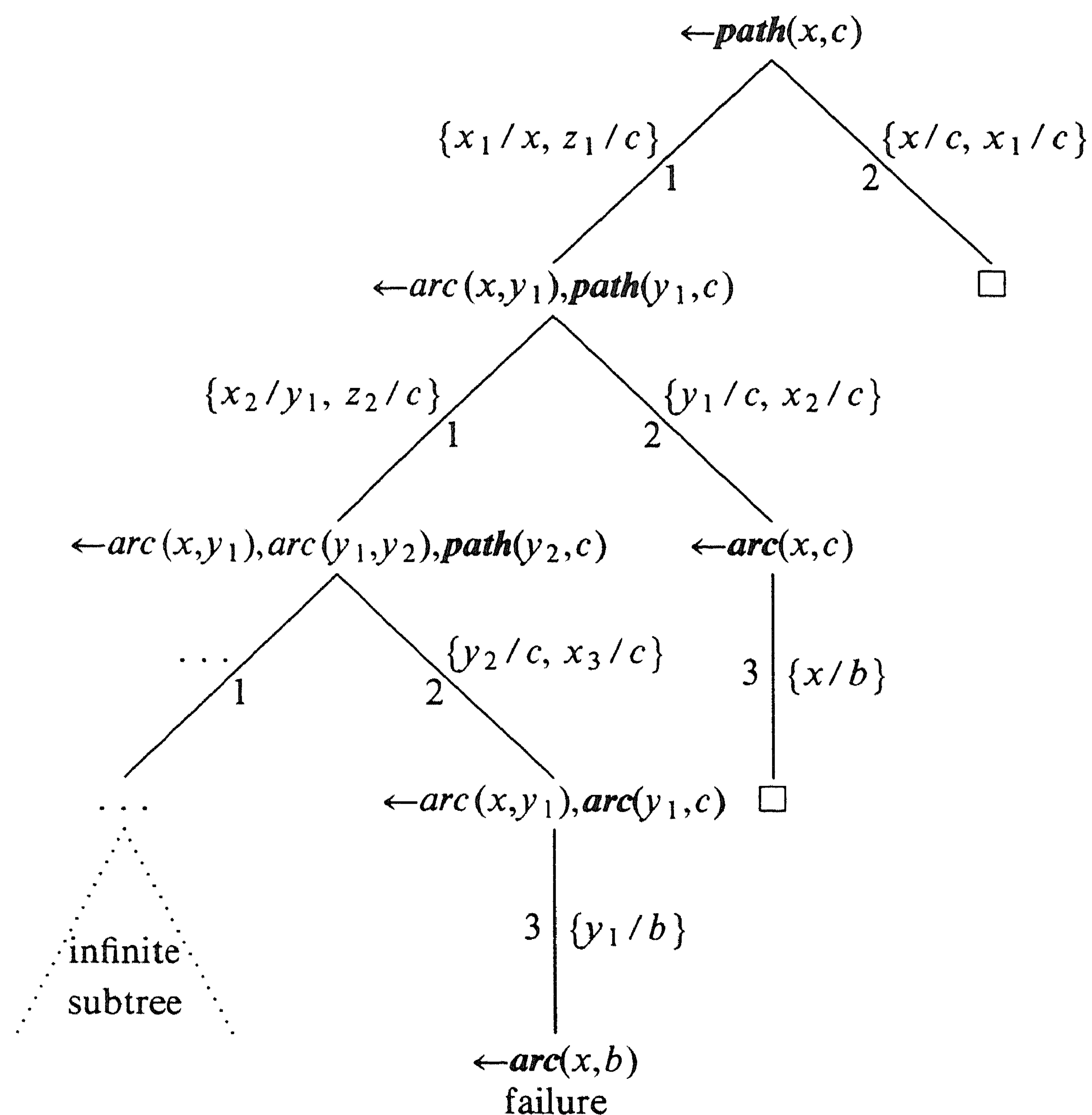

FIGURE 2

PROLOG uses a unification algorithm without a so-called *occur-check* : it unifies x with $f(x)$, whereas this unification is impossible since x occurs in $f(x)$. As a consequence the goal $\leftarrow q$ is successfully refuted by PROLOG from the logic program P consisting of $q \leftarrow p(x,x)$ and $p(x,f(x)) \leftarrow$, whereas obviously we do not have $P \models q$.

