Rewrite Systems, Pattern Matching, and Code Generation

by
Eduardo Pelegrí-Llopart

Technical Report UCB/CSD 88/423

June 9th, 1988

This technical report is an updated version[1] of a dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy
in
Computer Science
in the
Graduate Division
of the
University of California, Berkeley

Computer Science Division
Department of Electrical Engineering and Computer Sciences
College of Engineering
University of California, Bekeley
Berkeley, CA 94720, USA

[1] Minor revisions were made to proofs and notation in Chapter 5.

Rewrite Systems, Pattern Matching, and Code Generation

by
Eduardo Pelegrí-Llopart

I have made this letter longer than usual
because I lack the time to make it shorter

[Blaise Pascal [1623-1662]]

Rule seventeen:
Omit needless words!
Omit needless words!
Omit needless words!

[William Strunk Jr [1869-1946]]

Rewrite Systems, Pattern Matching, and Code Generation

by
Eduardo Pelegrí-Llopart

Abstract

Trees are convenient representations because of their hierarchical structure, which models many situations, and the ease with which they can be manipulated. A ***rewrite system*** is a collection of ***rewrite rules*** of the form $\alpha \rightarrow \beta$ where α and β are tree patterns. A rewrite system defines a transformation between trees by the repeated application of its rewrite rules.

Two research directions are pursued in this dissertation: augmenting the expressive power of individual rewrite rules by using new types of patterns, and analyzing the interaction of the rewrite rules. The dissertation contains new algorithms for linear and non-linear patterns, for a new type of non-local pattern, and for typed patterns in which the variables are restricted to tree languages.

The REACHABILITY problem for a rewrite system R is, given an input tree T and a fixed *goal* tree G, to determine whether there exists a rewrite sequence in R, rewriting T into G and, if so, to obtain one such sequence. REACHABILITY can be used to solve problems related to the mapping between concrete and abstract syntax trees, to construct a pattern matching algorithm for typed non-local patterns, and to provide algorithms for compiler code generation. A new class of rewrite system called finite ***bottom-up rewrite system*** (finite-BURS) is introduced for which the REACHABILITY problem can be solved efficiently with a table-driven algorithm.

The C-REACHABILITY problem is similar to REACHABILITY except that rewrite sequences are assigned costs, and the obtained sequence is required to have minimum cost over all candidates. If the cost of a rewrite sequence is defined as the sum of the costs of its rewrite rules, the algorithm for REACHABILITY can be modified for a subclass of finite-BURS to solve C-REACHABILITY in such a way that all cost manipulation is done at table-creation time. The subclass extends the machine grammars used by Graham and Glanville for code generation. A code generator generator based on this approach has been implemented and tested with several machine descriptions. The code generators obtained produce locally optimal code, are faster than comparable ones based on Graham-Glanville techniques, and are significantly faster than other recent proposals that manipulate costs explicitly at code generation time. Table size is comparable to the Graham-Glanville code generator.

Table of Contents

List of Figures

Acknowledgements

Many people have helped me during my stay at Berkeley. They all deserve my thanks:

To Sue Graham, for being my advisor, for emphasizing "measuring numbers", for collecting a very interesting research group, and for being picky about what gets written; to Paul Hilfinger, and John Addison, for agreeing to read this overweight dissertation; to my research group for being good friends, and for listening to the material in this dissertation one too many times; to Charlie Farnum for debugging Chapter 5.

To Peter Kessler for introducing me to juggling and for being continually amazed about everything; to Bob Ballance for dreaming PAN and sharing his dream with us; to Phil Hatcher for the puzzle; to David Chase for lending me his code for pattern matching generation; to Robert Henry for CODEGEN, UW-CODEGEN, the machine descriptions, for showing me what had to be done to finish, for reviving the hiking bug in me, and for general support.

To all the friends at Berkeley who have supported Vicki and me, especially Michael and Karen, and to the friends abroad that have kept constant despite the epistolary drought, especially to Arturo; to my parents and brothers; and, finally and most important, to Vicki O'Day for being here with me during all of this.

This research was partially sponsored by Defense Advance Research Projects Agency (DoD) Arpa Order No. 4871, monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089, Micro, by Xerox Corporation, and by Vicki O'Day.

CHAPTER 1

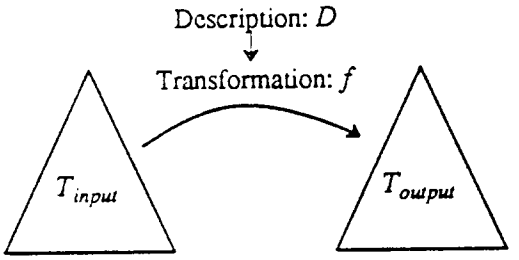
Trees and Tree Transformations

Quien a buen arbol se arrima,
buena sombra le cobija[1]

[Old Spanish Saying]

Trees are hierarchical mathematical objects. Their hierarchy makes them adequate models for many situations while their mathematical properties gives them a simple and sound basis. In addition, trees can be manipulated simply and efficiently in a traditional von Neuman computer. These attributes have made trees the preferred model for many applications. This dissertation is concerned mainly with problems related to tree transformations: mappings whose domain and image are sets of trees. The application areas that motivated this research are compilers and syntax directed editors in which the program is represented as a tree to show the hierarchical organization of its syntax or its semantics; another application area is the one referred to in the literature as technology mapping in logic synthesis systems [DGR87].

Tree transformation problems revolve around a ***description***, using some descriptive mechanism, denoting a ***transformation*** between trees in some ***input tree language*** and some ***output tree language***.



This dissertation studies several types of tree transformation problems. A first class of problems are ***notation*** problems: in most cases obtaining the description that corresponds to some desired transformation is a non-trivial task. The difficulty of the problem depends on the complexity of the transformation and the adequacy of the descriptive mechanism. In general, the descriptive mechanism has to be both expressive and natural, and it must support some reasoning on the properties of the transformation it describes. Some of the properties of the transformation that may be obtainable from the description are correctness relative to some other specification and whether the transformation is one-to-one or well defined.

Another class of tree transformation problems are ***application*** problems. There are two main varieties of application problems: ***forward*** application problems require applying the transformation to an input tree; ***backward*** application problems require applying the functional inverse of the transformation to an output tree. Since the mapping of the transformation may be, in general, many-to-many, both types of applications may return one or all of the possibilities.

[1] 'He who gets close to a good tree, will be protected by good shade'

Another class of problems are *reachability* problems. In some descriptive mechanisms, the (forward or backward) application of a transformation leaves a "trace", a justification of its existence. For example, in a transformation described as the iterated application of simpler transformations, the list of the applications provides such a trace. A reachability problem for such a descriptive mechanism consists of, given the description of the transformation, determining if a given tree can be transformed into another given tree and, if so, providing a trace for the transformation. Clearly, if the transformation is computable and maps any input tree into a single output tree, solving the reachability problem is trivial, but in general the situation may be quite complex.

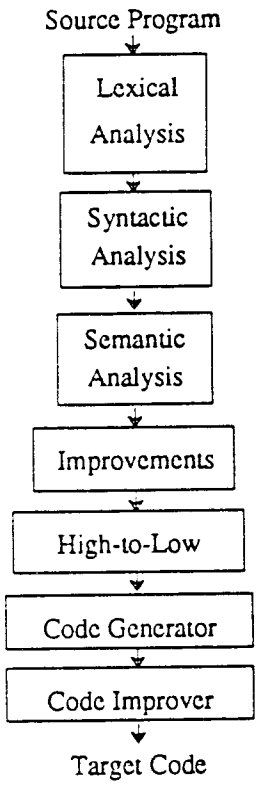
The taxonomy of problems given above only reflects the types of problems that are discussed in this dissertation and is not intended to be complete. Besides, a problem can be classified in more than once class; for instance, Chapter 7 solves application problems using reachability problems.

1.1. Some Examples of Tree Transformation Problems

Some examples of the the tree transformation problems mentioned above are the following:

Compilers

The main application of the tree transformation problems considered in this dissertation is ***compiler*** systems. Compilers are translators from programming languages to machine languages. Since this translation is a complex process, it is normally decomposed into components that can be solved more or less independently. Frequently these components are simpler translations between intermediate representations (IR) of the program, called ***phases***, and many of the IR are based on trees. Figure 1.1 shows a compiler model which decouples target machine dependency from source language dependency. (The descriptions in [Rip78] and [Joh77] approximately follow this model, however the one in [WJW75] does not). Some of the phases shown in the figure have been modeled successfully without using any of the above tree transformation problems, but others can be described as instances of those problems. Of these, this dissertation shows how some can be solved efficiently, and provides the foundation for handling others.



A Compiler Organization

Figure 1.1

The *lexical analysis* phase maps strings of characters into strings of tokens. It is frequently described by giving for each token a regular expression denoting the set of all strings of characters that map into the token. A program implementing the mapping can either be obtained automatically from the description by a program like Lex [LeS75], or can be written manually in some implementation language. Manually written lexical analyzers are frequently faster than automatically generated ones because the mapping is quite simple and all its details can be easily grasped by the programmer who can then exploit its peculiarities.

The *syntactic analysis* phase maps strings of tokens into syntax trees such that their "border" (the left-to-right concatenation of their leaves) is the string of tokens. This problem is also called the *parsing* problem and was the object of intensive research during many years. Currently there are several very successful techniques that can be used to solve this problem. These techniques are based on formal language theory [AhU73, Har78] and can be used to obtain syntax analyzers automatically from a context-free grammar describing the set of valid syntax trees. A popular example of a parser generator is Yacc [Joh78] which is based on LALR(1) parsing. Although some techniques, like top-down parsing, can be used to write syntax analyzers manually, a majority of users prefer the automatic techniques because they are more powerful, easier to use, and, in large grammars, tend to produce better results than all but the best

programmers.

The *semantic analysis* phase can be modeled as a tree transformation where the input tree is a syntax tree and the output is an *annotation* of it. The annotation makes explicit some information extracted from the syntax tree for later use and also uses this annotation to test the "static semantics". There are several techniques to describe the annotation. Attribute grammars [Knu68] have received much attention in recent years as a descriptive mechanism for this stage [Kas80, Pel80, Rai80], but are not accepted universally. A major problem in attribute grammars involves describing the distribution of information across "long distances" in the tree, a function encoded in other approaches in the notion of a symbol table. There have been other approaches to solve this problem, including Ballance's proposal for a system based on logic programming technology [Bal83]. Correct semantic analyzers generated automatically from one of the above descriptions can be obtained faster but execute less efficiently than manually written ones; current research is attempting to remove the efficiency disadvantages. This phase can be seen as a very specialized forward application problem, but it is not the main focus of this dissertation.

The next two phases are examples of forward application problems. For the purposes of this introduction, the name *improvements* includes a broad range of transformations that attempt to "improve" a program represented using some IR tree. Since the tree transformation may be quite general the description mechanism must be powerful and proving the correctness of the description becomes a non-trivial issue. The *high to low* transformation phase is responsible for the implementation of data types and control structures present in the source language using the simpler mechanisms available in the target machine. The phase can be described as a forward application problem for a tree transformation where the input and output trees use operators of different conceptual level. Currently there is no generally accepted formalism for describing these transformations and leading to efficient solutions to the forward application problems. Section 1.4 below discusses in some detail the properties required from the descriptive formalisms. Although this dissertation does not provide a complete proposal for a descriptive mechanism for these problems, it does provide the foundations on which to start building it.

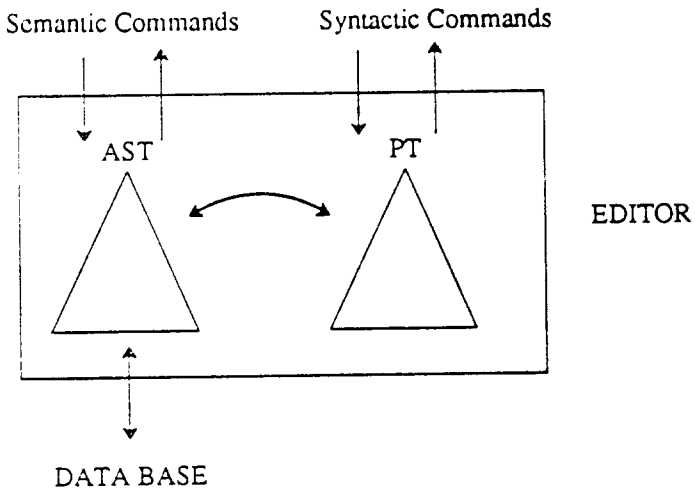
The *code generator* is a phase producing target machine code, or more properly, code for the Compiler Writer's Virtual Machine, CWVM [Hen84], from a low level IR tree. A convenient formalization for this phase is based on constructing a transformation encoding the possible sequences of instructions in the CWVM in such a way that an input tree can be rewritten into a fixed "goal" tree if and only if there is a sequence of instructions implementing the input tree. Solving the reachability problem then produces a trace that can be used to extract the desired instruction sequence. The reachability problem can be modified to incorporate a cost metric leading to instruction sequences with minimum cost. This approach has been explored extensively in recent years [AhJ76, GaF82, GlG78, Hen84]. It is currently possible to automatically generate code generators that rival the quality of the best manually written code generators. This dissertation provides new and very efficient solutions to this problem (Chapters 6 and 8).

The *code improver* phase transforms target machine code into "improved" target machine code. It is best described as a string-to-string transformation phase, but some researchers [Kes84] use tree transformations when constructing the code improver. None of the different proposals for this problem has gained a definitive advantage over the others; a substantial part of the problem here is deciding exactly what has to be done in this phase and what is done elsewhere, for instance, in the code generation phase. Again, although this problem is outside the scope of this dissertation, some approaches to it may use the foundations.

There are other important components of a compiler, notably the register manager [Mck84], which interact with the phases mentioned above. None of them will be considered here except in the context of the implementation of the code generator reported in Chapter 8.

Language-Based Editors

Another example of a program manipulating programs is a language-based editor: an editor that "understands" the program that it manipulates. Typically such an editor will keep an internal representation of the program being edited in the form of some variation of a tree and the editor commands will modify that representation. Figure 1.2 is an schematic representation of a language-based editor where there are two internal representations of the program: the ***parse tree*** (PT) represents faithfully the program according to the context-free grammar of the language, while the *abstract syntax tree* (AST) only represents the "deep structure" of the program.



Language Based Editor

Figure 1.2

Both PT and AST are useful in the editor. The PT is used while parsing pieces of programs entered as text, while the AST is a more compact form and the preferred form to perform program modifications or queries that require knowledge of the semantic structure of the program. If these were the only requirements, one could avoid ever constructing the PT and apply the PT→AST transformation on-the-fly as the input program is being parsed. But the PT is also useful to produce pretty-printed versions of programs that have been modified by changes to the AST, and to allow the application of incremental parsing algorithms that work only on the PT. One possibility is to keep both representations in the editor. A better one is to store only the AST and to regenerate the PT from it as an inverse application problem. Since the PT → AST mapping is many-to-one, its inversion has to choose one of the possible parse trees.

A language-based editor contains other components that are outside the scope of this dissertation. See [Bal83] for more details.

Technology Mapping in Logic Synthesis

Tree transformation problems have other applications beyond just programs manipulating programs. One recent example the area called "technology mapping in logic synthesis". Logic synthesis starts with a set of boolean equations describing the desired relation between a

collection of output signals and a collection of input signals and attempts to obtain an optimized implementation of it for some particular technology. One approach that has been explored recently is to divide this problem into two different parts. The first part tries to optimize the set of equations independently of the final target technology. The particular method used in [BDK86] involves approximating the problem in Z_2 (integers modulo 2) instead of the boolean field, minimizing the equations there, and then "patching up" the solution to get a result in the booleans (this essentially involves adding the results that depend on idempotency).

The result after this first stage is a "better" set of equations. This set of equations is then mapped into the desired technology. The problem can be described as one of finding a covering of the directed acyclic graph (dags) represented by the equations using the graphs that correspond to the operations available in the desired technology and such that the "cost" of the covering is minimal [DGR87]. Solving this problem for dags is expensive, but, if dags are approximated with trees by "pulling out" common subtrees, the corresponding problem is another reachability problem with cost information for a tree transformation, and can be solved very efficiently using the results of Chapter 6.

1.2. Scope of this Dissertation

The focus of this dissertation is on one class of descriptive mechanisms, providing some research on its foundations and details on a particular subclass that has a useful range of applications. The class of descriptive mechanisms is a variation of term rewrite systems called tree rewrite systems; they are described informally below in Section 1.5, and formally in Chapter 2. Fundamental problems related to this class are explored in several chapters, including Chapters 3, 4, and 7. The subclass explored in some detail is called the bottom-up rewrite systems (BURS), described in Chapter 5. This dissertation shows how to solve efficiently some forward application, backward application, and reachability problems using BURS, and also reports on a complete experiment where a code generator was constructed by providing a solver for a reachability problem for transformations based on BURS (Chapters 6 and 8). Although not implemented, the dissertation also shows how to deal with the forward and backward applications of the PT→AST transformation, and the transformations appearing in technology mapping in logic synthesis (Chapter 7). The high-to-low transformations seem accessible from the results of this dissertation with some extra effort; they would use the foundational results provided in the dissertation. The transformations appearing in the improvement phase are quite more complex and seem to require additional research beyond the results described in this dissertation.

1.3. General Considerations for Solving Tree Transformation Problems

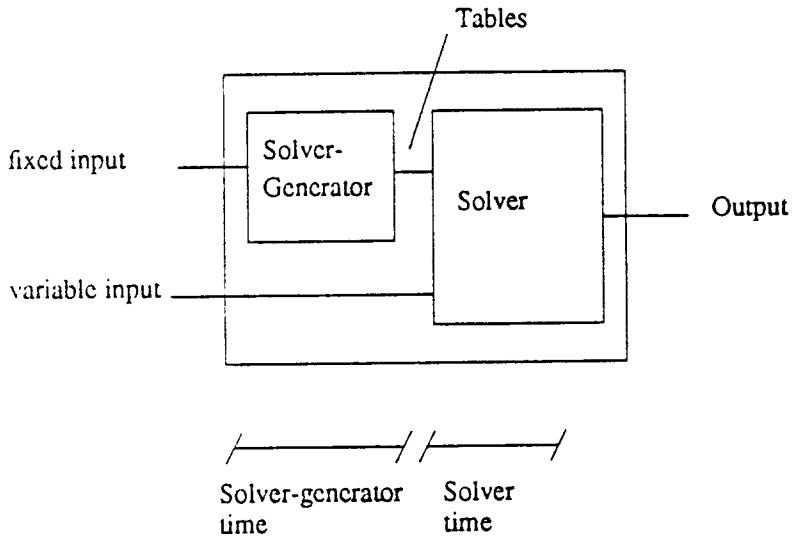
There are some considerations that appear frequently when solving any of the tree transformation problems listed above.

1.3.1. Solvers and Solver Generators

The traditional way to measure the performance of some algorithm is by its time and space complexity. Regardless of whether a worst-case or an average case behavior is used, these measures treat identically all the arguments defining a problem. This approach is inadequate for the problems studied in this dissertation. For example, a forward application problem has two inputs, a tree T_{input} and a description D, but typically D is "fixed" and T_{input} is "variable". It is not accurate to treat D and T_{input} in the same way when quantifying the difficulty of the problem.

The solutions to the problems considered in this dissertation can be separated in two different phases: a *solver generator phase*, and a *solver phase*. Their interaction is shown in Figure 1.3. Some of the input arguments are passed to the solver generator which analyzes them and produces a specification that is probably larger than the input but easier to handle. This specification is called a "table" although the particular details of the encoding may vary. The

solver receives the rest of the input specification and the processed tables and solves the desired problem.