PROLOG's syntax often deviates from that of logic programming. For all but the representation of lists in PROLOG we shall ignore these deviations and stay as close as possible to the syntax of logic programming. For the representation of the *empty list* we introduce a special constant []. For the construction of non-empty lists we introduce a special binary function, similar to CONS in LISP. If L is a list and t a term, then the application of this special binary function to t and L, denoted by $[t \mid L]$, represents the list with t as first element (the *head*), followed by the list L (the *tail*). Moreover $[t_1, \ldots, t_n \mid L]$ abbreviates the list $[t_1 \mid [\cdots [t_n \mid L] \cdots]]$, provided that L is a list. Finally $[t_1, \ldots, t_n]$ abbreviates $[t_1, \ldots, t_n \mid []]$ $(n \geqslant 1)$.

4.2. In defence of PROLOG

After such severe objections, let us see whether we can do something useful with PROLOG. We shall demonstrate its use by three examples. The first two examples concern the concatenation of lists in PROLOG, with the second one nicely illustrating the power of unification. The third example (map colouring) illustrates the power of backtracking.

Consider the (classical) logic program consisting of the following two clauses.

1. $append([],z,z) \leftarrow$
2. $append([w \mid x],y,[w \mid z]) \leftarrow append(x,y,z)$

If we interpret $append(x,y,z)$ as 'the list x with the list y appended at the end equals the list z', then the two clauses state exactly what is necessary and sufficient concerning *append*. Figure 3 shows the SLD-tree for the goal $\leftarrow append([a,b],[c,d],z)$, which is successfully refuted yielding the answer substitution $\{z/[a \mid [b \mid [c,d]]], \ldots\}$. The branches of the SLD-tree are labelled by the number of the program clause used, and by the most relevant bindings of the mgu's.

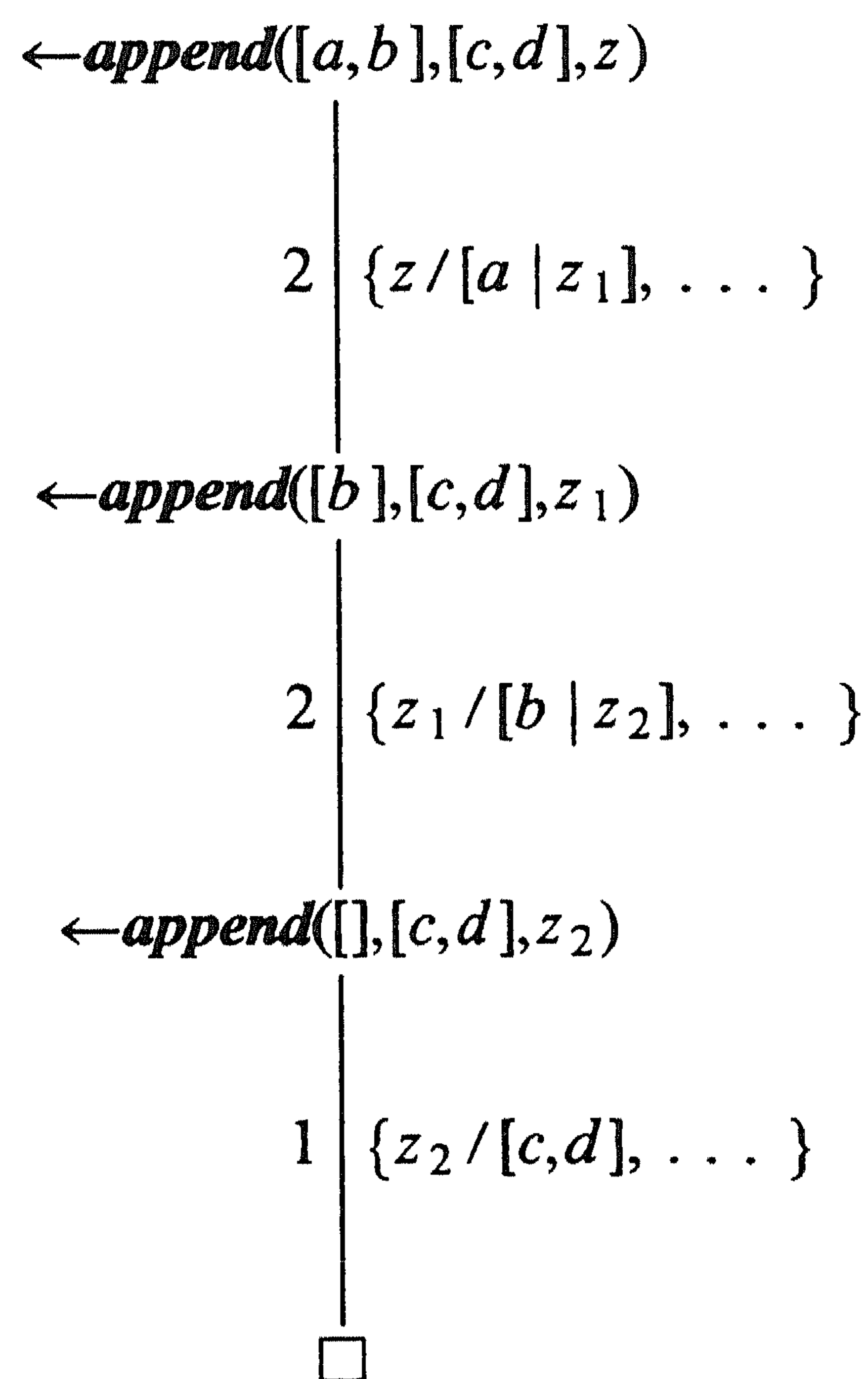

FIGURE 3

If we change the textual order of the program clauses above, then the goal $\leftarrow append([a,b],[c,d],z)$ will still be successfully refuted by PROLOG, yielding the same answer substitution. The reason is of course that in every step of the refutation procedure only one program clause applies, so that the SLD-tree consists of one single path. However, the goal $\leftarrow append(x,y,z)$ will (wrongly) *not* be refuted, although answer substitutions exist. For example $\{x/[a,b],y/[c,d],z/[a,b,c,d]\}$ is an answer substitution for the goal $\leftarrow append(x,y,z)$, but also $\{x/[],y/z\}$. The reason is that the leftmost path in the SLD-tree is infinite, so that PROLOG misses all answer substitutions, since they are all on the right of this infinite path. See Figure 4 (and recall that program clause 2 now precedes 1).