The Solver and Solver Generator Approach

Figure 1.3

Normally, the execution time of the solver will be much smaller that that of the solver generator. In such an arrangement the solver phase may have a linear time worst-case behavior, even if the complete problem is NP-complete. If the complete problem is NP-complete, the time spent in the solver generator *plus* the time spent in the solver must be exponential on the combined input size, at least for some inputs[2]. In practice, however, many problems that are NP-complete show their extreme behavior only in some infrequently occurring sets of inputs. One example of this paradigm is optimal plan assignment in attribute grammars [Pel], where the preprocessing phase can be exponential on the description of the attribute grammar but the actual assignment can be done in linear time.

For most of the applications considered in this dissertation, the size of the tables generated by the solver generator is a more important measure than the time spent executing the solver generator. An application, not covered here, where this is not true is to algorithms in which the description (and the transformation) is continuously changing. This is the case of the Knuth-Bendix completion algorithms [KnB70], which rely on a pattern matching algorithm, but add patterns dynamically to the pattern set. Chapter 3 studies the problem of linear pattern matching. Some of the algorithms presented there could be used for Knuth-Bendix completion either because they have a fast table construction algorithm or because their tables can be modified incrementally, but this dissertation does not explore the issue any further.

[2] Unless P ≡ NP, of course.

Other details of how the problem is used may be of importance in measuring the efficiency of an algorithm. An example is when the matching algorithm is part of a larger match-rewrite cycle where a tree is alternatively analyzed for matches and modified. In this case it would be advantageous to have an algorithm capable of re-using parts of previous computations.

1.3.2. Bottom-Up and Top-Down Paradigms

Many tree transformation problems can be solved by defining a notion of a state associated with a node characterizing the "interaction" between the transformation and either the input or the output tree. There are two basic approaches to this characterization named after the order in which the states of a tree can be computed: "bottom-up" and "top-down". In a bottom-up approach, the state of a node N depends only on those of the descendants of N. In a top-down approach, the state of a node N depends only on those of the ancestors of N. In some cases it is convenient to consider states with more complex dependencies. Bottom-up characterizations are more powerful than top-down characterizations in many applications. For instance, Section 2.2 shows that bottom-up tree automata, a tree recognition device, are more powerful than top-down tree automata.

This dissertation uses two classes of states: most states are bottom-up states, but some notions of state can be described as computed in a bottom-up pass followed by a top-down pass. In all notions, the objective is to limit the amount of information present in the states to reduce the cost of computing them. If, in addition, the set of all possible states that may characterize an (input or output) tree can be computed at solver generation time and is finite, then the result of the operations involved in computing the state can be precomputed and stored into a table and the state can then be computed very efficiently at solving time by replacing all computation with simple table lookups. Some of the nicest results of this dissertation involve showing how to encode in a (relatively small) finite set of states information that initially would seem to require an infinite number of states.

1.3.3. Syntax and Semantics

A final issue is that of *syntax* vs *semantics*. A tree used to represent some object will normally contain a significant amount of information. This information can be classified as being either *syntactic* or *semantic*. The first one can be said to define the "structure" of the tree, the second, to provide a "decoration" of the tree. In some sense, it would be possible to move everything from one class to the other. One extreme corresponds to a single node decorated with a very complex semantic data value, the other to a large complex tree with no decoration at all. The distinction between one and the other is normally done based on the cardinality of the domains involved: small domains lead to syntactic information, large domains to semantic information. Infinite domains must be encoded semantically, with the exception of the child relationship. Many algorithms treat differently both types of attributes, being capable of extracting more information from syntactic attributes than from semantic ones. Encoding a value syntactically normally leads to larger tables. Through out this dissertation, syntactic information will be used to direct the transformations, and semantic information will be used only when necessary.

1.4. Describing Tree Transformations

The difficulty of solving the different tree transformation problems depends very strongly on the descriptive mechanism used.

The considerations for notational problems are similar to those present in the design of programming languages. A first requirement is one of ***expressiveness***: every tree transformation in the intended application domain must be expressible in the mechanism, The mechanism should also lead to an efficient solution to whatever problem is solved using it: what ***efficient*** means frequently ends up being "at least as fast as a good hand-coded solution to the problem". Clearly

these requirements are inter-dependent; for example, increasing the expressiveness of the mechanism is likely to lead to less efficient implementations.

A much more fuzzy requirement is that the descriptive mechanism should model the "natural" notions present in the problem domain. Most mechanisms describe a complex tree transformation as a composition of basic transformations, which are then described in two parts: a ***condition***, indicating when it applies, and an ***action***, indicating what it does. A condition can be called ***local*** when it depends on a specific contiguous portion of the tree; similarly, an action can be called ***local*** if it affects such a portion of the tree. Otherwise, conditions and actions are called ***non-local***[3]. For instance a transformation that takes a single node labeled "foobar" and replaces it by a subtree containing nodes labeled "foo" and "bar" is local, but a transformation depending on the presence of a node "foo" with a descendant (at an imprecise location) "bar" is non-local. Most individual tree transformations in compiler phases are local but some are non-local, either in the conditions or in the transformations. Ideally a descriptive mechanism should support efficiently both types of transformations.

There are two main composition mechanisms used in descriptive mechanisms. In a ***parallel composition*** several transformations are applied to the same input tree and the results are combined by some tree function to produce the final result. In a ***serial composition***, the result of one transformation is used as the input to a second one; it can be applied just ***once***, or it can be ***iterated*** until some condition is met.

The descriptive mechanisms for tree transformations published in the literature can be classified in four groups: operational, tree automata-based, attribute grammar-based, and term rewrite-based, with some mechanisms having characteristics of more than one group.

1.4.1. Operational Descriptions

Operational descriptions can also be called "description by implementation". This descriptive mechanism is, unfortunately, still the one used in the majority of the compilers. Its main advantage is, supposedly, one of expressibility: if the implementation language is reasonable, any computable function will be expressible. It lacks most other desirable properties.

One disadvantage of the technique is that by its nature, such a description is suitable only for forward application problems and all the other tree transformation problems have to be mapped into them. For example, if the problem is more naturally described as an inverse application, the user must determine how to invert the transformation and encode the inverse transformation. This increases the effort required to produce a solution to the desired tree transformation problem. Eventually, proving anything about the problem being solved corresponds to proving properties about its solving program: termination corresponds to the halting problem; correctness is meaningless unless combined with some other descriptive method; and deciding whether the transformation is many-to-one is difficult. Only showing that the transformation is not one-to-many is easy, as a consequence of the determinism of most implementation languages.

Another shortcoming of the technique is that small changes to the specification of the problem may require large changes to the implementation. Since the problem specification tend to change very frequently, this disadvantage increases very substantially the effort required to obtain a solution. Yet another disadvantage is that, in many types of problems, a truly efficient solution may require exhaustive and quite complete analysis of the transformation, which, for the above

[3] The notions of local and non-local are very fuzzy, but I do not attempt to formalize them since they are only used very informally. For the descriptive mechanism used in this dissertation locality corresponds to untyped N-patterns, while non-locality can be described using either typed N-patterns or X-patterns; see Chapter 2 for the formal definitions.

reasons, a programmer may not be able to provide. A final disadvantage is that related tree transformation problems will lead to unrelated solving programs and maintaining consistency may be non-trivial.

There are many examples of the operational approach to the description of tree transformations, most of them undocumented in the research journals. Those that are documented are normally build around a, frequently narrow, model for the specification of the transformation which effectively leads to some type of higher-level descriptive mechanism. [WJW75] is an example including transformations related to program improvement and code generation. Code generation was particularly prone to this approach, see, for instance, [Wir71]. This dissertation, along with other recent research, shows that a code generator based on a formalized tree transformation technique can be very successful (see Chapter 8).

1.4.2. Tree Automata

Tree automata and tree transducers (Section 2.2) [Eng75, Tha75], are the generalization of word automata to tree domains. There are many variations of the concept with different expressive power. Normally these descriptive mechanisms are used as abstract devices to characterize classes of expressive power. As such, they tend to be limited in the support that they provide for the "natural" description of transformations and in rendering an efficient implementation. They have some advantages, though. Some automata classes are closed under some operations, like composition, union, complementation, etc. This means that a transformation may be described as the combination of independent subtransformations. If the closure proof is constructive, it is possible to build a system that will take the transformations and obtain their composition. Another advantage of tree automata and transducers is that normally they can be provided with induction principles that can be used to prove properties of the transformations they define.

There are two main classes of tree automata, depending on how strings are generalized to trees: top down or bottom up. Both classes have limited locality in specifying both conditions and actions. [Eng75] provides an extension of top down automata with some global conditions. Tree automata tend to be too low level to be proposed as direct descriptive mechanisms, and I know of no proposal to use them unmodified, but they can be used as the foundations for other mechanisms. Section 2.2 presents tree automata in more detail.

1.4.3. Attribute Grammars

Attribute grammars [Knu68, Rai80] are a descriptive mechanism to annotate a ***parse tree***. A parse tree is composed of instances of terminals and nonterminals, the nodes of the tree, and instances of productions relating these nodes. An attribute grammar associates a set of symbols, the ***attributes***, with each node, and a set of equations, the ***semantic equations***, to each production. Each semantic equation is of the form $a=f(B)$, where a is an attribute, B a set of attributes, and f is called the *semantic function*. The *evaluation* of a parse tree using an attribute grammar is an assignment of values to the attributes satisfying the set of equations associated with the parse tree. In traditional attribute grammars, the set of equations are required to have an acyclic dependency graph which, when linearized, can then be used to find the values of the attributes by a process of "substitute and evaluate". The implementation need not follow this naïve algorithm and many alternatives have been proposed and investigated [Kas80, Pel80].

A set of labeled trees is called *local* (Section 2.2) if it is the set of parse trees for some context-free grammar G. A function with a local set as a domain can be defined by using an attribute grammar and selecting one of the attributes associated with the root of the parse tree as the value of the transformation. To define a tree transformation from a non-local set, the description writer must find a local set L′ which includes L and define the transformation on L′ instead of on L.

The straightforward way to define tree transformations using attribute grammars is to use attributes and semantic functions over tree domains (for instance, [NMS83]). Such a mechanism inherits many properties from traditional attribute grammars. Since the semantic functions are unrestricted, attribute grammars are a completely general descriptive mechanism: any computation can be expressed trivially by collecting the input tree into an attribute of the root and applying there a function denoting the desired computation. This solution is not really useful since it pushes the problem to a different place without solving it. The advantages of attribute grammars come from using simple semantic functions and relying on the structure of the parse tree to combine them into more complex transformations. [Kam3] presents some results on the expressive power of a particular version of attribute grammars, but few other results are known.

Probably the biggest disadvantage of attribute grammars is that they are "too local". Since semantic functions are associated with productions in the parse tree, attribute grammars limit their locality to a production. When the problem requires a larger local context, auxiliary attributes and semantic functions must be defined. These attributes will collect the information in the local context and transmit it to the desired node where it can be used. The problem with this approach is that it separates single transformations into several constructs, increasing the difficulty of understanding and implementing the description correctly. When the transformations are non-local, the use of auxiliary attributes cannot be avoided, and since the propagation needs to be done at each point in the path, many rules are involved.

Another problem is that since the number of attributes that are assigned to each node is fixed by the type of the node, it is frequently necessary to collect information into a larger attribute (a frequent example is the "symbol table") which is then moved to the places where the information may be required. This reduces the clarity of the description and may also reduce the efficiency of solutions to problems based on it.

Attribute grammars have a flexible parallel composition mechanism: at any non-terminal any number of attributes can be combined to obtain a new one. Serial composition is not straightforward because the result of the transformation is not available directly in the original tree. The lack of iteration forces the use of complex semantic functions in some cases where they could be avoided. For example, if a transformation is to be invoked under some particular context, the attribute grammar must check for the context, not only in the initial parse tree, but also in the intermediate tree that will be obtained by the application of successive transformations. Keeping track of these changes may be very complex.

Attribute grammars are at their best in describing how to annotate parse trees, for instance in a semantic analysis phase. There have been some proposals for proof methodologies for attribute grammars [Der83] based on proving local properties at each of the productions of the grammar and then proving properties on the interactions between the productions. The applicability of this approach depends on the complexity of the semantic functions in the attribute grammar.

The main obstacle to efficient implementations of the transformations is, again, dealing with non-local transformations where values must be pushed around from one place of the tree to the other. In general, since tree attributes are potentially large, it is particularly important to optimize the copy and allocation of the attribute values [Rai81]. On the other hand, one advantage of attribute grammars is that all the dependencies are explicit and they can be analyzed statically to select the best possible evaluation order [Pel80].

Some of the disadvantages of the attribute grammars can be overcome by building more specific mechanisms on top of them. One such proposal is attribute coupled grammars.

Attribute Coupled Grammars

Attribute coupled grammars [GaG84] are a mechanism based on attribute grammars where *expressive power* is traded for *control* over the result of the transformation, effectively providing automatic proofs of the form of the transformed tree. Attribute coupled grammars have as

domain and co-domain local sets which are explicitly described in the attribute coupled grammars as context-free grammars. The attributes are divided into "syntactically valued" and "semantically valued" attributes. Semantically valued attributes are normal attributes; syntactically valued attributes are tree-valued attributes which can have as values only (sub)parse trees of the output grammar and are strongly typed by their root operator. The result of the transformation is a syntactic attribute of the root. The semantic functions of syntactic attributes are constructors of the output grammar that may depend on the value of semantic attributes.

Attribute coupled grammars are closed under composition: there exists a computable function that will assign to two attribute coupled grammars another attribute coupled grammar that describes the (functional) composition of the transformations described by the first two. This allows one to trade description size against efficiency in the implementation of the transformation.

This gain in control is obtained at the loss of expressive power. For example it is impossible to describe in attribute coupled grammars the replacement of all "uses" in a tree by their corresponding "definitions" because each definition would have to be propagated as a single tree and this cannot be done in general because the number of attributes is fixed. In contrast, in an unrestricted attribute grammar all the definitions can be bundled together in a single attribute which can then be used in the desired places. Intuitively, attribute coupled grammars have restrictions on the generative portion of the mechanism, while the analytic portion is largely unrestricted.

In summary, although attribute grammars and attribute coupled grammars are quite adequate for some applications, they are lacking for others. The two main disadvantages are the lack of non-local facilities, and the lack of serial composition of individual transformations. The first disadvantage has been attacked by several researchers ([Hoo86, RMT86], but not very successfully in my opinion), but the second is implicit in the method.

1.4.4. Term Rewrite Systems

Term rewrite systems have advantages and disadvantages complementary to those of attribute grammars: the basic operation is serial iterative composition but there is no parallel composition.

Term rewrite systems [Chu41, Huc80] are used to describe transformations on ***terms*** over operators. Since there is an equivalence between terms over an alphabet and labeled trees where leaves correspond to 0-ary operators and internal nodes correspond to $n \geq 0$-ary operators, a term rewrite system can be used to describe tree transformations. Term rewrite systems are the basis for the descriptive mechanism used throughout this dissertation.

A pattern over an alphabet is a term on the alphabet extended with new 0-ary operators called *variables*[4]. A term rewrite system is a collection of pairs of patterns called term rewrite rules. The two patterns are called the ***input pattern*** and the ***output pattern***, and the variables used in the output pattern are required to be a subset of those used in the input pattern. A pattern matches at a node if there is a ***substitution*** for the variables that makes the pattern identical to the subtree rooted by the node. A rewrite rule defines a tree transformation: an input tree is transformed into an output tree by replacing the portion matched by the input pattern by a new tree obtained from the substitution and the output pattern.

The basic component of a term rewrite system is iterative serial composition. A rewrite system transforms an input tree by repeatedly applying rewrite rules until a tree is obtained where

[4] Section 2.3 defines more formally all the concepts mentioned in this section; in particular, these variables are extended to allow also for non-zero arities.

no rule applies. Like many tree automata, but unlike attribute grammars, tree-to-tree grammars are potentially non-deterministic in their output since several subtrees of a tree may match one or more rewrite rules. Traditional research in term rewrite systems is concerned with determining if a given system defines a function or not. This can be proved by a number of techniques, many based on showing that two properties hold: confluency and termination. Most of the proof techniques for termination are formalizations based on well-founded sequences [Moz83], while there are several local properties which imply confluence [HuO80]. Since transformational grammars have the expressive power of a Turing Machine it comes as no surprise that, in general, confluency and termination are undecidable.

Local conditions and actions are described in the mechanism by simply adding a rewrite rule with left term equal to the local condition and right term equal to the desired result. In the absence of additional requirements, like order of application, this description is the only one needed and is very natural. On the other hand, non-local transformations are not supported directly and must be simulated by sets of rewrite rules. Non-local conditions are described in an awkward way since the only way to transmit information is indirectly through modifications of the tree. This means that what should be a simple process of gathering contextual information is described as a complex process, which not only initially changes the contents of the tree, but also has to return it later to its original state. The technique employed to describe non-local actions is similar.

Similar complex interactions are needed to implement non-iterated serial composition and parallel composition. This style of programming is unnatural and very unstable: small changes to the set of rewrite rules can cause large changes to the behavior of the system. This unstability has been observed in production systems, a computational mechanism based on an organization similar to term rewrite systems [DaK77, For79, KDR78].

The implementation of term rewrite systems is based on tree pattern matching algorithms [HoO82, KMR72] to detect the rewrite rules applicable on the subject tree. The pattern matching algorithms employed should deal efficiently with changes to the tree as rewrite rules are applied.

The main advantages of term rewrite systems are its support of iterated serial composition, providing both expressive power and descriptive convenience; and the support of local transformations. Unfortunately, when the basic transformations cannot be described as a single rewrite rule, the complexity of the interaction increases radically.

Another disadvantage of term rewrite systems is the inability to control the order of application of rules. In some cases the rewrite rules are designed to be evaluated in some particular order, and enforcing that order with the standard mechanism may be very laborious; transformational grammars are one attempt to alleviate this problem.

Transformational Grammars

Transformational grammars were originally used in computational linguistics; DeRemer and Kron, [DeR69, Kro75] introduced the use of transformational grammars to describe transformations in compiler systems. Transformational grammars are like term rewrite systems with some extra control. First, if there are two different rules which apply at some node and one of them has an input pattern more "specific" than the other, then the more specific rule will be applied. Second, there is a "global" rule that is used to choose between several nodes at which a rule is applicable. The global rules presented in [Kro75] are "top-down" and "bottom-up". In both cases the rule does not define a total order on the nodes of the tree and for some nodes the selection between rules will still be random, but confluency can be proved for some classes of transformation grammars [Kro75].

There have been several systems that have been implemented based on variations of transformational grammars. One such system is Bonsai [Wul81], which can be described as being "pattern-driven". A transformation in Bonsai is specified as a collection of rules, each rule

having a condition and an action. The condition is a tree pattern extended with optional invocations to generic semantic routines. The action is just a routine in some implementation language that will return a tree value to replace the matched subtree. As in transformational grammars, there is a global specification to indicate the order in which the nodes are going to be tested. Within a node the rules are tested in order and, if the condition of a rule is true, then its action is invoked. The order in which the nodes are visited can be modified using special actions.

The main advantage of Bonsai is the expressiveness of the technique: when needed one can always escape to a complex action routine to produce the replacement tree. This is also its new disadvantage: since it is impossible to predict the interaction of the different rewrite rules without understanding the action routines this descriptive mechanism can be used only to solve forward application and reachability problems. Also in some cases the implementation may suffer, since there is less information available to the solver generator.

Twig [Tji] is a descriptive mechanism that has the normal rewrite rules, plus rules similar to those in Bonsai and additional cost information that is used to select the rules to apply. Twig was designed for generating code generators and is discussed again in Chapter 9, where it is compared against the solution presented in Chapter 8.