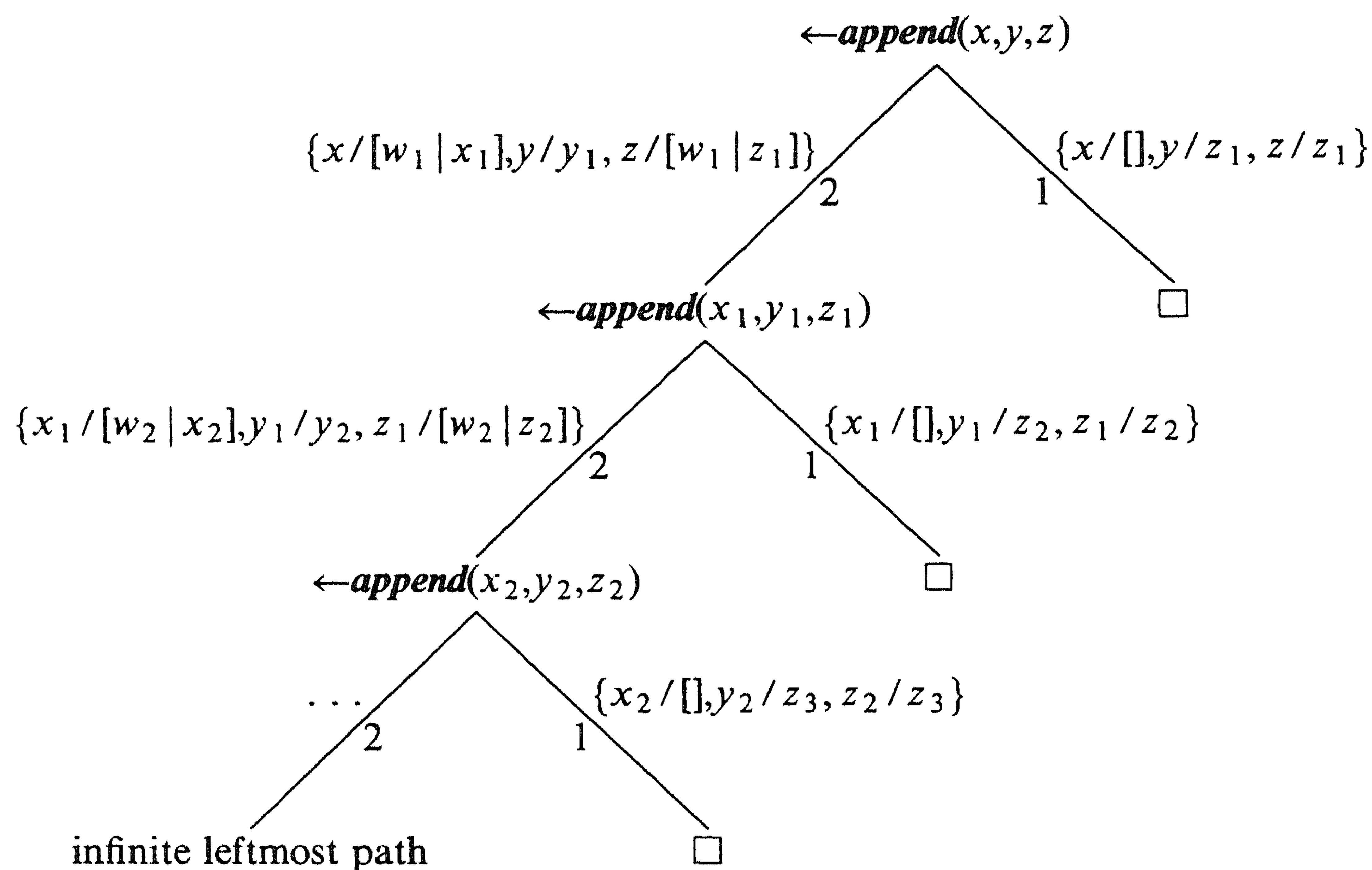

FIGURE 4

The way of appending lists such as described in the previous paragraphs is considered *naive*, since it is not efficient. It is not difficult to see that the time it takes to append lists this way is *linear* in the length of the first list. With a more clever representation of lists it is possible to append lists in *constant* time. This representation is called *difference list*, and amounts to representing a list $[t_1, \ldots, t_n]$ $(n \geqslant 1)$ by the term $[t_1, \ldots, t_n|x]-x$ ($-$ is a new binary function symbol written as infix). The empty list is represented by $x-x$. Here the key idea is that the end of the list is immediately accessible by means of the variable x. Thus the (linear) walk through a list (such as done by *append*) is avoided. Concatenation of lists can now be achieved elegantly and efficiently

by the following logic program consisting of just one clause.

$$concat(x-y, y-z, x-z)\leftarrow$$

Figure 5 shows the concatenation of $[a,b]$ and $[c,d]$, thus illustrating the power of unification (the essential binding is $z/[a,b,c,d\,|\,y]-y$).