Attributed Transformational Grammars

Attributed transformational grammars [GMW80, MWW84] are an attempt to combine the strengths of attribute grammars with those of transformational grammars into a practical tool for the specification of compiler phases. The interaction of the attribute grammars and the transformational grammar has two facets. First, input patterns in the term rewrite rules can be extended with an arbitrary predicate on attributes of the non-terminals in the input tree; the predicate must be true and the input pattern must match for the rewrite rule to be applied. Second, after an application of a transformation rule, attributes may be recomputed according to new semantic functions described in the transformation rule. The re-evaluation may involve a larger context than the one directly affected by the transformation. A direct consequence of the characteristics of the interaction is that the domain of the tree transformation must be a local set which is described explicitly with a context-free grammar.

The evaluation of the rewrite rule is controlled as in transformational grammars, with a general traversal order. The attributed transformational grammar papers [GMW80, MWW84] also contain a proposal for non-local conditions called combined attributed transformations, and an abstraction mechanism. A combined attributed transformation ties together a local rule with a global rule connected through a connector node which normally is an attribute explicitly updated through the description specification. When the conditions for the local rule are satisfied, the conditions on the global rule are checked and, if valid, both rules are performed. The global rule can then access the information in the local rule to perform non-local actions. The abstraction mechanism can abstract a set of local and global rules, attribute reevaluation, and traversal control into a transformation unit which can later be invoked.

Attributed transformational grammars are a very interesting proposal. The expressive power and the description of local transformations is unchanged from transformational grammars, but non-local conditions can now be described through attributes which describe the condition, while non-local actions use combined attributed transformations. The abstraction mechanism also provides new flexibility in the composition facilities allowing non-iterated composition. As in transformational grammars, there is still no parallel composition.

Proving properties is simplified somewhat by the presence of procedures which make it possible to decompose the problem into smaller and simpler problems. Non-local transformations can be analyzed, at least partially, using techniques from attribute grammars. [GMW80] provides no proof methodology for combined attributed transformations.

The efficient implementation of attributed transformational grammars is more complex than for transformational grammars. Finding the set of transformations which are applicable to some tree implies both an attribute evaluation and a pattern matching over the tree. Any later change to the tree as a result of the application of a rewrite rule will force an attribute reevaluation and an update of the pattern matching information. Special algorithms are used to reduce the number of attributes which must be changed in a reevaluation.

Some properties are not catered by the mechanism as well as desired. Non-local conditions must be described semantically using several attributes and semantic functions, even when the described condition is strictly syntactic. The restrictions of a general purpose attribute evaluator may produce inefficient implementations, and the dispersion of the description through several attributes and evaluation rules will make it less understandable. Non-local actions require that the description writer update the value of the attribute which provides the connection between the local and the global rule. This updating will be dispersed in the semantic revaluation functions of several rewrite rules, which is unnatural and may also be inefficient.

Tree-to-tree Grammars

Tree-to-tree grammars [KMP84] can be seen as an extension of either attribute grammars, or, as in this subsection, term rewrite systems. They are a specialized descriptive mechanism, less expressive than attributed transformational grammars. Tree-to-tree grammars are a collection of ***extended rewrite rules***; each rule relates an ***input subgrammar*** and an ***output subgrammar***. The rule matches at subtrees that belongs to the input subgrammar and the output subgrammar describes the replacing subtree[5].

Unlike traditional term rewrite systems, there is no iteration. Instead the input tree is partitioned, and a single rewrite rule is applied to each portion to obtain the output tree. Properties of the output set can be obtained automatically from the set of rewrite rules.

A tree-to-tree grammar is ***implemented*** by translating it into an attribute grammar with new attributes which determine what input subgrammars apply to a given input tree. The semantic functions used for the new attributes implement a tree pattern matching algorithm (see everywhere in this dissertation). This new attribute grammar satisfies the constraints of attribute coupled grammars, thus proving that the ***expressive power*** of tree-to-tree grammar is not larger than attribute coupled grammars.

The main advantage of this approach is that (some) non-local actions can be described quite naturally. Probably the main disadvantage is the lack of iteration, and the somewhat imprecise state of the proposals in [KMP84]. Chapter 7 formalizes a similar proposal around the notion of a ***projection system***. The results of that chapter show how to solve, strictly in terms of rewrite systems, not only the "forward" application problem, but also its inverse function.

1.5. Thesis of this Dissertation

The intention of this dissertation is to start exploring how much it is possible to improve the "pure" rewrite system paradigm to overcome its shortcomings, and how to solve with it the tree transformation problems mentioned at the beginning of this chapter.

The biggest asset of descriptive mechanisms based on term rewrite systems is their support for serial composition and local transformations. But the basic term rewrite mechanisms support poorly both non-local transformations and parallel composition, and have trouble imposing constraints on the order of evaluation. Some descriptive mechanisms, notably attributed transformational grammars, have attempted to alleviate the problems by departing from the "pure"

[5] The proposal of [KMP84] is not very precise in some fine details.

approach and adding attribute grammars. Attribute grammars have their own problems, they still do not provide a good description of non-local conditions, and detract from the simplicity of rewrite systems. The goal of this dissertation is to extend the power of the individual rewrite rules in a way that is a "nice" extension to the "traditional" rewrite rules to deal with non-local rewrites so that, in most cases each conceptually single rewrite action can be described as a single action in the mechanism.

This is an ambitious goal. [Pel84] contained several proposals for future research. This dissertation is a first step in that direction, and restricts its attention mainly to extending the notion of pattern to allow non-locality in the form of the typed N-patterns and X-patterns of Chapter 7, and the non-linear patterns of Chapter 4.

Another goal of this dissertation is to show that even simple descriptive mechanisms based on term rewrite systems can be very useful to solve the tree transformation problems presented at the beginning of this chapter. This goal is achieved by defining and exploring the notion of bottom-up rewrite systems.

1.6. Introduction to the Remaining Chapters

The dissertation is divided into four major parts. The first, introductory part consists of this chapter and the next, which introduces basic definitions on trees, pattern matching, tree languages and tree automata, rewrite systems, and finite state tree transformations. The reader is advised at least to skim through the chapter since some of the definitions are new, particularly those on tree transformation problems and on labeled bottom-up tree automata (Definition 2.37).

The second part of the dissertation contains foundational material for the descriptive mechanism outlined in Section 1.5. Chapter 3 investigates pattern matching for the simplest type of patterns, the linear N-patterns, which are the "traditional" patterns where variables appear only once and have 0 arity. The chapter contains a consistent description of several table-driven algorithms which cover a space of table-size × matching speed. Some of the algorithms have appeared previously in the literature, but others are new and may be useful in some applications.

Chapter 4 studies non-linear N-patterns, that is, patterns as before but in which variables may appear more than once. Matching for these patterns is solved by combining matching for linear N-patterns with the testing of additional equality predicates. The chapter contains several table-driven algorithms. The size of the tables depends, in part, on how much of the results of the tests is stored: faster algorithms require larger tables.

The logical continuation of Chapters 3 and 4 is the first part of Chapter 7 where algorithms for typed X-patterns are studied. X-patterns are a generalization of traditional patterns in which variables are allowed to have non-zero arity. X-patterns are designed to describe non-local conditions. Types are constraints imposed on what a variable can match, and can appear in both N-patterns and X-patterns. Again, types allow the introduction of some non-locality into patterns. Pattern matching for typed N-patterns can be solved using techniques similar to those of Chapters 3 and 4, but pattern matching for X-patterns uses results from the third part of this dissertation.

The third part of the dissertation contains an analysis of a simple type of rewrite systems called bottom-up rewrite systems (BURS) and of its applications. Chapter 5 introduces BURS and shows how to solve a reachability problem for them. The algorithm used is based on computing a state characterizing the interaction of the rewrite system and the input tree. If the set of possible states for a given rewrite system is finite, the algorithm can be implemented very efficiently. The chapter discusses the algorithm and conditions for the finite number of states.

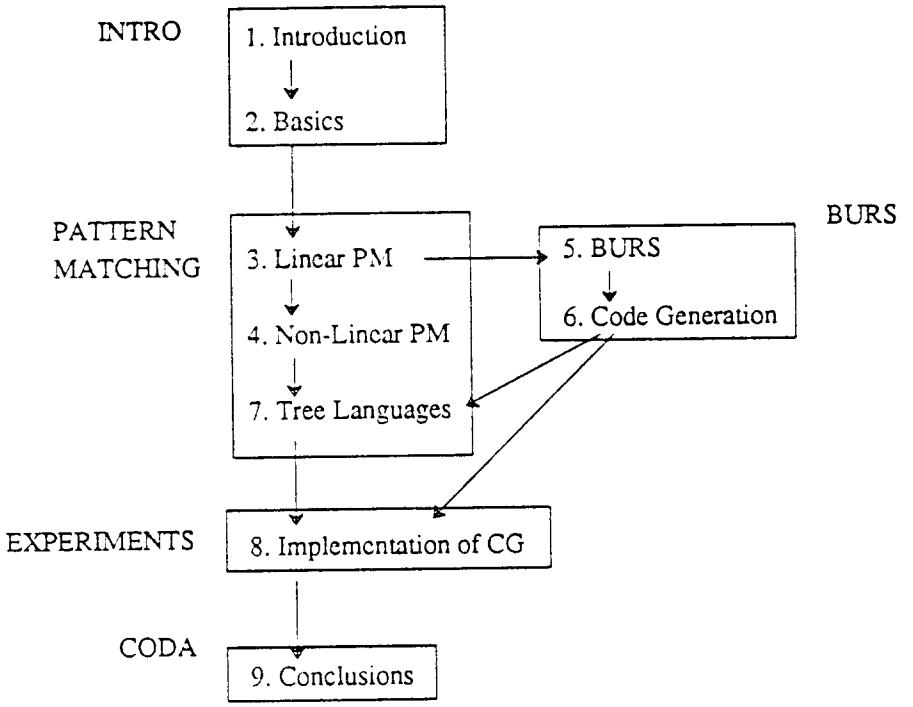
Chapter 6 then analyzes the problem of code generation and shows how to solve it using a modified reachability problem over BURS that have been extended with cost information.

Finally, Chapter 7 shows how to do typed X-pattern matching using BURS and defines a new type of descriptive mechanism, called projection systems, for which forward and inverse

projection systems can be solved by solving a reachability problem for BURS. Projection systems are expressive enough to describe the transformations between abstract and concrete syntax trees of language-based editors.

The last part of the dissertation, Chapter 8, reports on an implementation of a locally optimal code generator generator based on the results of Chapter 6. The implemented algorithm is compared with several other recent proposals and shown to be substantially faster than other optimal code generators and even faster than implementations of successful techniques that are not guaranteed to be optimal [GlG78]. Moreover, the tables obtained are of competitive size.

Chapter 9 discusses the dissertation and contains some conclusions. Figure 1.4 indicates the inter-dependencies among the chapters, which are represented by their chapter number, together with a short identifying keyword.



Dependencies among Chapters

Figure 1.4

CHAPTER 2

Basics of Tree Rewrite Systems

'When I use a word,'
Humpty Dumpty said, in rather a scornful tone,
'it means just what I choose it to mean
– neither more nor less.'

'The question is,' said Alice, 'whether you
can make words mean so many different things.'

'The question is,' said Humpty Dumpty,
'which is to be master – that's all.'

[Lewis Carroll [1832-1898]]

This chapter collects definitions, notation conventions, and results related to the notions of tree, pattern, tree language, and rewrite system. Some of the terminology is traditional, some is new, and some corresponds to traditional notions in a new perspective. It is always difficult to find a happy balance between formality and convenience. This chapter leans towards formality to compensate for the possibly different backgrounds among readers; later chapters can then be more informal and rely on the definitions presented here for fine points.

First some basic notation. A ***sequence*** $a_1 a_2 \cdots a_n$ sometimes will also be represented as $a_1 \bullet a_2 \bullet \cdots \bullet a_n$ to emphasize its components. The empty sequence is represented as "ε", and a sequence consisting of a single element a will frequently be denoted by a. Sequences with elements from some set S are also called ***words*** over the ***alphabet*** S. The concatenation operation between sequences is denoted explicitly by // or implicitly simply by the concatenation of the two operands. Other notations and definitions not defined in this dissertation are from elementary set, graph, and language theory. The introductory chapters of most textbooks on those topics (for instance [AhU73]) will contain the necessary definitions.

Definitions will frequently be presented as a collection of "defining equations" where the left hand side describes a syntactic case, the symbol $\triangleq$ indicates it is a defining equation, and the right hand side gives the definition. $\triangleq$ can be read as "is defined to be". This convention is commonly used in areas such as denotational semantics. For example, the length of a sequence can be defined as:

$$\mathit{length}(\varepsilon) \triangleq 0$$
$$\mathit{length}(x // \alpha) \triangleq \mathit{length}(\alpha)+1$$

Note that in the last defining equation there are implicit universal quantifiers; that is, it should be read as $\forall \alpha \forall x$..... Moreover, α is assumed to range over the set of sequences and x over the set of sequence elements. Both constraints are left out to reduce verbosity in the definitions. There is an implicit defining equation defining all remaining cases (if any) to have value undefined. The example above is a total function.

Section 2.1 defines trees, patterns, linear and non-linear patterns, N- and X-patterns, and pattern matching. The section also compares pattern matching to unification and to subgraph isomorphism.

Section 2.2 discusses a class of accepting mechanisms for tree languages. Tree languages are useful in several applications, including problems related to pattern matching and rewrite systems. The topic is revisited in Chapter 7.

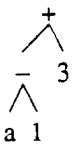
The main application of pattern matching in this dissertation is to rewrite systems. A rewrite system is a collection of rewrite rules, each one describing the operation of replacing a tree matching some pattern by another tree. Section 2.3 presents the basic definitions, including some traditional problems: confluence and termination, and some non-traditional ones: reachability and blocking.

Section 2.4 defines a class of tree transformations. It also relates tree languages defined through accepting mechanisms to those defined through generating mechanisms.

2.1. Trees and Pattern Matching

The notion of a tree can be formalized in several ways. This dissertation uses two main formalizations. The basic definition identifies a tree with a subset of "well-formed" words over an alphabet. This alphabet is composed of a set of operators and three other symbols: "(", ")", and ",". For example, $+(-(a,1),3)$ is a tree over the set of operators $\{+,-,a,1,3\}$. The subtrees of a tree are its subwords that are trees (that is, they are well formed). For example both $-(a,1)$ and 3 are subtrees of the tree above, but $-(a$, is not.

A subtree of a tree is identified through its ***position*** relative to the root of the tree. The notion of a position can be related intuitively to an alternate representation of a tree: a rooted, directed, and acyclic graph, with nodes labeled with the operators, and all nodes having a single incoming arc except for the root which has none. For example, $+(-(a,1),3)$ is represented as:



In a graph representation of a tree, the position of a subtree is a sequence of integers describing a path from the root of the tree to the root of the subtree, each integer indicating (in a left-to-right numbering of the children of a node starting from 1) the next node in the path. In the above example, the position of the subtrees $-(a,1)$ and 1 are " 1 ", the left-most subtree of the root, and " 1 • 2 ", the second subtree of the first subtree of the root, respectively.

The notion of a position can be formalized and leads to an alternative definition of a tree as a "tree shape", the set of all the positions in the tree, together with a labeling from positions in the shape into the operators. This is the approach used, for example, in [Knu73].

In this dissertation, trees are formalized either as a set of words, or as a tree shape with a labeling. Informally, trees will also be represented as graphs.

Definition 2.1 *An **operator set** Op is a set of symbols together with an **arity function**, assigning to each symbol a non-negative integer, its arity. Op will always be assumed not to have the special symbols "(", ")", and ",". Operators with arity 2, 1, and 0 are said to be **binary**, **unary**, and **nullary** operators, respectively.*

The set of trees over Op, denoted by L_{Op}, is the smallest set of non-null words with alphabet $Op \cup \{"(", ")", ","\}$ satisfying

$$\forall\, op \in Op, \text{ with arity } n,\ \forall\, t_1, \ldots, t_n \in L_{Op},\ "op \bullet (\bullet t_1 \bullet, \bullet \cdots \bullet t_n \bullet)" \in L_{Op}$$

A member of L_{Op} is said to be a **term** *over* **Op**, *or a* **tree** *over* **Op**.

If $t \in L_{Op}$, the set of all **positions in** *t, denoted by P_t, is the set of sequences of non-negative integers defined recursively as:*

$$P_{op(t_1,\ldots,t_n)} \triangleq \{\varepsilon\} \cup \{k/\!/s_k \text{ such that } 1 \le k \le n \text{ and } s_k \in P_{t_k}\}.$$

If p, q, and r are positions and $p = q/\!/r$ we say that q is an **ancestor** *of p, and that p is a* **descendant** *of q.*

If t is a tree, and $p \in P_t$ is a position in t, the **subtree of** *t at position p, $t_{@p}$, is defined recursively as:*

$$t_{@\varepsilon} \triangleq t$$

$$op(t_1,\ldots,t_n)_{@k/\!/s} \triangleq (t_k)_{@s}, \text{ if } 1 \le k \le n^{6}.$$

The **set of subtrees** *of t is the collection of trees $t_{@p}$, for $p \in P_t$. Because of the one-to-one correspondence, positions and subtrees can be used interchangeably. A subtree is also called a* **subterm**.

The **frontier** *of t, $fr(t)$ is the word over* **Op** *defined by:*

$$fr(op) \triangleq op, \text{ if } arity(op)=0;$$

$$fr(op(t_1,\ldots,t_n)) \triangleq fr(t_1)\cdots fr(t_n), \text{ if } arity(op)=n \ge 0.$$

The **children** *of a tree t are the trees $t_{@k}$ for $1 \le k \le n$, n being the arity of the operator at the root of t.*

The **leaves** *of a tree t, $leaves(t)$ is the subset of* **Op** *defined by:*

$$leaves(op) \triangleq op, \text{ if } arity(op)=0;$$

$$leaves(op(t_1,\ldots,t_n)) \triangleq leaves(t_1) \cup \cdots \cup leaves(t_n), \text{ if } arity(op)=n \ge 0.$$

The **height** *of t, $height(t)$ is the natural number defined by:*

$$height(op) \triangleq 1, \text{ if } arity(op)=0;$$

$$height(op(t_1,\ldots,t_n)) \triangleq 1+\max\{height(t_1),\ldots,height(t_n)\}, \text{ if } arity(op)=n \ge 0.$$

A **forest** *is a collection of trees.*

Thus, $t \equiv +(-(a,1),3)$ is a tree (a term) over $\{+,-,a,1,3\}$, where the arities of + and − are 2, and those of a, 1, and 3 are 0. P_t is $\{1 \cdot 1, 1 \cdot 2, 2, \varepsilon\}$, with associated subtrees: $t_{1 \cdot 1} \equiv a$, $t_{1 \cdot 2} \equiv 1$, $t_1 \equiv -(a,1)$, $t_2 \equiv 3$, and $t_\varepsilon \equiv t$. The frontier is $fr(t) = a\ 1\ 3$. The set of leaves is $\{a,1,3\}$. The sequence +(3) is not a tree.

An alternate way to characterize a tree is as a tree shape plus a labeling.

Definition 2.2 *A* **tree shape** *is a collection of position sequences S over the non-negative integers satisfying:*

if $(s_1/\!/s_2) \in S$, then $s_1 \in S$; and,
if $(s_1/\!/k \in S)$, $k>0$, and $1 \le k\,1 < k$, then $s_1/\!/k\,1 \in S$.

---

[6] The parentheses in $(t_k)_{@s}$ are added for readability to delimit the subscript k and have no other significance.

*The **arity** of a position p in a shape S is the largest integer n such that $p \cdot n \in S$. A **labeling** of a shape S is a mapping, f, from S into **Op**, such that: if $f(p) \equiv op$, for $p \in S$, $op \in Op$, the arity of op in **Op** is the same as the arity of p in S,*

The reader can easily prove:

Proposition 2.1 *The following two statements are true:*

For every tree T, the set P_T is a tree shape.

Let S be a tree shape and f a labeling of S. Then, there is a unique tree T with set of positions S and such that the root of $T_{@p}$ equals $f(p)$

The next chapters will use either of the two characterizations of trees. In this dissertation, the arity of an operator is assumed to be fixed. This models closely the semantics of the operators in the applications to programming systems, and although no basic result uses this assumption, some implementations (like the one of Chapter 8) depend on it for their efficiency. The European school of algebraic trees is similar to the definition of a tree as a labeled shape except for having operators with variable arity.

Tree Patterns

Patterns are intended to be used to describe "shapes" of trees. Intuitively, they can be thought of as trees where some positions are labeled not with an operator but with a "variable". The variables can then be replaced by trees to obtain all the trees that have the "shape" described by the pattern. In general, there may be more than one position in the tree that is labeled with a variable. When there are no constraints between the values used to replace any two variables, the pattern will be called **linear** (Def. 2.3). If the values used to replace two or more variables are constrained to be identical, the pattern will be called **non-linear**. Finding a convenient formal definition for linear patterns is quite simple: we can use a reserved operator symbol, for instance V, and define patterns as trees over a set of operators extended with V. Thus, if a is a nullary operator and + is a binary operator, we could describe the shape of the trees rooted by a + and with a as right child, by the tree $+(V,a)$. This definition is unique (that is, each "different" shape of trees corresponds to a different pattern), and, since a pattern is formally a tree, we can use tree operations in patterns (for instance, the left child of the pattern $+(V,a)$ is the pattern V).

The situation is substantially more complex if we want to describe non-linear patterns. For example, consider two patterns, ρ_1 describing the shape of trees rooted by a binary operator +, and ρ_2 describing the shape of trees rooted by the binary operator + and having a left child identical to a right child. Following the approach outlined in the previous paragraph, we could define ρ_1 and ρ_2 as the two trees $+(V_1,V_2)$, and $+(V_1,V_1)$, where V_1 and V_2 are two symbols that represent variables. The problem with this approach is that now patterns do not have a unique representation. Thus, the pattern represented by $+(V_1,V_2)$ could also be represented by $+(V_2,V_1)$. There are several solutions for this problem.

A first possibility is just to let the two representations exist as patterns, and to define an equivalence relation $\equiv$ between patterns so that two patterns ρ_1 and ρ_2 are equivalent if there is a one-to-one renaming of the variables in one into the variables of the other. Thus, $+(V_1,V_2)$ would be equivalent to $+(V_2,V_1)$. This approach has the disadvantage that the writer must be careful to use $\equiv$ instead of = whenever appropriate, and that some operations, such as constructing a set of patterns, must be done very carefully to avoid having equivalent patterns in the set.

Another possibility is to define a pattern as an equivalence class under $\equiv$ of the trees defined above. Such a definition simplifies many operations but makes others more complex. In particular, since patterns are no longer trees, tree operations can not be applied to patterns. This problem could be solved by defining "coercion" operations between patterns and trees, taking a tree into its equivalence class, and an equivalence class into any of its members. Then, an expression like

$+(X,a)$ used in some context would mean either a tree or a pattern depending on the context. This approach to the formal definition of a pattern has the disadvantage that some contexts do not uniquely determine whether the desired value is a tree or an equivalence class. The approach also has some other disadvantages. For instance, the notion of an assignment to the variables in a pattern would have to be independent of the names of the variables, probably by using positions. Tree positions are less convenient that variable names because positions are relative while names are absolute.

A final possibility is to define a pattern as some canonical tree. For instance, we could require all the variables to be taken from a fixed set $\{V_1,V_2,\ldots,\}$ and to be used in a predefined order, so that $+(V_1,V_2)$ would be a pattern, but $+(X,Y)$ and $+(V_2,V_1)$ would not. One problem of this approach is that tree operations applied to patterns may produce non-patterns: for instance, the right child of $+(V_1,-(V_2,V_3))$ is not a pattern. This renaming also reduces the readability of some operations: for instance, it makes it less visible that $-(V_1,V_2)$ is the right child of $+(V_1,-(V_2,V_3))$.

The approach that we use in this dissertation is to define a pattern as a tree over an operator set which is extended with a disjoint set of variable symbols (Definition 2.3) and to provide an equivalence relation on patterns (Definition 2.7). In addition we use flexible canonical forms (the set of subpatterns Π_F of Definition 3.2, and the **extended pattern set** of Definition 5.7) to reduce the number of cases in which we have to make a distinction between equality and equivalence and yet retain some readability in our examples.

Although the notation used in this dissertation is correct, the author is not completely satisfied with it. Most of the complexity arises from the presence of non-linear patterns. It should be possible to obtain an overall improvement in clarity of presentation by assuming patterns to be linear in all chapters except in Chapter 4, where they could be defined as extensions to linear patterns. Time constraints have prevented us from following this approach in this dissertation.

This dissertation deals with two types of patterns, depending on the arity of the variables. If all variables have arity 0 we obtain the traditional notion, called here an "N-pattern"; if some variable has non-zero arity, the pattern is called an "X-pattern". X-pattern is a new notion used to describe non-local situations.

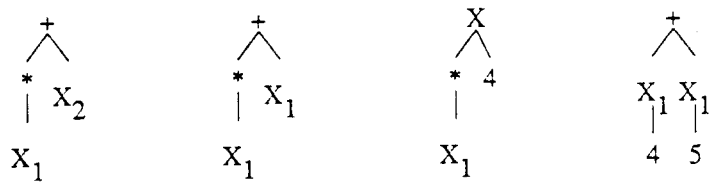
Definition 2.3 *A **pattern** is a member of $L_{Op\cup V}$, where Op and V are disjoint sets called the **operators** and the **variables** respectively. The pattern is called an **N-pattern** if all the variables have arity 0, and an **X-pattern** otherwise. A pattern is called **linear** if it contains at most one occurrence of any particular variable; otherwise it is called **non-linear**.*

*Associated with each variable X appearing in a pattern there is a set of trees called its **type**, denoted by type(X). If the arity of a variable is n, it has a default type which is the set of all linear patterns over **Op** with exactly n variables. Alternatively, the type of a variable can be any subset of the default type.*

*If ρ is a pattern over **Op** and **V**, **Vars**(ρ) denotes the set of variables in ρ. If ρ_1 and ρ_2 are patterns over **Op** and **V**, we say that ρ_1 and ρ_2 are **variable-disjoint** if **Vars**(ρ_1)$\cap$**Vars**(ρ_2)$=\varnothing$. By extension, given a set of patterns $F=\{\rho_1,\rho_2,\ldots,\rho_n\}$, F is **variable-disjoint** if for i,j, $1\le i,j\le n$, **Vars**(ρ_i)$\cap$**Vars**(ρ_j)$=\varnothing$.*

Typed variables are useful to constrain the valid replacements for the variable (see Def. 2.6). Note that a variable with arity 0 has as default type the set of all trees over the operator set. Figure 2.1 shows some examples of patterns. In a convention followed throughout the dissertation, the names of variables start with an upper case letter. From left to right the examples are linear and non-linear N-patterns, and linear and non-linear X-patterns. In the rightmost pattern, the default type of X_1 is the set of all linear patterns with exactly one variable. Traditional tree

patterns [HoO82] are linear N-patterns.



Examples of Patterns

Figure 2.1

Patterns are used for matchings. Intuitively, a pattern "matches" against a tree, called the "subject", if there is a way to associate a value with each variable in the pattern, so that when each instance of the variables is replaced by its associated value, the resulting tree is identical to the subject tree.

The matching of N-patterns and X-patterns differs only in what are the values that can be associated with a variable: in the first case a tree without variables, in the second an N-pattern with as many variables as the arity of the X-variable. With some care, the formal definition can be made identical in both cases. First some auxiliary definitions and new notation.

Definition 2.4 *Let T be a tree with labeling f_T, $p \in P_T$ a position in T, and t a tree with labeling f_t. The* **replacement** *of the subtree $T_{@p}$ by t is denoted as $T_{@p \leftarrow t}$, and is defined as the new tree with shape S, and labeling f' such that:*

$$\text{If } r \in P_T \text{ is not of the form } p//s \text{, then } r \in S \text{, and } f'(r)=f_T(r);$$
$$\text{If } r \in P_t \text{, then } p//r \in S \text{, and } f'(p//r)=f_t(r).$$

Let T be a pattern over **Op**, *X a variable with arity $n \geq 0$ appearing in T at position p, and t a pattern over* **Op** *with n distinct variables with arity 0 at positions $q_1,\ldots, q_n$ (and maybe others with non-zero arity). The* **replacement** *of X in T at p by t, denoted by $T_{X@p \leftarrow t}$, is defined to be equal to $T_{@p \leftarrow B^n}$, where $B^0 \triangleq t$, and, for $1 \leq i \leq n$, $B^i \triangleq B^{i-1}{}_{@q_i \leftarrow T_{@p_i}}$.*

If T is a pattern over **Op**, *X is a variable in T appearing at positions $p_1, \ldots, p_m$, and t is a pattern over* **Op**, *the* **replacement** *of X by t in T, is denoted by $T_{X \leftarrow t}$, and defined to be equal to A^m, where $A^0 \triangleq T$, and $A^i \triangleq A^{i-1}{}_{X@p_i \leftarrow t}$.*

For example, if $T=+(3,-(9,4))$, and $t=5$, then $T_{@1 \leftarrow t}$ is the tree $+(3,5)$. If $T=+(X,-(4,X))$, then $T_{X \leftarrow 3}$ is the tree $+(3,-(4,3))$. And, if $T=+(X(a,b),3)$, and $t=*(X_1,9,X_2)$, then $T_{X \leftarrow t}$ is the tree $+(*(a,9,b),3)$.

Formally, the correctness of the above definition of the replacement of a variable requires a proof that the order in which the variables are listed does not change the result. The proof is immediate and is left to the reader.

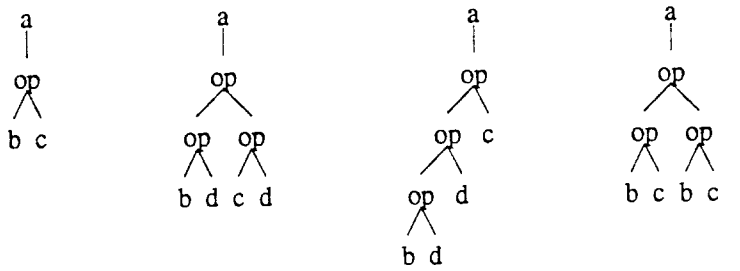
Definition 2.5 *Let Op and V be disjoint sets of operators and variables with arity. An assignment over Op for V is a function assigning to each variable $A \in V$ of arity n_A, a linear N-pattern over Op with exactly n_A variables.*

Let σ be an assignment over Op for V, and ρ a pattern over Op and V, with n variables V_1 ,..., V_n. The application of σ to ρ, denoted $\sigma(\rho)$, is defined to be A^n, where $A^0 \triangleq \rho$, and, for $1 \le i \le n$, $A^i \triangleq A^{i-1}{}_{V_i \leftarrow \sigma(V_i)}$.

Definition 2.6 *Let ρ be a pattern over V and Op, and T be a tree over Op. An assignment σ over Op and V is a match of ρ at T, if $\sigma(\rho)=T$, and, for every variable X in ρ, $\sigma(X) \in type(X)$. ρ is said to match at T, if there exists such a match. Let ρ_0 be a subtree of ρ; the tree associated with ρ_0 by σ is defined to be $\sigma(\rho_0)$. If ρ matches at T, then T is a subject at which ρ matches.*

For example, the pattern $+(X,Y)$ matches twice in the tree $+(-(2,a),+(3,4))$; in the first case the associated assignment is $\{X \leftarrow ``-(2,a)'', Y \leftarrow ``+(3,4)''\}$; in the second case, $\{X \leftarrow ``3'', Y \leftarrow ``4''\}$. If the pattern were modified so that $type(X)$ included only those trees whose leafs are digits, then only the second assignment would be legal.

Since, given an N-pattern and a tree, there is at most one match of the pattern at the tree, the assignment can always be computed from the pattern, although in some applications it might be convenient to keep the representation explicit. X-patterns are different. X-patterns may have more than one match at a given node. As an example, consider the X-pattern $\rho = a(X(b,c))$ and the four trees:



Each of the first 3 trees has one match for ρ, namely: $\{X \leftarrow op(X_1,X_2)\}$, $\{X \leftarrow op(op(X_1,d),op(X_2,d))\}$, and $\{X \leftarrow op(op(op(X_1,d),d),X_2)\}$, but it has three matches at the last tree: $\{X \leftarrow op(op(X_1,X_2),op(b,c))\}$, $\{X \leftarrow op(op(b,c),op(X_1,X_2))\}$, and $\{X \leftarrow op(op(X_1,c),op(b,X_2))\}$.

One way to reduce the number of matches of an X-pattern, and thus make the X-pattern more precise, is by giving non-default types to its variables. This can also be done for N-patterns.

In the introduction to this subsection we mentioned that the names of the variables in a pattern are non-significant. Formally:

Definition 2.7 *Let Op be an operator set, let V be a set of variables, and let ρ_1 and ρ_2 be two patterns over Op and V. ρ_1 and ρ_2 are equivalent ($\rho_1 \equiv \rho_2$) if for any tree T, ρ_1 matches at T if and only if ρ_2 matches at T.*

It is straightforward that

Proposition 2.2 *The $\equiv$ relation between patterns is an equivalence relation.*

It is also straightforward that

Proposition 2.3 *Let ρ_1 and ρ_2 be two patterns over Op and V. If there is a one-to-one mapping between Vars(ρ_1) and Vars(ρ_2) that preserves the types of the variables, then $\rho_1 \equiv \rho_2$.*

Section 3.1 defines more relations between patterns.

This dissertation studies some new and old "problems". The format used to present these problems formally follows that of [GaJ80], where first the parameters of the problem are given and then the problem is described There are two main classes of problems: ***decision*** problems where the answer is either "yes" or "no", and ***construction*** problems where the answer is some object that has to be obtained.

Definition 2.8 *We say that a decision problem is **decidable** if there exists a procedure that always terminates and that will determine for every possible input whether the answer to the problem is "yes" or "no". If no such procedure exists the problem is said to be **undecidable**. If there is a procedure that provides the correct answer when it terminates but is guaranteed to terminate only when the answer to the problem is "yes", the problem is said to be **semi-decidable**.*

*We say that a construction problem is **solvable** if there exists a procedure that always terminates and that will determine for every possible input the correct answer. If no such procedure exists the problem is said to be **unsolvable**.*

An example of an unsolvable problem is the REACHABILITY problem (Definition 2.28) for general rewrite systems. The main problem of pattern matching is a construction problem:

Definition 2.9 *Given: a set of trees, $T \subset L_{Op}$, and a set of patterns $F \subset T_{Op \cup V}$; the **tree pattern matching problem**, over T and F, abbreviated as PATTERN MATCHING, consists of, given a **subject tree** $T \in T$, finding for every subtree of T, all the patterns in F that match at it, and matching assignments for those patterns.*

Unification

Unification is a problem that seems very similar to matching for N-patterns, yet it is significantly different. Formally, unification can be defined as follows:

Definition 2.10 *Let V be a set of 0-ary variables. Two terms t and t' in $T_{Op \cup V}$ are **unifiable** if there is an assignment σ over $Op \cup V$, such that $\sigma(t) = \sigma(t')$. σ is called a **unifier** for t and t'.*

An important property of unification is the following:

Proposition 2.4 *[Rob65] For any two terms ρ_1 and ρ_2 there is a unique (up to a renaming of variables) **most general unifier**, σ_{mgu}, such that for any other unifier σ for ρ_1 and ρ_2, there is a replacement τ such that for all variables X, $\sigma(X) = \tau(\sigma_{mgu}(X))$.*

Definition 2.11 *The **unification problem** (UNIFICATION) consists of, given two terms, finding their most general unifier.*

Unification can be solved in linear time on the size of the two patterns [PaW78]. In one sense, unification is more general than pattern matching. Either term in an unification problem may contain variables, and the same variable may be present in both terms, but only the pattern in a matching problem can contain variables. Thus, any algorithm used for unification can be used for pattern matching.

Yet, in another sense unification is simpler than pattern matching. In unification only the two complete terms are considered; in pattern matching, any subterm of the subject tree is considered. Also, unification only considers a single pair of terms, while the applications of pattern matching studied here consider a set of patterns to be matched against a single subject. The algorithms for pattern matching presented later in this dissertation are significantly more efficient than

the straightforward application of unification algorithms to the same problems.

In some cases it is convenient to represent a tree in such a way as to underscore the common parts of it.

Finding Common Subexpressions in the Subject

Since a tree may have more than one identical subtree, in general, determining if the subtrees at two positions are equal requires a recursive comparison. An alternative is to encode the tree as a directed acyclic graph (dag) where multiple occurrences of subtrees are replaced by a single subdag. This dag is called a ***computation dag***.

A computation dag can be computed from a tree using a bottom-up traversal that works in time linear on the size of the subject tree. The main operation computes, given the dags corresponding to the children of a node, the dag correspond to the node, and in the process also finds if an identical dag had been computed previously (for some other subtree of this tree). This operation can be computed efficiently with a hash function. The resulting algorithm was named the ***value number method*** and presented by Cocke and Schwartz in [CoS70]; it also appears in [AhU77].

Computation dags allow the replacement of subtree comparisons by pointer comparisons. Chapter 4 shows that this type of comparison is a basic operation in non-linear pattern matching. Also, the algorithms for linear N-pattern matching [7], which are based on a bottom-up traversal of the subject tree, can use either a tree representation or a computation dag representation. Using a computation dag has the advantage of requiring traversal of fewer nodes.

Representing the subject tree as a computation dag suggests considering another problem: "subgraph isomorphism".

Subgraph Isomorphism

The technical definition of graph and subgraph isomorphism are the following:

Definition 2.12 *Given two graphs $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$, G_1 is **isomorphic** to G_2 if there is a one-to-one function $f: V_1 V_2$ such that $\langle u,v\rangle \in E_1$ if and only if $\langle f(u), f(v)\rangle \in E_2$.*

Definition 2.13 *The **graph isomorphism problem** consists of determining, given two graphs G_1 and G_2, if G_1 is isomorphic to G_2. The **subgraph isomorphism problem** is determining if there is a subgraph of G_1 that is isomorphic to G_2.*

The complexity of graph isomorphism between general graphs is a famous open problem. The problem can be solved in polynomial deterministic time[8] for several special cases, including planar graphs and, hence, trees. Graph isomorphism differs from pattern matching in its concern with only two graphs.

Subgraph isomorphism is known to be NP-complete for the general case and even for some simple special cases. One such case is when G_1 is a directed forest[9] and G_2 is a directed tree ([GaJ80]). Given the comments of the previous subsection, a careless reading of this result might suggest that pattern matching between a subject tree and an N-pattern would be NP-complete if both were represented as computation dags. This is not true; The two problems are different. A

---

[7] Unfortunately, there are some problems with X-pattern matching: the technique employed requires three passes over the subject, and using a dag conflicts with the top-down pass. More on this in a future publication.

[8] See [GaJ80] for a definition of this notion and that of NP.

[9] A directed forest is a collection of directed trees.

main difference is that matching has less freedom: if two nodes match, they must have exactly the same number of children; this is not true for subgraph isomorphism.