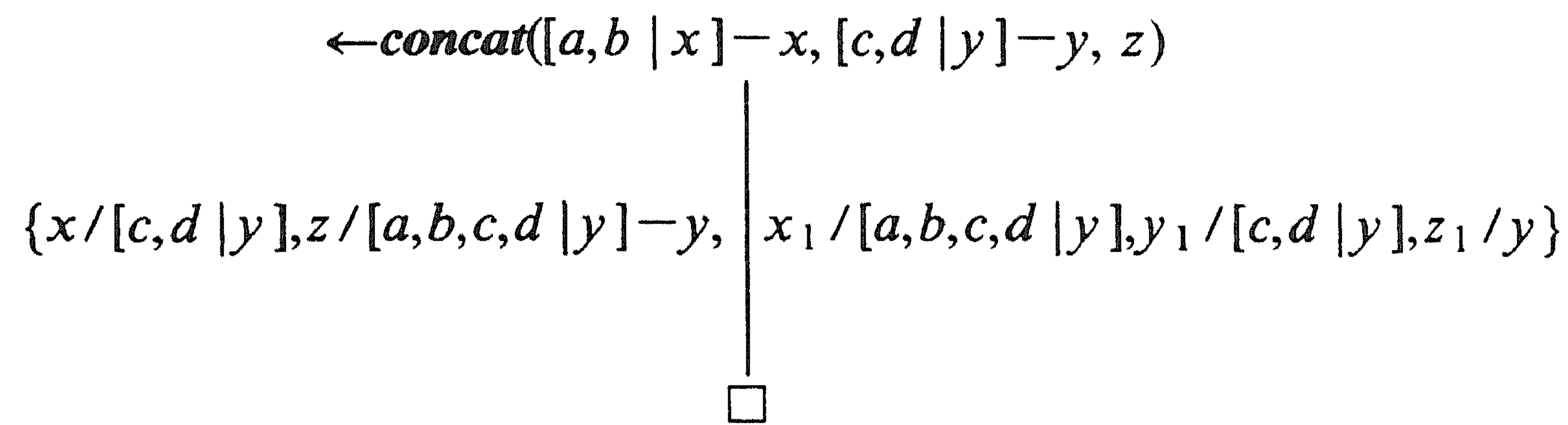

FIGURE 5

Our next example illustrates the power of backtracking in PROLOG. To this purpose we consider a classical problem: the problem of colouring a map with four different colours in such a way that no two countries with a borderline in common share the same colour. The colours are denoted by the constants *red*, *white*, *blue* and *orange*. To exclude any pretence of chauvinism, consider that part of Europe which is centered around Belgium, i.e. the well-known countries denoted by the capitals B, NL, D, L and F. A colouring of these countries is defined by binding the variables x_B, x_{NL}, x_D, x_L, x_F to the colours above. Taking the map of Europe into account, the correct solutions to the colouring problem are given by the following program clause.

1. $solution(x_{NL},x_B,x_L,x_F,x_D)\leftarrow differentlycoloured(x_{NL},x_B),$
$differentlycoloured(x_B,x_D),$
$differentlycoloured(x_D,x_L),$
$differentlycoloured(x_L,x_B),$
$differentlycoloured(x_{NL},x_D),$
$differentlycoloured(x_D,x_F),$
$differentlycoloured(x_F,x_B),$
$differentlycoloured(x_L,x_F)$

Of course the program is not yet complete, since we haven't given a proper definition of *differentlycoloured*. This will be done in the next program clause.

2. $differentlycoloured(x,y)\leftarrow colour(x),$
$colour(y),$
$notequal(x,y)$

The next four program clauses state what the colours are.

3. *colour*(*red*)←

4. *colour*(*white*)←

5. *colour*(*blue*)←

6. *colour*(*orange*)←

Finally we define in twelve (!) program clauses which colours are different. Note that in the absence of negation we do not have any other way of defining *notequal*. We shall remedy this obvious shortcoming in the next subsection.

7. *notequal*(*red*, *white*)←

8. *notequal*(*red*, *blue*)←

9. *notequal*(*red*, *orange*)←

.

.

18. *notequal*(*orange*, *blue*)←

The reader easily verifies that the successful refutation of the goal ←*solution*$(x_{NL}, x_B, x_L, x_F, x_D)$ from the logic program above yields an answer substitution which represents a correct colouring of the countries involved. The backtracking search procedure which would have to be programmed out in most other programming languages coincides with the way in which PROLOG searches for a successful refutation in the SLD-tree.

4.3. Negation as failure

The last example shows an urgent need for (at least some kind of) negation. In the present subsection we shall introduce a type of negation called *negation as failure*. First note that equality of colours in the example above could very well be defined by the following program clause.

equal(x,x)←

It so happens that two colours c and c' are different if and only if the goal ←*equal*(c,c') fails with respect to the above program clause. Negation as failure is the general rule according to which a negated atom A without variables is inferred from a program P if and only if the goal ←A fails with respect to P. (We recall that the refutation procedure can lead to success, to failure, or to infinite computation. A goal is said to *fail* if the PROLOG search in the SLD-tree *terminates in a finite number of steps* without success.) In most PROLOG systems negation as failure is built-in and negated atoms (even with variables) are allowed at the right hand sides of ← in program and goal clauses. This practice cannot be completely justified; for a discussion on the use of negation in logic programming we refer the reader to [1]. For the remaining part of this tutorial we restrict ourselves to demonstrating how,

based on the idea of negation as failure, a convenient definition of *notequal* can be implemented by means of a control primitive for pruning SLD-trees. Then the generalization to a built-in negation as failure (allowing $\neg p$ instead of defining *notp* for every predicate p which is used negatively) can easily be imagined.

PROLOG's control primitive for pruning SLD-trees is called *cut*. It is denoted by '!' and may be used on the syntactic position of an atom at the right hand side of $\leftarrow$ in program and goal clauses. The cut is difficult to explain and difficult to use. From a theoretical standpoint the cut must be considered as an aberration when (often on purpose) successes are pruned from SLD-trees; from a practical point of view the cut is very powerful, but makes programs hard to read and to debug. In order to explain how the cut works we start with an example. Assume the following two clauses are part of a logic program.

1. $p \leftarrow !, q$

2. $p \leftarrow r$

Assume moreover that, in some SLD-tree, the goals $\leftarrow p$ and $\leftarrow p'$ are just below a certain goal G, with $\leftarrow p$ left from $\leftarrow p'$. Application of program clause 1 to $\leftarrow p$ yields the goal $\leftarrow q$, while passing a cut, a fact which we indicate by attaching the label '!' to the corresponding branch of the SLD-tree. Of course the application of program clause 2 yields the goal $\leftarrow r$. Finally assume that the subtrees below $\leftarrow q$, $\leftarrow r$ and $\leftarrow p'$ are, respectively, T_q, T_r and $T_{p'}$. The situation is now as depicted in Figure 6.

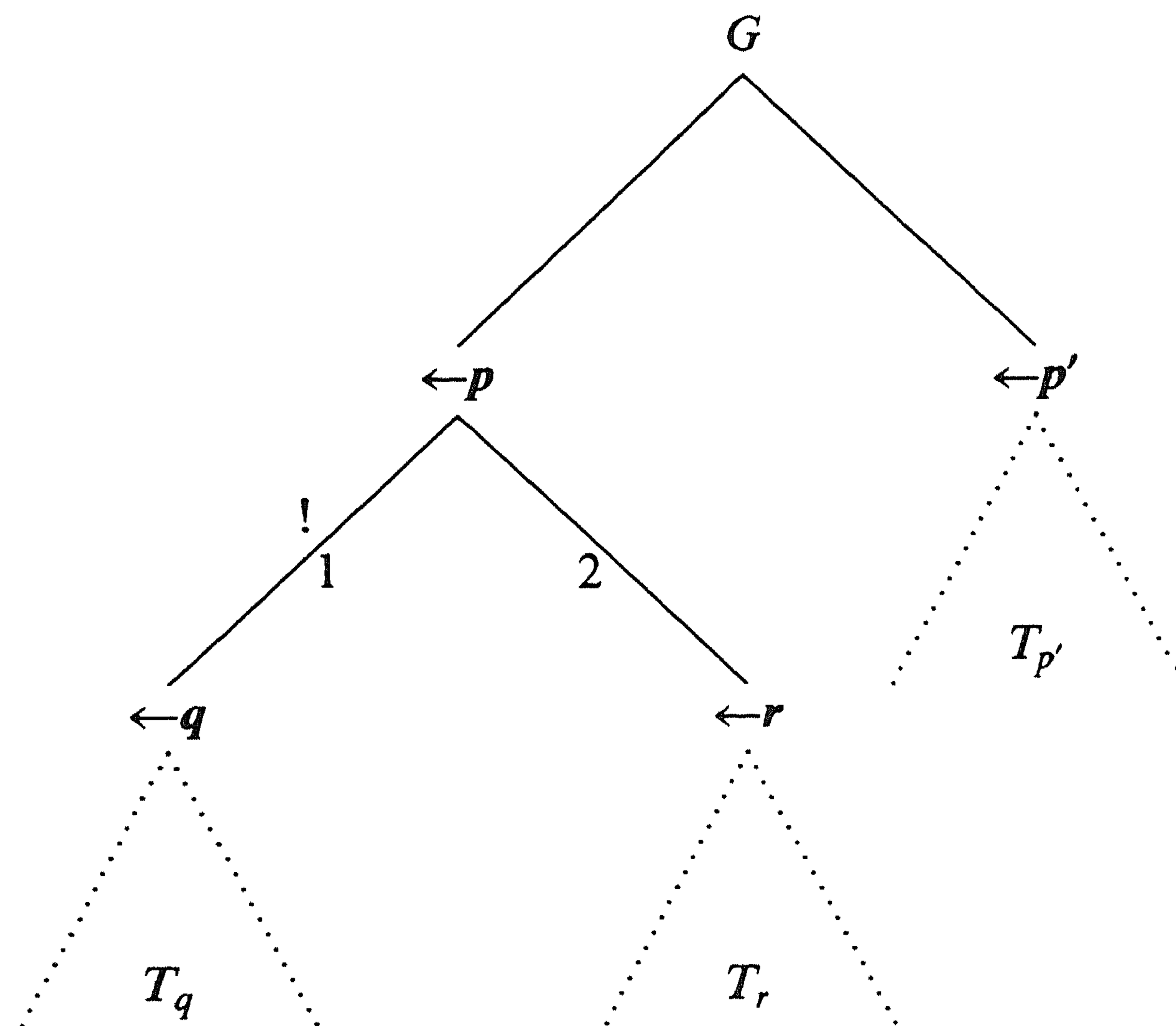

FIGURE 6

The effect of the cut is that, if the search procedure has exhausted T_q without success, then T_r is pruned and the search for success is continued in $T_{p'}$. The analogy between the cut and a forward GOTO in imperative programming languages should be evident.