2.2. Tree Languages and Automata

Any tree transformation involves at least two different sets of trees: the domain of the transformation, the ***input*** set, and the image of the transformation, the ***output*** set. In most tree transformation problems it is important to be able to describe these sets so that some properties can be obtained automatically. This section presents some background material on sets of trees, also called tree languages. Most of the material in this section is from [Tha75], and [Eng75]. Recall our convention that, unless otherwise indicated, the type associated with a variable in a pattern is its default type.

A tree automaton is a tree language recognizer, that is, it defines a mapping from L_{Op} into $\{true, false\}$. The definition of a tree automaton is a generalization of that of a (word) one-way finite state automaton (FSA). Normally, the behavior of a FSA is defined through a sequence of pairs, containing a position in the input tape and a state. Without loss of generality, this can be understood as an assignment of a state to each position in the input tape. A tree automaton is like a FSA except that the state is associated with the nodes of a tree. There are two main approaches to defining a tree automata depending on the "direction" in which the "state tree" is traversed: top-down or bottom-up.

Definition 2.14 *A deterministic* ***top-down automaton*** *(DT-fsa) consists of a finite set of states S; an initial state s_0; a transition function assigning to each nullary operator op a set of final states $F_{op} \subset S$; and, for each arity n, a function $Trans_n : Op \times S \rightarrow S^n$. The automaton associates with each subtree in an input tree a single state from S. The state associated with the input tree is s_0. If $T = op(T_1, \ldots, T_n)$ is an n-ary node, then its i-th child has as its state the i-th component of $Trans_n(op, st)$, where st is the state associated with T. An input tree is accepted if every leaf of the tree with label op is associated with a state in F_{op}.*

A ***non-deterministic top-down automaton*** *fsa (T-fsa) is like a DT-fsa except that the function $Trans_n$ above has functionality $Op \times S \rightarrow (2^S)^n$, and s_0 is not a state but a set of states. The automaton associates with each subtree in the tree a set of states from S. The set of states associated with the input tree is s_0. If $T = op(T_1, \ldots, T_n)$ is an n-ary node, then the i-th child has as its set of states the union of all the i-th components of $Trans_n(op, st)$, where st is a member of the set of states associated with T. An input tree is accepted if every leaf subtree of the tree is associated with a set containing at least one final state.*

The class of languages that can be recognized with a T-fsa will be called T-RECOG. The class of languages that can be recognized with a DT-fsa will be called DT-RECOG.

T-fsa are also called ***root-to-frontier*** automata (RFA) in the literature. DT-fsa are not very powerful and therefore not very interesting. For example, the (finite) set of trees $\{op(a,b), op(b,a)\}$ cannot be recognized by a DT-fsa. Intuitively the reason is that a DT-fsa can only recognize that all the "paths" from the root to the leaves of the tree are of some form. Thus, in the example above, the automaton accepts as valid, for the tree $op(a,b)$, the paths $op \cdot a$ for $1 \cdot 1$, and $op \cdot b$ for $1 \cdot 2$; and, for $op(b,a)$, the paths $op \cdot b$ for $1 \cdot 1$, and $op \cdot a$ for $1 \cdot 2$. Hence, it must also accept the trees $op(a,a)$, and $op(b,b)$, and $\{op(a,b), op(b,a)\}$ cannot be a language accepted by a T-fsa. But, note that this set can be recognized by a (non-deterministic) T-fsa: the automaton just "guesses" the right tree to check.

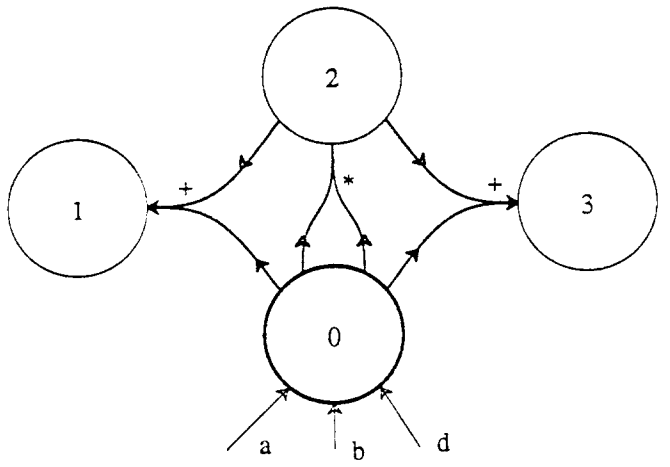
An alternate approach to defining a tree automaton is based on a bottom-up traversal of the tree.

Definition 2.15 *A **deterministic bottom-up automaton** (DB-fsa) consists of a set of states S; a final set $F \subset S$; and, for each arity n, a transition function $Trans_n : Op \times S^n \rightarrow S$. The automaton associates with each subtree in the tree a single state from S. If $T = op(T_1, \ldots, T_n)$ is an n-ary node, then the state associated with T is $Trans_n(op, st_1, \ldots, st_n)$, where st_i is the state associated with T_i. An input tree is accepted if the state associated with it is a final state, that is, a member of F.*

*A **non-deterministic bottom-up automaton** (B-fsa) is like DB-fsa except that the function $Trans_n$ has functionality $Op \times S^n \rightarrow 2^S$. The automaton associates with each subtree in the tree a set of states from S. If $T = op(T_1, \ldots, T_n)$ is an n-ary node, then the state associated with T is the union of all $Trans_n(op, st_1, \ldots, st_n)$, where st_i is the state member of the set associated with T_i. An input tree is accepted if the the automaton associates with it a set including a final state.*

The class of languages that can be recognized with a B-fsa is denoted B-RECOG. The class of languages that can be recognized with a deterministic B-fsa is denoted DB-RECOG.

The initial state in a B-fsa at a leaf is obtained from the application of $Trans_0$ to the operator at the leaf. B-fsa are also called ***frontier-to-root*** automata (FRA) in the literature. In this dissertation, B-fsa are frequently represented graphically in a way resembling a graph: states are drawn as nodes, while transitions are drawn as some type of higher-order directed edges. In addition, most of the B-fsa are complete, that is, there is a new state at any node for any combination of states of its children, and one node will be distinguished as the "default" transition, to which any non-specified transition will go. Figure 2.2 shows a B-fsa with 4 states numbered 0 to 3 (See Figure 3.2 for another interpretation of the same example). This example is a non-deterministic B-fsa. The defaul: state is **shown at the bottom**; the default transitions **are not shown** to avoid cluttering the figure.



Example of B-fsa

Figure 2.2

Non-deterministic B-fsa have the same expressive power as deterministic B-fsa. There is an algorithm, similar to the one used for string fsa, that obtains a deterministic B-fsa from a non-

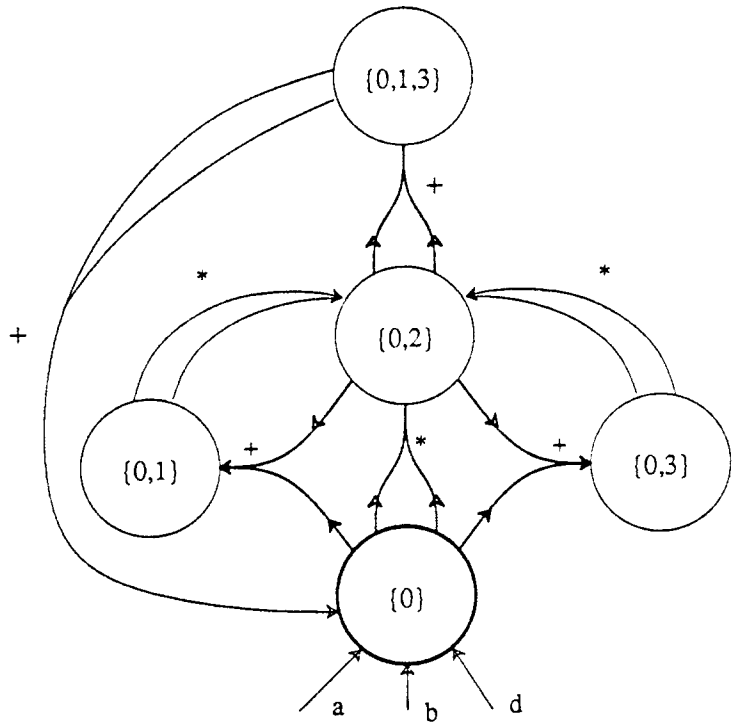
deterministic one; the states of the DB-fsa model the power sets of the states of the original B-fsa. Figure 2.3 shows this algorithm, which is used in later chapters.

```
procedure ND-to-D-Bfsa(ND)
    let ND be a non-deterministic B-fsa,
     and Trans^ND_n its transfer function.
    let D be the new (deterministic) B-fsa;
     initially it has no states;
     let Trans^D_n be the new -to be determined- transfer function.
    while D has changed
        for each possible n-ary operator Op
            for each n-tuple pair (N_1, . . . , N_n) of nodes in D do
                for each n-tuple pair (st_1,....,st_n), with st_i ∈ N_i do
                    let S be the set of all states st in ND such that
                     st=Trans^ND_n(st_1, . . . , st_n)
                    if no node in ND has label S
                    then
                        create a new node;
                        add it to ND;
                        make Trans^D_n(N_1, . . . , N_n) equal to the new node;
                    if there is a node but no transition
                    then
                        make Trans^D_n(N_1, . . . , N_n) equal to the pre-existing node;
```

Non-Deterministic to Deterministic B-fsa

Figure 2.3

Figure 2.4 shows part of the deterministic equivalent of the automaton of Figure 2.2 (and, again, see Figure 3.7 for a related figure). In the figure, all the states of the new automaton are shown, but not all the edges; the reader may want to complete the figure. As before, the "default" state is shown at the bottom, and all the "edges" entering a given state are labeled with the same operator. The "pseudo-graph" representation of the B-fsa is adequate for non-deterministic automata with a small number of non-default transfers, but it becomes quite unmanageable for larger examples. Nevertheless, the graph representation is quite suggestive, and not much worse than other alternatives.



Example of a Deterministic B-fsa

Figure 2.4

Another algorithm that will be used later is the one that, given a DB-fsa, will obtain the DB-fsa with the smallest number states that recognizes the same set. The algorithm, shown in Figure 2.5, is a simple modification of the one used in string fsa. The algorithm keeps and updates equivalence classes between states. The first two classes are characterized by whether a state is in the final set or not. Then two states are kept in the same class if, for every operator, they transfer to equivalent states. The algorithm terminates when no more changes occur.

```
procedure minimize()
    let new_class_of and class_of be arrays over the set of states.
    for each state st
        if st is a final state then
            new_class_of [st]=0;
        else
            new_class_of [st]=1;
    changes =true;
    class_cnt =2;
    while (changes) do
        changes =false;
        for each class C do
            for each state st^a ∈ C do
                new_class_of [st^a]=class_of [st^b];
                for each state st^b ∈ C do
                    if transfer_equivalent(st^a,st^b) then
                        new_class_of [st^b]=class_of [st^b];
                    else
                        new_class_of [st^b]=class_cnt;
                        changes =true;
                class_cnt +=1;
            exchange new_class_of and class_of;

function transfer_equivalent (st^a,st^b)
    for each operator Op with arity n
        for each n-1 states st^2,...st^n
            for each permutation π of 1..n
                let π^a be π_o(st^a,st^2, . . . ,st^n);
                let π^b be π_o(st^b,st^2, . . . ,st^n);
                let c^a be class_of [Trans_n(Op,st_{π_1^a}, . . . ,st_{π_n^a})];
                let c^b be class_of [Trans_n(Op,st_{π_1^b}, . . . ,st_{π_n^b})];
                if c^a ≠c^b
                    return (false);
    return (true);
```

Minimizing a DB-fsa

Figure 2.5

The reader can furnish the proofs for the other propositions related to tree automata: they are all quite similar to those in the case of string languages. Propositions 2.5 to 2.7 can be proved in a similar way to the corresponding properties for sequential automata; [Tha75] and [Tha67] contain some more details.

Proposition 2.5 *The following equations are true:*

$B\text{-}RECOG \equiv T\text{-}RECOG$.

$DB\text{-}RECOG \equiv B\text{-}RECOG$.

$DT\text{-}RECOG \subseteq T\text{-}RECOG$.

Because of the above equivalences, B-RECOG is normally called RECOG.

The members of RECOG are the ***recognizable sets***. They are a nice and stable class:

Proposition 2.6 *RECOG is a boolean algebra.*

There are tree recognizers more powerful than FRA. An example is the "push down tree automaton" of Guessarian [Gue81]. This is a top-down tree automaton with unlimited memory. The memory is a push-down store and it is, in the most general form, a tree structure from which the components can be accessed. When the tree structure is a chain, the memory corresponds to the traditional notion of a stack.

An example of a tree language that is recognizable by a push-down tree automaton but is not in RECOG is $\{Plus(Minus^i(1),Minus^i(2)) \mid i \geq 0\}$, where $Minus^i$ means a chain of i *Minus* (non-membership in RECOG can be shown using Proposition 2.7 below). Although push-down tree automata are more powerful than B-fsa, it is more difficult to prove properties of them, and, since B-fsa seem adequate to the applications investigated in this dissertation, they are the only ones used.

In regular word languages, there exist effective procedures to determine whether a language is empty and whether it is finite. Both results follow from the existence of a pumping lemma. A similar result is true for recognizable sets.

Proposition 2.7 *Let A be a B-fsa over Op with r states, and let T be a tree accepted by A with height larger than r. Then there are linear N-patterns* ρ *and* ϕ *over* $Op \cup \{X_1\}$ *and a tree t over Op such that:*

- *The height of* ϕ *is larger than 1*
- $T = \rho_{X_1 \leftarrow T'}$, *where* $T' = \phi_{X_1 \leftarrow t}$.
- *A accepts* $\rho_{X_1 \leftarrow T^n}$, *where* $T_0 = t$, *and* $T^i = \phi_{X_1 \leftarrow T^{i-1}}$.

The pumping lemma can be proved in a similar way as in the sequential case. The lemma leads to:

Proposition 2.8 *Finiteness and emptiness are decidable for recognizable sets.*

RECOG can also be characterized through a class of "regular" expressions (see [Tha67]), but this dissertation does not use that characterization.

Context-Free Grammars

Recognizable sets are strongly related to context-free grammars. This section assumes the standard notions of ***context-free grammar*** and ***derivation tree***, see [Har78] or [AhU73] for the definitions. Unless otherwise specified, context-free grammars are assumed to be ε-free.

Definition 2.16 *A set of trees U is called* ***local*** *if there is a context-free grammar* $G = (N, \Sigma, R, S)$, *and an* $M \subseteq N$, *such that the set of derivation trees of G rooted with variables in M is U.*

Given G and M, their associated local set can be recognized by a B-fsa with $|N|+2$ states representing the nonterminals in N plus the starting state S_0, and an error state S_{error}. The automaton just assigns to a tree T (1) the state S_0, if T is a leaf, (2) S_{error} if T is not a derivation tree

for G, and (3) S_B if T is a derivation tree in G rooted by $B \in N$. The final set of states of the automaton is the set S_B for $B \in M$. Thus,

Proposition 2.9 *Local sets are in RECOG.*

The converse of the above proposition is not true, but can be patched up:

Definition 2.17 *A **tree relabeling** is a mapping between two sets of operators Op and Op' such that their arity is preserved. A relabeling induces a function assigning to a tree T a new tree f (T) where each operator has been replaced by its image under the relabeling.*

Relabelings satisfy:

Proposition 2.10 *Every recognizable set is the image of a local set under a relabeling transformation.*

Proposition 2.10 can be proved by constructing a context-free grammar that will simulate the behavior of the B-fsa. There each non-terminal will characterize the state of the tree, together with the original operator. The relabeling discards the state and retrieves the operator.

Another possible generalization of the notion of a relabeling is that of a homomorphism:

Definition 2.18 *Let Op and Op' be two sets of operators with arity and let $X = \{X_1, \ldots, X_i, \cdots\}$, be a set of variables of arity 0 disjoint from both Op and Op'. A **homomorphism** between Op and Op' is a mapping $h : Op \rightarrow T_{Op' \cup X}$, such that, if op has arity n, h (op) has (at most) only variables $X_1, \ldots, X_n$.*

*A homomorphism is **linear** if, for all operators op, h (op) is a linear pattern.*

An homomorphism h induces a transformation h' between tree languages. For $t \in L_{Op}$, h' (t) is defined recursively by:

$$h'(op(t_1, \ldots, t_n)) \triangleq h(op)_{X_1 \leftarrow h'(t_1) \cdots X_n \leftarrow h'(t_n)},$$
$$h'(op) \triangleq op$$

[Eng75] contains full proofs of the next two propositions.

Proposition 2.11 *RECOG is not closed under general homomorphisms. But, RECOG is closed under linear homomorphisms (including relabelings).*

Proposition 2.12 *RECOG is closed under (general) inverse homomorphisms, that is, if $A \in RECOG$, and h is a homomorphism, then $h^{-1}(A) \in RECOG$.*

Section 7.2 defines a generalization of homomorphism called a "projection system". Corollaries 7.3 and 7.2 show that RECOG is closed under both projections and inverse projections.

2.3. Rewrite Systems

The basic definition of rewrite system is:

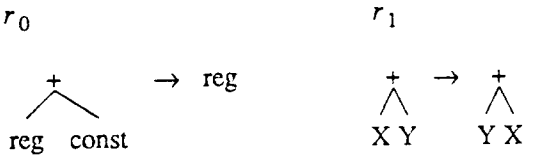
Definition 2.19 *A **rewrite rule** over $Op \cup V$ is of the form $\alpha \rightarrow \beta$, where α and β are patterns in $Op \cup V$, and $Vars(\beta) \subset Vars(\alpha)$. α is called the **input pattern** of the rewrite rule, and β is the **output pattern**. A **rewrite system** over $Op \cup V$ is a collection of rewrite rules over $Op \cup V$.*

*A rewrite system is **linear** if all the patterns of all the rewrite rules are linear. Otherwise it is **non-linear**.*

A rewrite rule $\alpha \rightarrow \beta$ *is* ***erasing in*** X *if* $X \in Vars(\alpha)$ *and* $X \notin Vars(\beta)$. *A rewrite rule* $\alpha \rightarrow \beta$ *is* ***non-erasing*** *if* $Vars(\alpha)=Vars(\beta)$.

Unless otherwise indicated, the rewrite systems considered in this dissertation are linear and contain only N-patterns.

A simple example of a rewrite system is the system of Figure 2.6 describing a commutative operator and its "reduction".



A Simple Rewrite System

Figure 2.6

Rewrite rules are applied to an input tree to obtain a new tree.

Definition 2.20 *Let* R *be a rewrite system over* Op, *and let* $r = \alpha \rightarrow \beta$ *be a rewrite rule. Let* T *be a tree over* Op, *and let* p *be a position in* T. r *is* ***applicable at*** p *in* T *if* α *matches at* $T_{@p}$ *for some assignment* σ. *If so, the* ***application*** *of* r *at* p *is* $T_{@p \leftarrow \sigma(\beta)}$. *A* ***rewrite application*** *is a pair* $<r,p>$ *where* r *is a rewrite rule and* p *is a position.*

Trees that cannot be rewritten are important.

Definition 2.21 *A tree* T *is said to be* ***irreducible*** *with respect to some rewrite system* R, *when no rewrite rule in* R *applies to any position in* T.

The ***language generated*** *by a rewrite system and an initial tree* T *is the set of all the irreducible trees* T' *into which* T *can be rewritten.*

Rewrite rules are normally applied in a sequence. The appropriate notion is that of a ***rewrite sequence.***

Definition 2.22 *Let* R *a rewrite system over* Op. *A* ***rewrite sequence*** *for* R *is a sequence* τ *of rewrite applications. If* $\tau = <r_0,p_0> \cdots <r_n,p_n>$ *is a rewrite sequence, then* τ *is* ***applicable*** *to a tree* T *if* r_0 *is applicable to* $T_{@p_0}$ *and its application yields* T_1, *and for* $1 \leq i < n$, r_i *is applicable to* $(T_i)_{@p_i}$ *and its application is* T_{i+1}. *The application of* τ *to* T *is denoted* $\tau(T)$ *and is* T_{n+1}.