Now for the (cumbersome) general case. A cut is said to be *introduced* when it first appears (not necessarily on the first atom position) in a goal. A cut is *passed* when it appears on the first atom position of a goal; the cut is then immediately deleted from the goal. In the example above the cut is introduced and passed in the same goal. In general, the effect of a cut is that, when PROLOG's search procedure backtracks from the goal where the cut was *passed*, the goal just above the goal in which the cut was *introduced* is taken to fail. Thus possible solutions in the pruned part of the SLD-tree are ignored. (If the cut is introduced in the root of the SLD-tree, then this root is taken to fail.)

The implementation of *notequal* by means of the cut is a logic program consisting of the following three program clauses.

1. $equal(x,x) \leftarrow$
2. $notequal(x,y) \leftarrow equal(x,y),$
$!,$
$fail$
3. $notequal(x,y) \leftarrow$

The atom *fail* is a predicate to which no program clause applies, so that it necessarily fails.

Consider the goal $\leftarrow notequal(red, white)$ (see Figure 7, omitting mgu's from now on). This goal succeeds due to the third program clause; the search procedure never passes the cut.

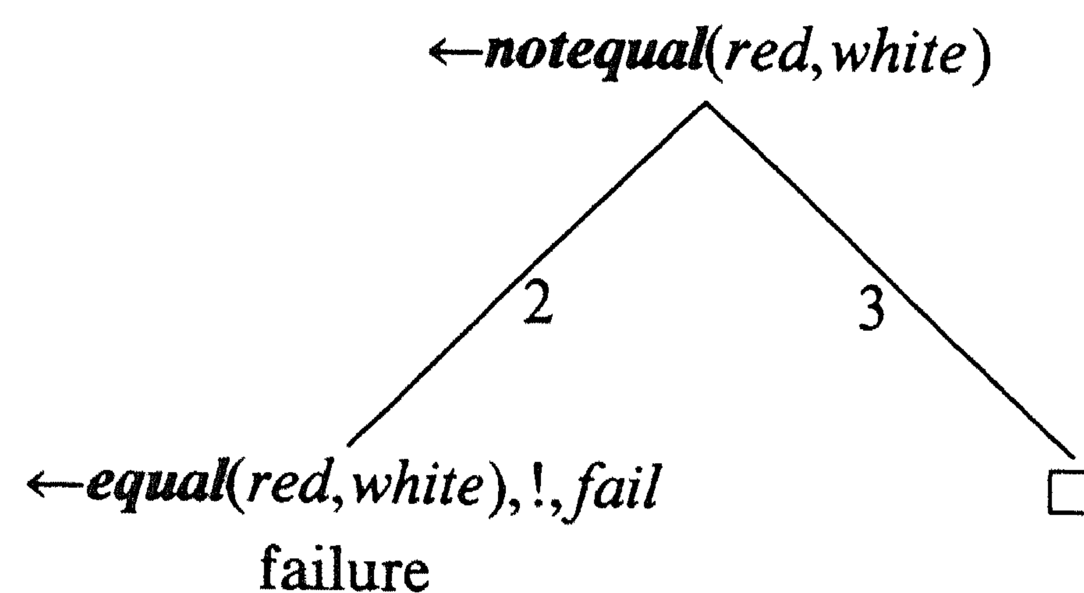

FIGURE 7

Consider the goal $\leftarrow notequal(red,red)$. The SLD-tree of this goal is depicted in Figure 8. The first program clause (a unit clause), applies to the atom

equal(*red*,*red*) and the cut is passed, but the goal fails because of *fail*. The cut is used here to prune the success due to the third program clause. This concludes our tutorial.