If τ *is a rewrite sequence and* T *is a tree to which* τ *is applicable, then* τ *is said to be* valid at T. *A rewrite sequence is said to be* ***valid*** *if there is some tree* T *at which it is valid. Two rewrite sequences* τ_1, *and* τ_2 *are said to be* ***equivalent*** *if, for every tree* T, τ_1 *is valid at* T *if and only if* τ_2 *is valid at* T, *and, when valid,* $\tau_1(T)=\tau_2(T)$. *The* ***length*** *of a rewrite sequence is the number of rewrite applications in it.*

Referring to the example of Figure 2.6 a valid rewrite sequence for $+(+(const,reg),const)$ is $\tau = <r_1,1> <r_0,1> <r_0,\varepsilon>$. The application of τ to the tree produces *reg*. The length of τ is 3.

A rewrite sequence may involve only a subtree of the original input tree. In this case, the rewrite sequence can be ***restricted***, to obtain another sequence.

Definition 2.23 *Let R be a rewrite system over Op, T be a tree over Op, p a position in T, and $t = T_{@p}$. Let τ be a rewrite sequence for T such that all the rewrite applications in τ have positions below p. The* ***restriction*** *of τ to p, $\tau_{@p}$, is the sequence of applications identical to τ except that every position is stripped of the initial sequence corresponding to p. $\tau_{@p}(t)$ is frequently denoted simply as $\tau(t)$.*

It is easily shown that:

Proposition 2.13 *Let T be a tree over Op, p a position in T, and τ a rewrite sequence with all its applications below p. Then, $\tau(T) = T_{@p \leftarrow \tau_{@p}(T)}$. Equivalently, $\tau_{@p}(T_{@p}) = \tau(T)_{@p}$.*

Loops can always be removed from a rewrite sequence.

Proposition 2.14 *Let R be a rewrite system for Op, and let τ_1 be a valid rewrite sequence over R. τ_1 is said to "loop" if it contains a proper subsequence τ_2 such that, for any tree T for which τ_2 is valid, $\tau_2(T) = T$. For any rewrite sequence over R with a loop there is a rewrite sequence over R which is an order-preserving subsequence of the original one and does not loop.*

The proposition follows by repeatedly "removing" the undesired subsequence. Obviously the removal of a loop leaves a rewrite sequence with a length strictly smaller than the original one.

The notion of the "composition" of rewrite applications is used in later chapters.

Definition 2.24 *Let R be a rewrite system over a set of operators Op, and let τ be a valid rewrite sequence over R. The* ***composition*** *of τ is a rewrite application $<r_\tau, p>$, with r_τ possibly not in R, such that, for any tree T, r_τ is applicable at position p in T if and only if τ is applicable to T, and, if so, $\tau(T) = r_\tau(T)$.*

Not every rewrite sequence has a composition that is expressible with a rewrite rule using N-patterns, but the following condition is sufficient.

Proposition 2.15 *Let R be a (linear) rewrite system over Op, let $r_i = \alpha_i \rightarrow \beta_i$, for $i = 1,2$ be two rewrite rules in R, and let $\tau = <r_1, p_1><r_2, p_2>$ be a rewrite sequence. Let P_{β_1} be the set of positions in β_1, and let P_{α_2} be the set of positions in α_2. If $\{p_1 /\!/ q \mid q \in P_{\beta_1}\} \cap \{p_2 /\!/ q \mid q \in P_{\alpha_2}\} \neq \emptyset$, then τ has a composition.*

Proof There are two cases to consider.

Case 1. Let p_2 be a descendant of p_1; that is, $p_2 = p_1 /\!/ q$, for some position q. Let $X_1, \ldots, X_n$ be those variables in β_1 whose position is of the form $q /\!/ q_1, \ldots, q /\!/ q_n$, (that is, their positions are descendents of q). Let $\alpha_2^1, \ldots, \alpha_2^n$ be $(\alpha_2)_{@q_1}, \ldots, (\alpha_2)_{@q_n}$. If r_3 is $\alpha_3 \rightarrow \beta_3$, with α_3 being $(\alpha_1)_{X_1 \leftarrow \alpha_2^1 \cdots X_n \leftarrow \alpha_2^n}$, and β_3 being $(\beta_1)_{@q \leftarrow \beta_2}$, the composition of τ is $<r_3, p_1>$. This can be proved by a straightforward case analysis of the tree positions.

Case 2. Let p_1 be a descendent of p_2, that is $p_1 = p_2 /\!/ q$ for some position q. Let $X_1, \ldots, X_n$ be those variables in α_2 whose position in α_2 can be expressed as $q /\!/ q_1, \ldots, q /\!/ q_n$. Let $\beta_1^1, \ldots, \beta_1^n$ be $(\beta_1)_{@q_1}, \ldots, (\beta_1)_{@q_n}$. The composition of τ is $<r_3, p_2>$ where $r_3 = \alpha_2 \rightarrow \beta_3$, where $\alpha_3 = (\alpha_2)_{@q \leftarrow \alpha_1}$, and $\beta_3 = (\beta_2)_{X_1 \leftarrow \beta_1^1 \cdots X_n \leftarrow \beta_1^n}$. Again, this can be proved by a case analysis of the tree positions. □

The linearity of the rewrite rules is critical for the validity of Proposition 2.15. The proposition can be used iteratively to provide a sufficient condition for a sequence to have a composition.

Another useful notion that will be used later in the dissertation is that of a "permutation" of a rewrite sequence.

Definition 2.25 *Let r_0 and r_1 be two rewrite rules in R, and let $<r_0,p_0><r_1,p_1>$ be a valid rewrite sequence for R. An "exchange" of the two applications is an equivalent rewrite sequence of the form: $<r_1,p_2><r_0,p_3>$.*

Let τ_1 and τ_2 be two valid rewrite sequences in R. τ_1 is a ***permutation*** *of τ_2 if τ_1 can be obtained from τ_2 through a sequence of exchanges.*

The definition of exchange is quite loose: it only requires that the order in which the two rewrite applications appear is inverted, and that the net effect remains unchanged. There are some cases where two rewrite applications can be exchanged regardless of the particulars of the rewrite rules themselves. The next proposition is valid even in non-linear rewrite systems:

Proposition 2.16 *Let R be any rewrite system over Op, and let $<r_0,p_0><r_1,p_1>$ be a valid rewrite sequence over R, with $r_0=\alpha_0\rightarrow\beta_0$, and $r_1=\alpha_1\rightarrow\beta_1$. If p_0 is not a prefix of p_1 and p_1 is not a prefix of p_0, then $<r_0,p_0><r_1,p_1>$ is equivalent to $<r_1,p_1><r_0,p_0>$.*

If the rewrite system is linear, we can also add some additional sufficient conditions:

Proposition 2.17 *Let R be a linear rewrite system over Op, and let $<r_0,p_0><r_1,p_1>$ be a valid rewrite sequence over R, with $r_0=\alpha_0\rightarrow\beta_0$, and $r_1=\alpha_1\rightarrow\beta_1$. Then:*

(1) if p_1 is $p_0 // q // t$ where q is a position in β_0 corresponding to a variable X, and s is the position of X in α_0, then $<r_0,p_0><r_1,p_1>$ is equivalent to $<r_1,p_0 // s // t><r_0,p_0>$

(2) if p_0 is $p_1 // q // t$, where q is a position in α_1 corresponding to a variable X and s is the position of X in β_1 and r_1 is non-erasing in X, then $<r_0,p_0><r_1,p_1>$ is equivalent to $<r_1,p_1><r_0,p_1 // s // t>$

The proofs of the last two propositions are left to the reader.

Problems in Rewrite Systems

Two traditional properties involving a rewrite system, R_{Op}, and a set of input trees, $L \subset L_{Op}$, are termination and confluence. These properties lead to corresponding decidability problems:

Definition 2.26 *Let R be a rewrite system over a set of operators Op, and L a set of trees over Op. The TERMINATION problem for R and L is to determine whether for some tree $T \in L$ there is an infinite rewrite sequence applicable to T.*

Definition 2.27 *Let R be a rewrite system over a set of operators Op, and L a set of trees over Op. The CONFLUENCE problem for R and L is to determine whether, for all trees $T \in L$, all rewrite sequences applicable at T producing an irreducible tree produce the same one.*

The previous two properties are quite standard in the literature, see, for instance, [Ros73]. A non-standard problem is the reachability problem, a construction problem that comes in several variations.

Definition 2.28 *Let R be a rewrite system over Op, and let L_i and L_o be two sets of trees over Op. The REACHABILITY problem for R, L_i, and L_o is, given $T \in L_i$ and $T' \in L_o$, to determine whether there is a rewrite sequence τ for R applicable at T such that $\tau(T)=T'$, and, if so, to produce one such τ.*

If L_o is a singleton $\{G\}$, then the REACHABILITY problem is called the ***fixed goal*** *REACHABILITY problem, and G is called the* ***goal****.*

Without loss of generality, the fixed-goal REACHABILITY problem can be further restricted by assuming that the goal tree is a nullary symbol in Op. All the applications of REACHABILITY in this dissertation are fixed goal REACHABILITY problems where the goal is restricted as indicated above. Hence, from here on, unless otherwise indicated, REACHABILITY means fixed goal REACHABILITY.

Note that, if the set L_o is finite, the variable-goal REACHABILITY problem is equivalent to a fixed-goal REACHABILITY problem: the equivalent problem is obtained by extending the rewrite system with new rewrite rules of the form $t \rightarrow G$ for all t in L_o, where G is the fixed goal and is a new nullary operator that did not appear in R.

BLOCKING is a decision problem closely related to REACHABILITY. Given a rewrite system R over Op, a tree language, $L_i \subset L_{Op}$, and a fixed goal G, the BLOCKING problem is to determine whether there is a tree $T \in L_i$ that cannot be rewritten to G.

The above problems are unsolvable (undecidable) for general rewrite systems since they can be used to solve the halting problem, but they are solvable for some classes of rewrite systems.

Rewrite Systems and Generating Devices

Some restricted types of rewrite systems are related to the tree languages mentioned in Section 2.2.

Definition 2.29 *Let N be a set of nullary operators, called the* ***non-terminals****, disjoint from set of operators T called the* ***terminals****. A* ***regular tree grammar*** *over $T \cup N$, is a rewrite system where all rewrite rules are of the form $N \rightarrow t$, where N is a variable, and t is any tree over $T \cup N$.*

The language ***generated*** *by a regular tree grammar is the language generated by its rewrite system applied to the start non-terminal.*

It is simple to prove that:

Proposition 2.18 *A tree language is in RECOG if and only if it can be generated by a regular tree grammar.*

Guessarian [Gue81] shows that there is a similar situation with push down tree automata.

Definition 2.30 *Let N be a set of nullary operators, called the* ***non-terminals****, disjoint from set of operators T called the* ***terminals****. A* ***context-free tree grammar*** *over $T \cup N$ is a rewrite system where all rewrite rules are of the form: $op(N_1, \ldots, N_n) \rightarrow \rho$, where, for $1 \leq i \leq n$, $N_i \in N$, and ρ is a tree containing operators from T and, possibly, also $N_1, \ldots, N_n$.*

For this generating device, the corresponding accepting device is the push-down tree automata briefly presented in Section 2.2

Definition 2.31 *A tree language is generated by a context-free tree grammar if and only if it is recognized by a push down tree automata.*

2.4. Finite State Tree Transformations

Finite state automata for strings can be extended to obtain transforming devices over (string) languages called transducers. The accepting automata of Section 2.2 can be modified in a similar way to obtain tree transformation devices that are called finite state tree transducers. The formalizations and results of this section are from [Eng75]; most of them are not used elsewhere in this dissertation and are included here only for completeness. The transducers are described using a variation of a rewrite system.

Definition 2.32 *A **bottom-up (finite) tree transducer**, M, is a tuple (R, Op_I, Op_O, Q, Q_f), where R is a rewrite system over $Op_I \cup X \cup Q$; with X the variables with arity 0, and Q the states, with arity 1; Op_I is the set of input symbols; Op_O is the set of output symbols; $Q_f \subset Q$ are the final states; and the rewrite rules in R are of the form:*

$$op(q_1(X_1),\ldots,q_n(X_n)) \rightarrow q(t),$$

for $op \in Op_I$ of arity n, $q_i \in Q$, $X_i \in X$, and $t \in T_{Op_O \cup \{X_1,\ldots,X_n\}}$.

*The set of **tree transformations realized by** a transducer M is*

$$\{(s,t) \mid s \in L_{Op_I},\ t \in L_{Op_O},\ \text{and } s \text{ rewrites into } q(t) \text{ for some } q \in Q_f\}$$

The set of transformations realized by M is denoted by $T(M)$. The class of transformations realized by bottom-up finite state tree transducers will be denoted by $B\text{-}ftt$.

As an example,

$$a \rightarrow q(c)$$
$$b(q(X_1)) \rightarrow q(d(X_1, X_1))$$

are the rewrite rules of a transducer mapping trees over $\{a, b\}$ into those over $\{c, d\}$. The transducer ***implements*** the tree homomorphism defined by:

$$h(a) \triangleq c$$
$$h(b) \triangleq d(X_1, X_1)$$

T-fsa lead to a similar definition for a transducer:

Definition 2.33 *A **top-down (finite) tree transducer**, M, is a tuple (R, Op_I, Op_O, Q, Q_s), where R is a rewrite system over $Op_I \cup X \cup Q$; X, Op_I, and Op_O are as in def 2.32; $Q_s \subset Q$ are the starting states; and, the rewrite rules in R are of the form:*

$$q(op(X_1, \ldots, X_n)) \rightarrow t,$$

for $op \in Op_I$ of arity n, $q \in Q$, $x_i \in X$, and $t \in T_{Op_O \cup \{q(X_1),\ldots,q(X_n)\}}$.

*The set of **tree transformations realized by** a transducer M is*

$$\{(s,t) \mid s \in L_{Op_I},\ t \in L_{Op_O},\ \text{and } q(s) \text{ rewrites into } t \text{ for some } q \in Q_s\}$$

The class of transformations realized by top-down finite state tree transducers will be denoted by $T\text{-}ftt$.

As an example,

$$q(b(X_1)) \rightarrow d(q(X_1), q(X_1))$$
$$q(a) \rightarrow c$$

are the rewrite rules of a transducer that realizes the same transformation as the previous example.

Bottom-up and top-down transducers are generalizations of the transformational devices presented in the Section 2.2. As a set, every function between tree languages that can be described using those transformational devices can be described using both bottom-up and top-down transducers.

Proposition 2.19 *Both B-ftt and T-ftt include:*

REL	*relabelings*
FTA	*(t,t) such that $t \in RECOG$*
HOM	*homomorphisms*

LHOM linear homomorphisms

The capabilities of bottom-up transducers and top-down transducers are incomparable, in the sense that neither set is included in the other.

Proposition 2.20 *B–ftt and T–ftt are incomparable. Moreover, both B–ftt and T–ftt are not closed under (functional) composition.*

Both *T–ftt* and *B–ftt* can be modified as follows:

Definition 2.34 *A transducer in T–ftt or in B–ftt is said to be* **linear** *if the output pattern of each rewrite rule is a linear pattern. It is said to be* **non-deleting** *if the output pattern of each rewrite rule contains instances of all the variables appearing in the input pattern.*

A transducer in B–ftt is said to be **(partial) deterministic** *if, for each op* $\in$ ***Op*** *with arity n, there is at most one rewrite rule with left hand side* $op(q_1(X_1), \ldots, q_n(X_n))$. *It is said to be* **total deterministic** *if for each op* $\in$ ***Op*** *there is exactly one such rewrite rule.*

A transducer in T–ftt is said to be **(partial) deterministic** *if, the set of initial states is a singleton, and, for each op* $\in$ ***Op*** *with arity n, there is at most one rewrite rule with left hand side* $q(op(X_1, \ldots, X_n))$. *It is said to be* **total deterministic** *if for each op* $\in$ ***Op*** *there is exactly one such rewrite rule.*

The classes of transformations implementable by each one of the above restrictions are denoted by T-ftt and B-ftt prepended by one of L, N, D, or D_p.

The classes of transformations definable by top-down and bottom-up transducers restricted as suggested above are still incomparable:

Proposition 2.21 *LB-ftt, the class of transformations implementable with a linear bottom-up ftt, and LT-ftt, the class of transformations implementable with a top-down ftt, are incomparable.*

DB-ftt and DT-ftt are incomparable.

[Eng75] contains several decomposition propositions. A simple class transformations is useful in them.

Definition 2.35 *A* ***top-down relabeling*** *is a top-down transducer in which all the rewrite rules have the form:*

$$q(a(X_1, \ldots, X_n)) \rightarrow b(q_1(X_1), \ldots, q_n(X_n))$$

for $a \in Op_I$ *of arity n,* $b \in Op_O$ *of arity n, and* $q, q_i \in Q$.

A ***bottom-up relabeling*** *is a bottom-up transducer in which all the rewrite rules have the form:*

$$a(q_1(X_1), \ldots, q_n(X_n)) \rightarrow q(b(X_1, \ldots, X_n))$$

where a, b, q, q_i *are as above.*

The class of transformations implementable by top-down relabeling is denoted by T-QREL. That implementable by bottom-up relabeling is denoted by B-QREL. DT-QREL and DB-QREL are the corresponding deterministic transducers. The non-deterministic versions of the relabelings are equivalent since the corresponding top-down and bottom-up automata are equivalent, but the deterministic versions are not. In particular:

Proposition 2.22 *The following equations hold:*

DB-QREL $\subseteq$ *B-QREL*

$B\text{-}QREL = D\text{-}QREL$

$DB\text{-}QREL \subseteq DT$

$DT\text{-}QREL \subseteq DB$.

Proof Left to the reader. If in trouble, consult [Eng75]. □

Relabelings play an important role in the decomposition of more complex transformations:

Proposition 2.23 *Let T^* denote the set of zero or more compositions of T with itself. The following equations are true:*

$B\text{-}ftt \subseteq QREL \circ HOM$.

$B\text{-}ftt = LB\text{-}ftt \circ HOM$

$T\text{-}ftt \subseteq HOM \circ LT\text{-}ftt$.

$T^* = B^* = (REL \cup FTA \cup HOM)^*$

In some sense, bottom-up transducers are more powerful than top-down transducers:

Proposition 2.24 *The following two equations are true:*

$LT\text{-}ftt \subseteq LB\text{-}ftt$.

$NLT\text{-}ftt = NLB\text{-}ftt$.

To close the gap between linear T-ftt and linear B-ftt, definition 2.33 can be modified allow a look-ahead. The input patterns used in that definition did not explicitly indicate the type of their variables. Hence, following our convention, their type is the default type. Dropping this restriction effectively increases the descriptive power of the mechanism.

Definition 2.36 *A **top-down finite transducer with look-ahead** is a tuple as in the top-down finite transducer except that the variables appearing in the rewrite rules of the rewrite system may have as types any set in RECOG.*

The class of transformations implementable by a top-down finite transducer with look-ahead is denoted by T'.

The gain in expressive power is indicated in the next proposition:

Proposition 2.25 *The following equations are true:*

$LT'\text{-}ftt = LB\text{-}ftt$.

$T'\text{-}ftt$ and $B\text{-}ftt$ are incomparable.

$T'\text{-}ftt$ is not closed under composition.

$T'\text{-}ftt \subseteq DB\text{-}QREL \circ T\text{-}ftt$.

2.4.1. Labeled Bottom-Up Automata

A notion related to DB-QREL is that of "labeled B-fsa". This notion is used in Chapters 3 and 4 to provide algorithms for solving PATTERN MATCHING, but its main usefulness will not be apparent until Chapter 6[10].

[10] Chapter 5 can also be said to be based on LB-fsa, but this is pretty much a technicality since the labeling function is the identity.

Definition 2.37 *A (deterministic)* ***labeled bottom-up automaton*** *is a pair <A,L> where A is a (deterministic) B-fsa and L is a* ***labeling function*** *assigning to each state in A a subset of a given set of labels. Let St(t) denote the set of states assigned to a tree t by A (a singleton if A is deterministic), and Lb(t) the set of labels obtained by applying L to each state in St(t). The transformation described by a labeled bottom-up automaton is the set of all pairs (t,new_label(t)) where new_label() is defined recursively by:*

$$new\_label(op) \triangleq \langle op, Lb(op)\rangle, \text{ if } op \text{ is a nullary operator}$$
$$new\_label(op(t_1, \ldots, t_n) \triangleq \langle op, Lb(op)\rangle(new\_label(t_1), \ldots, new\_label(t_n))$$

The class of transformations definable by a labeled bottom-up automaton is denoted by LB-fsa; that of deterministic ones by DLB-fsa.

The standard notion of a B-fsa as an acceptor corresponds to a special case of a LB-fsa: one where the only two labels are ***accepted*** and ***non-accepted***. A B-fsa can also be seen as a translating LB-fsa by associating with each state a label that is the name of the state. This is the approach used in Chapter 3 to solve pattern matching for linear N-patterns. The general notion of LB-fsa is used in Chapter 5 to solve the REACHABILITY problem for a class of rewrite systems.

Deterministic labeled automata have the same power as DB-QREL, but non-deterministic LB-fsa are only as powerful as deterministic LB-fsa and, hence, less powerful than B-QREL.

Proposition 2.26 *DLB-fsa equals LB-fsa and DB-QREL. LB-fsa* $\subseteq$ *B-QREL.*

Proof All are easy and left to the reader □

An alternative definition for the notion of a "labeled B-fsa" could have associated the label with the transition function instead of with the state itself. The reader can verify that the resulting notion has the same expressive power as the one given here.

The algorithm of Figure 2.5 can be modified to obtain algorithms to minimize the number of states in a LB-fsa. to minimize the number of states in a DB-QREL.

For DB-QREL, if $Trans_n(op, st_1, \ldots, st_n)$ represents the new state and $Label_n(op, st_1, \ldots, st_n)$ the new label, the modification requires changing the routine *transfer_equivalent* so that the last *if* is replaced by:

let l^a be $Label_n(Op, st_{\pi_1{}^a}, \ldots, st_{\pi_n{}^a})$;
let l^b be $Label_n(Op, st_{\pi_1{}^b}, \ldots, st_{\pi_n{}^b})$;
if $c^a \neq c^b$ *or* $l^a \neq l^b$ *then*

CHAPTER 3

Matching Linear N-Patterns

These unhappy times call for the building of plans ...
that build from the bottom-up and not from the top-down ...

Radio address
April 7th, 1932

[Franklin Delano Roosevelt [1882-1945]]

Linear N-patterns are the simplest type of patterns studied in this dissertation. Their importance is twofold. First, the techniques used to perform pattern matching on linear N-patterns form the basis for extensions to deal with more complex types of patterns. Second, some very important applications use only linear patterns (Chapter 5).

All the pattern matching algorithms investigated in this chapter are based on the B-fsa of Chapter 2. In all cases, the idea is to assign to each node in the subject tree its ***match set***. This is the set containing all patterns **and** subpatterns (subtrees of a pattern) matching at the node. PATTERN MATCHING is then solved by extracting from these match sets the patterns. Section 3.1 defines match sets and related notions.

The main pattern matching algorithms of this chapter are based on LB-fsa (Section 2.4.1). The first type is a non-deterministic LB-fsa called a ***subpattern LB-fsa*** (Section 3.2). The states in this LB-fsa are individual patterns and subpatterns, and the labeling function assigns to a subpattern either the empty set or the singleton containing that subpattern, depending on whether the subpattern is or is not a pattern. The second type is a deterministic LB-fsa called a ***match set LB-fsa*** (Section 3.3). The states in this LB-fsa are the match sets themselves and the labeling function assigns to a match set the patterns it contains.

The algorithm using the match set LB-fsa is substantially faster than the one using the subpattern LB-fsa but requires tables that are bigger and take longer to construct. One way to compute the match set LB-fsa is to construct the subpattern LB-fsa first and then apply the algorithm of Figure 2.3 for converting a non-deterministic LB-fsa into a deterministic one. This is the approach presented in the second part of Section 3.3. An alternate approach to compute the match set LB-fsa, developed by David Chase, is presented and elaborated in Section 3.5.

The structure of the subsumption relation (Def. 3.1) plays an important role in the computation of the match set. Section 3.4 explores this role and introduces two new algorithms to compute match sets. The algorithms are based on the subsumption relation. They require smaller tables than a match set LB-fsa yet have smaller matching time than the subpattern LB-fsa, and their tables can be constructed faster.

Section 3.7 compares previous work with the results presented in this chapter. It discusses some other results in matching algorithms that are based on bottom-up traversals, as well as some that are based on top-down traversals. In general, algorithms based on top-down traversals seem to be intrinsically slower than the fastest algorithms based on bottom-up traversal. Chapter 8 contains some measurements for both bottom-up and top-down algorithms that solve REACHABILITY, a problem that Chapter 5 shows it is quite similar to PATTERN MATCHING.

Section 3.8 summarizes the results of this chapter.

3.1. Subpatterns and Match Sets

Since all the patterns considered in this chapter are linear N-patterns, they are frequently described simply as "patterns".

In bottom-up pattern matching, a subtree is characterized by its **match set**, the set of all the subpatterns matching at that subtree. This section introduces the notions required to describe algorithms based on match sets. Some of the notions are borrowed from [Kro75] and [HoO82]. The same terminology is used whenever possible and convenient.

Patterns and Subpatterns

Central to the development of the algorithms is the notion of a subpattern. The notion is influenced by the bottom-up bias of the matching algorithms. It is convenient to define operations and new relations between patterns in addition to the notion of equivalence of Definition 2.7.

Definition 3.1 *Let ρ_1 and ρ_2 be two patterns. They are* ***independent*** *($\rho_1 \sim \rho_2$) if there are trees T_1, T_2, T_3 such that ρ_1 matches at T_1 and at T_3, but not at T_2, and ρ_2 matches at T_2 and at T_3, but not at T_1. They are* ***inconsistent*** *($\rho_1 /\!/ \rho_2$) if there is no tree T such that ρ_1 matches at T and ρ_2 matches at T. ρ_1* ***subsumes*** *ρ_2 ($\rho_1 \geq \rho_2$) if for any tree T, if ρ_1 matches at T, then ρ_2 matches at T.*

If σ and σ' are two sets of patterns, we say that σ and σ' are equivalent if (1) every pattern in σ is equivalent to some pattern in σ', and (2) every pattern in σ' is equivalent to some pattern in σ.

If σ is a set of patterns, the $\equiv$-reduction of σ is an equivalent set of patterns σ' such that no two patterns in σ' are equivalent.

Clearly, the equivalent reduction of a set of patterns is unique up to equivalence.

Definition 3.2 *A* ***pattern set,*** *F over an operator set **Op** is any collection of patterns over **Op** such that no two patterns in F are equivalent.*

*Let F be a pattern set over **Op**. A* ***subpattern*** *of F is either a subterm of a pattern in F, or the pattern containing a single variable, X. Π_F is an $\equiv$-reduction of the set containing all subpatterns of F, and such that $X \in \Pi_F$, and $F \subset \Pi_F$.*

ρ_1 ***immediately subsumes*** *ρ_2 for a pattern set F ($\rho_1 >_i \rho_2$) if $\rho_1 \not\equiv \rho_2$, $\rho_1 \geq \rho_2$, and there is no subpattern $\phi \in \Pi_F$, such that $\rho_1 \not\equiv \phi$, $\rho_2 \not\equiv \phi$, $\rho_1 \geq \phi$ and $\phi \geq \rho_2$.*

If ρ is a pattern in Π_F, the immediate subsumption set of ρ, I_ρ, is the set of all those patterns ρ' in Π_F such that $\rho >_i \rho'$.

For a given F, Π_F is unique up to $\equiv$. In this dissertation, Π_F plays a role somewhat similar to that of a canonical representation for all the patterns that are subpatterns of patterns in F. Sets of patterns taken from Π_F are known to be $\equiv$-reduced, and set equivalence between two such sets is identical to set equality.

Since all sets Π_F are equivalent, we will not explicitly specify which one of them we use. The advantage of using Π_F instead of any fixed set of canonical representatives is that we can tailor which Π_F we use to the didactic requirements of our examples. Thus, in one particular case we could use $\{+(X, *(Y,R)), *(Y,R), X\}$ as our Π_F to emphasize that the second pattern in the set is a subpattern obtained from the first pattern in the set. Requiring $X \in \Pi_F$ and $F \subset \Pi_F$ simplifies many situations.

As an example of the relations defined above, if the original pattern set F is "$\{+(a,X),+(a,Y),+(b,X),+(a,a)\}$", then the following relationships hold: "$+(a,X) \sim +(X,b)$", "$+(a,X) /\!/ +(b,X)$", "$+(a,X) \geq X$", and, for Π_F, "$+(a,X) >_i +(a,a)$". Note that all the relations defined above are independent of the given pattern set, except for the notion of $>_i$: thus, if $+(a,a)$ had not been present in F, $+(a,X) >_i X$ would have been true.

The definition of $\geq$ satisfies:

Proposition 3.1 *$\geq$ is a partial order.*

Note that in a set of patterns, each pattern has a separate set of variables. For example, in the set $\{+(a,X), X\}$, the two uses of X are independent. It follows from Proposition 2.2 that we can always replace any pattern by an equivalent pattern that contains different variable names. Therefore, by systematic renaming, any pattern set is equivalent to a pattern set that is variable-disjoint.

A very useful notion is that of the meet of a set of patterns.

Definition 3.3 *[Kro75] The meet of two patterns ρ and ρ' ($\rho \oplus \rho'$) is a new pattern ϕ such that a tree is matched by ϕ if and only if it is matched by both ρ and ρ'*

As an example, the meet of $+(a,X)$ and $+(X,b)$ is $+(a,b)$. Given two patterns that are not inconsistent, their meet will always exist.

Proposition 3.2 *Let ρ_1 and ρ_2 be two patterns that are not inconsistent. Then, there is a pattern $\rho_3 = \rho_1 \oplus \rho_2$. In addition, $\rho_3 \geq \rho_1$ and $\rho_3 \geq \rho_2$.*

Proof Follows from the existence of a unique most general unifier. Without loss of generality, assume that ρ_1 and ρ_2 are variable-disjoint. If ρ_1 and ρ_2 are not inconsistent, then there is a subject tree T at which both match. This is equivalent to saying that ρ_1 and ρ_2 are unifiable. Let τ be the most general unifier of ρ_1 and ρ_2. Let δ be $\tau(\rho_1) = \tau(\rho_2)$. Then $\delta = \rho_1 \oplus \rho_2$. If δ matches at a tree T, then both ρ_1 and ρ_2 match at T: construct their substitutions from τ and the substitutions for δ. If both ρ_1 and ρ_2 match at T, then T provides a unifier for both ρ_1 and ρ_2. Since δ is the most general unifier, a substitution for it will also exist, and it will match at T □

Match Sets

Matches of subpatterns at the same tree are of interest in several ways. The following definitions are used throughout this and the next chapters.

Definition 3.4 *Let Op be an operator set, F be a pattern set over Op, and T be a tree over Op. The **match set associated** with T, σ_T, is the subset of Π_F containing those patterns that match at T. A subset σ of Π_F is called a **match set** if there is a subject tree T over Op with associated match set σ*[11].

For a given pattern set, the match set associated with any tree is unique up to choice of Π_F because Π_F is $\equiv$-reduced.

Note that not every subset of Π_F is a match set because a match set must include all subpatterns matching at some tree T, up to equivalence. For example, "$\{+(a,X)\}$" is a collection of matches, but not a match set because it is missing X, while "$\{+(a,X),+(b,X)\}$" is not a match set because there is no tree at which both patterns would match.

---

[11] Strictly, a match is an assignment of values to variables (Def. 2.6), but this assignment is uniquely implied by the pattern and the subject tree. Hence, the name of "match set" and "collection of matches" for sets of subpatterns, and the use of σ to denote them.

Definition 3.5 *The size of a pattern* ρ *is the cardinality of* P_ρ, *the set of all positions in* ρ. *Let* F *be a pattern set over an operator set* Op. *The size of* F *is the sum of the sizes of all the patterns in* F.

Proposition 3.3 *[HoO82] There are pattern sets* F *with a number of match sets exponential in the size of* F. *There are pattern sets* F *with match sets as large as the number of patterns in* Π_F. *The number of patterns in* Π_F *is no larger than the size of* F.

Proof As an example of the first two properties consider a pattern set as follows. It will contain $O(2^n)$ patterns. Each pattern will "mark" the presence of a single, distinguished, atom b at a specified position in its leaves, while all the internal nodes are identical binary operators labeled a. The patterns will differ in the position of the b. The patterns will be of height $O(n)$ and will have $O(2^n)$ leaves.

Now consider subjects that are binary trees with all the internal nodes labeled a, with leaves marked either b or c. There are $O(2^{2^n})$ such trees, and each one has a different match set.

Formally, define a family of balanced binary trees $p^i{}_j$, with $0 \le i$ and $0 \le j \le 2^i$, by

$$p^0{}_0 \triangleq X$$

$$p^0{}_1 \triangleq b$$

$$p^{i+1}{}_j \triangleq a(p^i{}_j, p^i{}_0), \qquad \textit{for } 0 \le j \le 2^j$$

$$p^{i+1}{}_j \triangleq a(p^i{}_0, p^i{}_{j-2^i}), \qquad \textit{for } 2^i < j \le 2^{i+1}$$

The pattern set $S_n \triangleq \{p^n{}_i \mid 1 \le i \le 2^n\}$ is of size $O(2^n)$. When used with a fully balanced tree of depth n with all its leaves b, there will be $O(2^n)$ elements in the match set. A pattern will be in the match set of the root if and only if the i-th leaf from left to right has a b. Since there are $O(2^{2^n})$ such trees, that is the number of distinct match sets

The result on the number of distinct subpatterns is immediate from the association of every subpattern with a subterm in a pattern in F. □

To investigate the properties of match sets, it is convenient to characterize them in two different ways. The first one characterizes a match set of F through a single pattern that may not be equivalent to any pattern in Π_F.

Definition 3.6 *Let* F *be a pattern set over* Op. *Let* ρ *be a pattern over* Op *(maybe not in* F*). We define the set* $M(\rho)$ *to be the subset of* Π_F *that contains every pattern in* Π_F *subsumed by* ρ.

Note that for a given Π_F $M(\rho)$ is unique since Π_F is $\equiv$-reduced.

Proposition 3.4 *Let* F *be a pattern set over* Op. *Let* σ *be a match set of* F. *Then there is a pattern* ρ_σ *(unique up to* $\equiv$*) such that* $\sigma = M(\rho_\sigma)$. *For any two match sets* σ *and* σ', $\rho_\sigma \equiv \rho_{\sigma'}$ *if and only if* $\sigma = \sigma'$

Proof Let σ' be a variable-disjoint set of patterns equivalent to σ. Let ρ_σ be the meet of all the patterns in σ'. ρ_σ is well defined because no two patterns in σ' can be inconsistent, since they all match at the same tree. The property follows from the definitions of match set and $\ominus$. □.

The second characterization uses only patterns in Π_F. In this case it may be necessary to use a collection of patterns to characterize the match set. The members of the collection are called the representatives.

Proposition 3.5 *Let F be a pattern set, and let σ be a match set of F. Define R_σ the set of representatives of σ, as that subset of Π_F containing all those patterns ρ such that ρ is in σ and for no other pattern ρ' in σ, $\rho' \geq \rho$. For any two match sets σ and σ', $R_\sigma = R_{\sigma'}$ if and only if $\sigma = \sigma'$.*
Proof Left to the reader. □

For example, let the match set be $\{+(a,X),+(Y,b),Z\}$. A match set associated with the input tree $+(a,b)$ is "$\{+(a,X),+(Y,b),Z\}$", and a set of representatives for it is "$\{+(a,X),+(Y,b)\}$".

Both R_σ and ρ_σ uniquely characterize the match set σ and can be used to represent it. In general, ρ_σ is a more concise representation than R_σ, but it may require (a potentially exponential number of) new patterns not previously present in Π_F. There are some conditions that guarantee that the exponential explosion will not occur. A first condition is the following:

Definition 3.7 *[HoO82] A pattern set F is simple if there are no two patterns in Π_F that are independent.*

For instance "$\{+(a,X),+(Y,b),+(a,b)\}$" is not a simple pattern set, whereas "$\{+(a,X),+(a,b)\}$" is. The relationship between the simple pattern set property and match sets is developed in the following propositions.

Definition 3.8 *The immediate subsumption graph $G >_i$ of a pattern set F is the directed acyclic graph in which the nodes are the patterns in Π_F, and there is an edge from a node ρ to a node ρ' if $\rho >_i \rho'$ in F.*
Proposition 3.6 *[HoO82] If F is a simple pattern set, the graph obtained by reversing the direction of the edges of the graph $G >_i$ induced by F is a forest.*
Proof By the definition of a forest, the above condition is equivalent to saying that there are no subpatterns ρ_1, ρ_2, and ρ_3 such that $\rho_1 >_i \rho_2$, and $\rho_1 >_i \rho_3$. If such ρ_1, ρ_2, and ρ_3 exist, ρ_2 and ρ_3 would not be independent, contradicting the definition of a simple pattern set. □.

Proposition 3.7 *[HoO82] Let F be a pattern set. If the graph obtained by inverting the direction of the edges in a $G >_i$ graph is a forest, then the number of distinct match sets is equal to the number of patterns in Π_F.*
Proof Immediate from the meaning of $G >_i$. □

The notion of a simple pattern set is unnecessarily restrictive. A more useful definition is the following:

Definition 3.9 *[Kro75] A pattern set F is a closed template forest (CTF) if for every two non-inconsistent patterns ρ and ρ' in Π_F, their meet ϕ is equivalent to some pattern in Π_F.*

It is easy to show that CTF is a less restrictive notion than that of a simple pattern set.

Proposition 3.8 *If a set of patterns is simple, then it is CTF.*
Proof If two subpatterns are neither independent nor inconsistent one of them must subsume the other and be identical to their meet. □

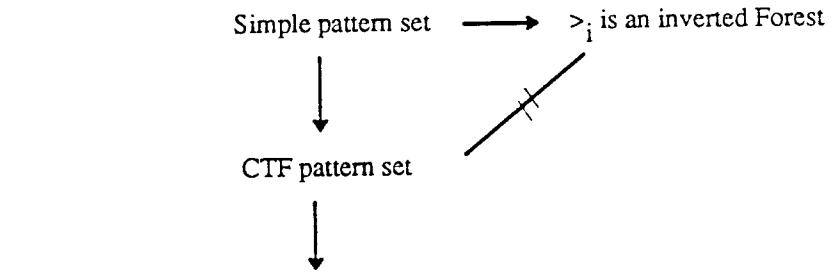
To see that the reverse implication is false, just consider the pattern set "$\{+(a,X),+(Y,b),+(a,b)\}$", which is CTF but not simple.