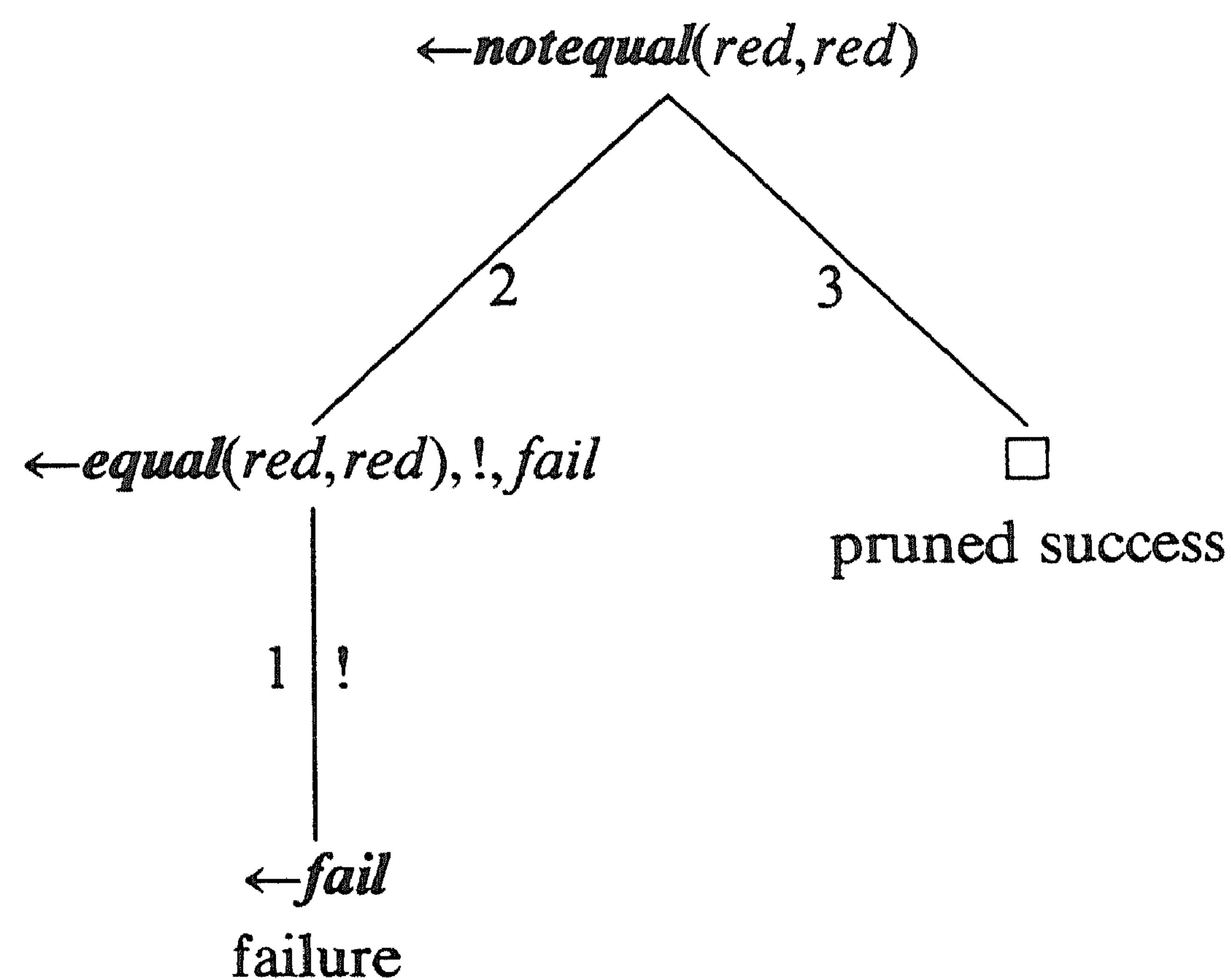

FIGURE 8

REFERENCES

1. K.R. APT (1988). *Introduction to Logic Programming*, Report CS-R8826, Centre for Mathematics and Computer Science, Amsterdam. To appear in: J. VAN LEEUWEN (editor). *Handbook of Theoretical Computer Science*, North-Holland, Amsterdam.
2. K.R. APT and M.H. VAN EMDEN (1982). Contributions to the theory of logic programming. *J. ACM 29*, 841-862.
3. I. BRATKO (1986). *PROLOG Programming for Artificial Intelligence*, Addison-Wesley, Reading (Mass.).
4. H.B. ENDERTON (1972). *A Mathematical Introduction to Logic*, Academic Press, New York.
5. G. HUET (1985). Deduction and computation. In: W. BIBEL and PH. JORRAND (editors). *Fundamentals of Artificial Intelligence*, Lecture Notes in Computer Science 232, Springer-Verlag, Berlin, 39-74.
6. R.A. KOWALSKI (1974). Predicate logic as a programming language. In: J.L. ROSENFELD (editor). *Information Processing 74*, Stockholm, North-Holland, Amsterdam, 569-574.
7. J.W. LLOYD (1987). *Foundations of Logic Programming*, Second Edition, Springer-Verlag, Berlin.
8. J.A. ROBINSON (1965). A machine-oriented logic based on the resolution principle. *J. ACM 12*, 23-41.
9. L. STERLING and E.Y. SHAPIRO (1986). *The Art of Prolog*, The MIT Press, Cambridge (Mass.).