Proposition 3.9 *Let F be a pattern set. F is CTF if and only if for every match set σ, R_σ is a singleton.*
Proof The only if part is immediate since the meet ρ_σ of all the patterns in σ is equivalent to a pattern in Π_F.

To prove the if part, assume that, for every match set σ, R_σ is a singleton. Let ρ_1 and ρ_2 be two non-inconsistent patterns in Π_F. Let $t = \rho_1 \oplus \rho_2$, and let σ be the match set at t. By hypothesis $R_\sigma = \{\rho\}$ for some ρ. Then $t \geq \rho_1$, and $t \geq \rho_2$ by definition of meet. $t \geq \rho$ because ρ matches at t. $\rho \geq \rho_1$ and $\rho \geq \rho_2$ by definition of R_σ, and from this and the definition of meet, $\rho \geq t$. From the antisymmetry of $\geq$, $t \equiv \rho$. □

Corollary 3.1 *Let F be a pattern set. The number of distinct match sets of F is, at most, equal to the number of patterns in Π_F.*

The properties established in Propositions 3.6 to 3.9 are summarized in Figure 3.1. The relationship between "CTF pattern set" and "$>_i$ is an inverted forest" is proved by finding two pattern sets each one satisfying one property but not the other, and is left as an exercise to the reader.



Summary of Pattern Classes

Figure 3.1

The algorithms presented in this chapter use different types of B-fsa to encode information that is used in the main operation in bottom-up matching: combining the match sets associated with the children of a node to obtain the match set for the node itself. The following proposition contains several facts related to this computation.

Proposition 3.10 *Let F be a linear pattern set over Op, let op be an operator in Op, let $T = op(T_1, \ldots, T_n)$ be a tree, let $\sigma_1, \ldots, \sigma_n$ be the match sets of $T_1, \ldots, T_n$ respectively, and let $\rho_1, \ldots, \rho_n$ be patterns equivalent to the meets of $\sigma_1, \ldots, \sigma_n$ respectively, where the variables are renamed, if necessary, so that $\{\rho_1, \ldots, \rho_n\}$ is variable-disjoint. Recall that σ_T denotes the match set at T, and $M(\rho)$ denotes the set of all patterns in Π_F subsumed by ρ. The following assertions are true:*

(1) Let $\psi_1, \ldots, \psi_n$ be any n variable-disjoint patterns. Then, $\{\phi \mid op(\psi_1, \ldots, \psi_n) \geq \phi\} \equiv \{X\} \cup \{op(\phi_1, \ldots, \phi_n) \mid \psi_1 \geq \phi_1 \wedge \psi_2 \geq \phi_2 \wedge \cdots \wedge \psi_n \geq \phi_n \wedge \{\phi_1, \phi_2, \cdots, \phi_n\}$ is variable-disjoint $\}$.

(2) $\sigma_T = M(op(\rho_1, \ldots, \rho_n))$.

(3) *If* $\rho_1 \equiv X_1, \ldots, \rho_n \equiv X_n$, *and* $op(X_1, \ldots, X_n)$ *is not equivalent to a pattern in* Π_F, *then* $\sigma_T =^{12} \{X\}$.

(4) *Let* $I_{\rho_i} = \{\rho_i^1, \ldots, \rho_i^{n_i}\}$ *denote the immediate subsumption set of each* ρ_i. $M(op(\rho_1, \ldots, \rho_n)) = \{ op(\rho_1, \ldots, \rho_n)$ *if* $op(\rho_1, \ldots, \rho_n) \in \Pi_F \} \cup \bigcup_{i=1}^{n} \bigcup_{j=1}^{n_i} M(op(\rho_1, \ldots, \rho_{i-1}, \rho_i^j, \rho_{i+1}, \ldots, \rho_n))$.

Proof

All proofs are done for $n = 2$, to reduce the notation burden. The generalizations are straightforward.

(1) follows directly from properties of subsumption and of linear patterns.

(2) can be proved by the following sequence of steps.

$\sigma_T = \{X\} \cup \{ op(\phi_1, \phi_2) \in \Pi_F$ *matching at* $T \}$,
$= \{X\} \cup \{ op(\phi_1, \phi_2) \in \Pi_F \mid \phi_1$ *matches at* $T_1 \wedge \phi_2$ *matches at* $T_2\}$,
$= \{X\} \cup \{ op(\phi_1, \phi_2) \in \Pi_F \mid \rho_1 \geq \phi_1 \wedge \rho_2 \geq \phi_2\}$,
$= \{ \phi \mid op(\rho_1, \rho_2) \geq \phi$ and $\phi \in \Pi_F\}$, by (1),
$= M(op(\rho_1, \rho_2))$, by definition.

(3) is straightforward

(4) can be proved by the following sequence of steps.

$M(op(\rho_1, \rho_2)) = \{X\} \cup \{ op(\phi_1, \phi_2) \in \Pi_F \mid \rho_1 \geq \phi_1 \wedge \rho_2 \geq \phi_2\}$, by (1),
$= \{X\} \cup \{ op(\phi_1, \phi_2) \in \Pi_F \mid \rho_1 > \phi_1 \wedge \rho_2 \geq \phi_2\} \cup \{X\} \cup \{op(\phi_1, \phi_2) \in \Pi_F \mid \rho_1 \geq \phi_1 \wedge \rho_2 > \phi_2\} \cup \{op(\rho_1, \ldots, \rho_n)$, *if* $op(\rho_1, \ldots, \rho_n) \in \Pi_F\}$, by definition of $\geq$ and $>$,
$= \{X\} \cup \{ op(\phi_1, \phi_2) \in \Pi_F \mid \rho_1 > \phi_1 \wedge \phi_1 \in \Pi_F \wedge \rho_2 \geq \phi_2\} \cup \{X\} \cup \{op(\phi_1, \phi_2) \in \Pi_F \mid \rho_1 \geq \phi_1 \wedge \rho_2 > \phi_2 \wedge \phi_2 \in \Pi_F \} \cup \{ op(\rho_1, \ldots, \rho_n)$, *if* $op(\rho_1, \ldots, \rho_n) \in \Pi_F \}$, by property of Π_F,
$= \{X\} \cup \{ op(\phi_1, \phi_2) \in \Pi_F \mid (\rho_1^1 \geq \phi_1 \vee \cdots \vee \rho_1^{n_1} \geq \phi_1) \wedge \rho_2 \geq \phi_2\} \cup \{X\} \cup \{ op(\phi_1, \phi_2) \in \Pi_F \mid \rho_1 \geq \phi_1 \wedge (\rho_{21} \geq \phi_2 \vee \cdots \vee \rho_2^{n_2} \geq \phi_2) \} \cup \{ op(\rho_1, \ldots, \rho_n)$, *if* $op(\phi_1, \ldots, \phi_n) \in \Pi_F \}$, by definition of $>_i$,

Which, by distribution of $\vee$ and definition of M produces the desired result. □

Assertions (2), (3), and (4) above provide a recursive algorithm to compute the match set associated with a tree given the match sets associated with its immediate subtrees. For example, if the pattern set is "$\{foo(V_1, *(Z,W), V_3), foo(*(X,Y), V_2, *(U,R))\}$", then the match set of "$foo(*(X,Y), *(Z,W), *(U,R))$" is equivalent to the union of the match sets of "$foo(V_1, *(Z,W), *(U,R))$", "$foo(*(X,Y), V_2, *(U,R))$", and "$foo(*(X,Y), *(Z,W), V_3)$". Of these, the match set of $foo(*(X,Y), V_2, *(U,R))$ is represented directly by a pattern in Π_F equivalent to that pattern, and repeated applications of the above assertions to the other two terms both yield "$foo(V_1, *(Z,W), V_3)$". Hence, the set of representatives of "$foo(*(X,Y), *(Z,W), *(U,R))$" is equivalent to "$\{foo(*(X,Y), V_2, *(U,R)), foo(V_1, *(Z,W), V_3)\}$". Section 3.4 analyzes in some more detail this recursive algorithm.

Bottom-up matching of linear N-patterns is based on the following proposition:

---

[12] An $\equiv$ is not necessary because $X \in \Pi_F$.

Proposition 3.11 *Let F be a set of linear patterns, and let $OP_0, \ldots, OP_n$ be the sets of operators of F of arity 0 up to n. Let A be the set of match sets of F. Then, for each k from 0 to n there exist functions f_k from $OP_k \times A^k$ (k+1-tuples) into A such that: for all trees T with root op of arity k and children $T_1, \ldots, T_k$, if σ_i is the match set at T_i, then the match set at T is $f_k(op, \sigma_1, \ldots, \sigma_k)$.*

Proof The key observation for the proof is that a substitution for the matching at T can be obtained directly by concatenating the substitutions for T_i, regardless of their particular value. □

Non-linear pattern sets do not have this property: the corresponding f_k functions need to refer to the actual values of T. This "independence" of T is reflected in the ability to encode the transition functions into a B-fsa.

3.2. Bottom-Up Matching Using Subpattern B-fsa

A subpattern LB-fsa is a non-deterministic bottom-up tree automaton together with a labeling function. The automaton encodes the functions of Proposition 3.11 by associating with each pattern in Π_F a single state and encoding the combination of the children of a pattern into the pattern.

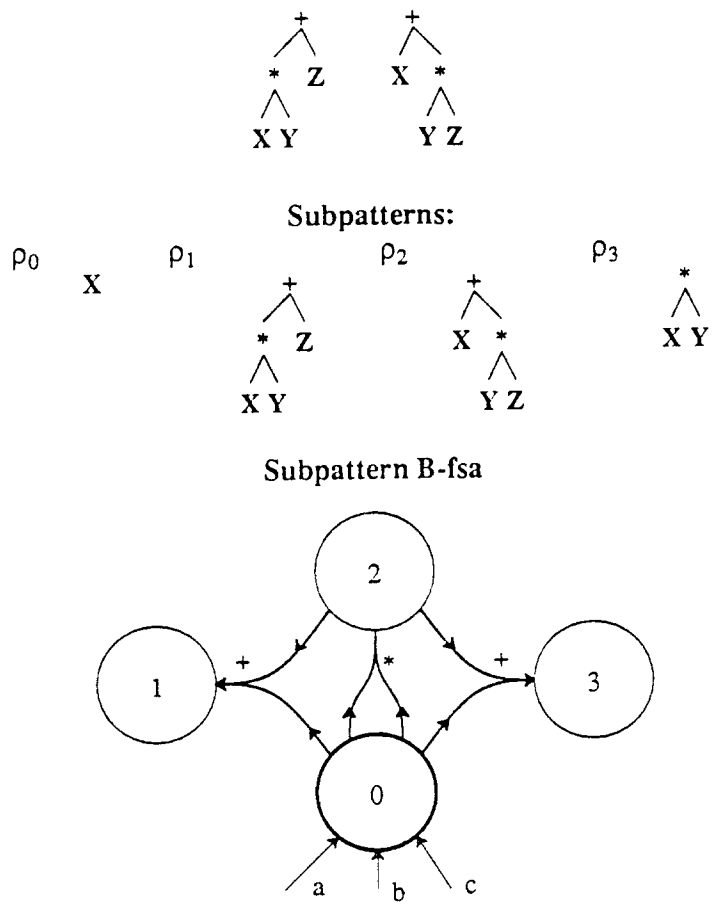
Definition 3.10 *The linear subpattern B-fsa associated with a pattern set F is a non-deterministic LB-fsa $\langle A, L \rangle$. The set of states of A is Π_F. Let St_ρ be the state associated with a pattern $\rho \in \Pi_F$.*

The transition function is defined as follows. Let $\rho_1, \ldots, \rho_n$ be patterns in Π_F. If there are equivalent patterns $\rho'_1, \ldots, \rho'_n$ such that $op(\rho'_1, \ldots, \rho'_n) = \rho \in \Pi_F$, then $Trans_n(Op, St_{\rho_1}, \ldots, St_{\rho_n}) \triangleq \{St_\rho, St_X\}$; otherwise, $Trans_n(Op, St_{\rho_1}, \ldots, St_{\rho_n}) \triangleq \{St_X\}$.

The labeling function L assigns to each state St_ρ the set $\{\rho\}$ if $\rho \in F$, and $\varnothing$ otherwise.

Since the labeling function is always fixed, the rest of this section concentrates on the B-fsa.

The automaton of Definition 3.10 is non-deterministic because it is always possible to transfer to a default state representing the subpattern X. Figure 3.2 shows a pattern set, its subpatterns, and its associated subpattern B-fsa. The B-fsa is presented without all its "default" transfers, and assumes that there are three nullary operators a, b, and c. The B-fsa is the same as in Figure 2.2.



Example of a Simple Pattern Set and its Subpattern B-fsa

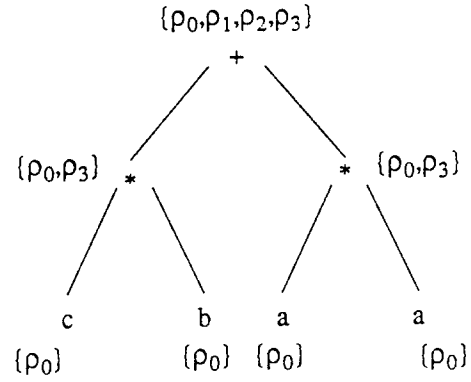
Figure 3.2

The matching algorithm simply tracks all the accessible states using the non-deterministic B-fsa. Once the match sets have been computed, the labeling function could be applied to obtain the set of matching patterns. The algorithm to compute the match sets is described in Figure 3.3. It is the standard algorithm for non-deterministic B-fsa, but it is listed here to allow some comparisons later. The states associated through B-fsa with a subject tree are shown in Figure 3.4.

```
procedure match-lSP(N: node in subject)
    let N be a node in the subject;
    let N_1 ,...., N_n be the children of N;
    let Op be the label of N;
    for each N_i
        call match-lSP(N_i)
    let σ_i be the match set associated with the subtree of the subject rooted at N_i
    set σ=∅;
    for each ρ_1 ∈ σ_1 do
        for each ρ_2 ∈ σ_2 do
            ...
            for each ρ_n ∈ σ_n do
                let ρ=Trans_n(op ,ρ_1, . . . , ρ_n)
                if (ρ≠X)∧ρ is not in σ then
                    set σ=σ∪{ρ};
    σ = σ ∪ {X};
    let σ be the match set associated with the subtree of the subject rooted at N;
```

Matching Algorithm for Subpattern B-fsa

Figure 3.3



Result of Solving Linear Pattern Matching on a Subject

Figure 3.4

The algorithm described in Figure 3.3 is quite straightforward, as is its correctness proof.

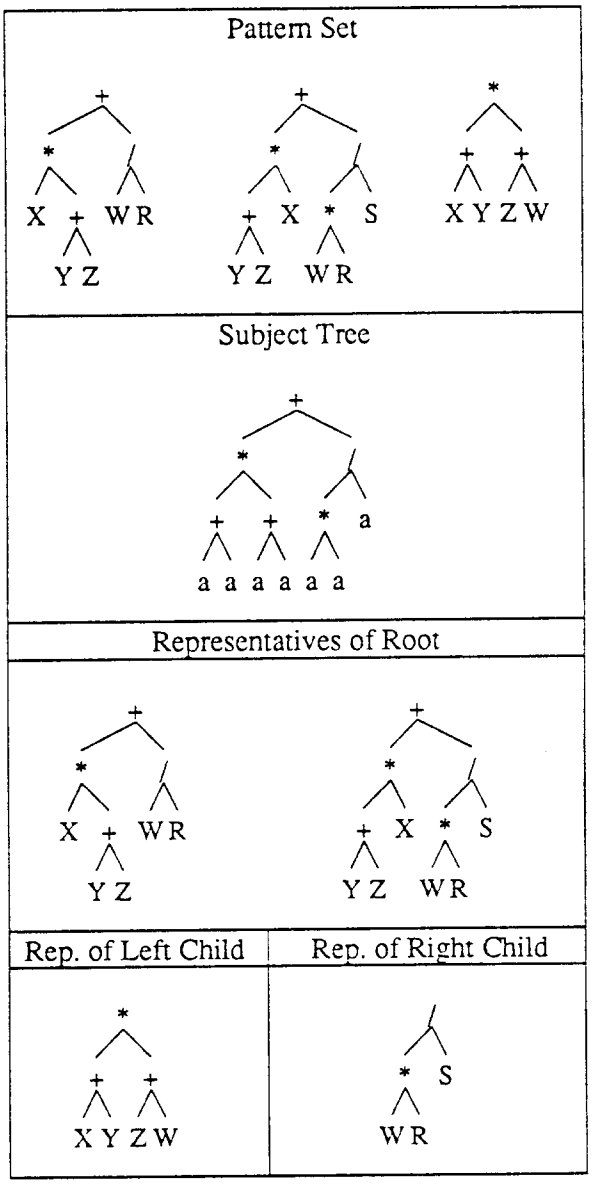
3.2.1. Representation of the Match Sets

There are two different but related representation issues in an implementation of the previous matching algorithm: how to represent the match set, and how to represent the subpattern LB-fsa.

Three possible representations for match sets are: as a simple list of all the patterns in the match set, through its set of representatives, or through a list of patterns in Π_F that subsume all the patterns in the match set, thus including all the representatives but, maybe, also some others. The first two representations are unique, but the last one is not.

The major cost of the algorithm of Figure 3.3 is in the inner loop, which, in turn, seems to depend on how many possible combinations of patterns in Π_F, one from the match set of each child, have to be considered. The first representation mentioned above is the one assumed up to this point. If it is used, it leads to the algorithm described in Figure 3.3. The algorithm is simple, but its main disadvantage is that, if the match sets are of substantial size, the algorithm is quite slow since the number of possibilities to consider may be considerable.

Representing a match set through its set of representatives leads to substantially smaller representations. Hence, one would expect a faster implementation. Unfortunately, there is no guarantee that considering the combination of representatives of match sets will lead to the desired representatives of the new match set. For instance, consider the example of Figure 3.5. The top row shows a pattern set composed of three patterns. The second row shows a tree whose match set is the set containing the first two patterns of the pattern set together with $+(X,Y)$ and X. The third row shows the set of representatives of the root of the pattern in the second row, while the fourth row show the sets of representatives corresponding to the immediate subtree of that pattern.



Example of Computing a Match Set Using the Set of Representatives

Figure 3.5

Figure 3.5 shows that finding the set of representatives for a tree given the set of representatives of its children requires additional computation: the representative of a subtree at a node is not the combination of the representatives of its children. The additional computation relates to the relationship implicit in a set of representatives: the subsumes relationship. This topic is described in more detail in Section 3.4.

A final type of representation is a "mixed" representation, in which a set of patterns in Π_F that includes the set of representatives, but maybe also other patterns in the match set, is used. This approach can be used in some applications to alleviate the difficulties of the representation by the set of representatives while retaining some of their advantages. Its main disadvantage is that it is not a unique representation and comparing two match sets for equality has non-trivial cost.

3.2.2. Representation of the Subpattern LB-fsa

A very interesting issue is how to represent the B-fsa. The choice of data structures to represent the B-fsa has a big influence on the performance of the table construction algorithms and the pattern matching algorithms. A first observation is that, since the default state (the match set $\{X\}$) is always in the new set of states, it is not necessary to encode it in the representation. Different representations allow different implementations of the internal loop in Figure 3.3. Two possible representations are "direct", and "inverse".

The most straightforward alternative is to encode the transition "directly". This can be done using one n dimensional table for each n-ary operator. This representation leads directly to the algorithm of Figure 3.3. The use of the table has two disadvantages. First, the table will contain a large percentage of empty entries since for most combinations of subpatterns and operators the new state set contains only the default state. Second, the representation does not reduce the large number of combinations that need to be considered.

The first problem could be corrected using, for example, some hash encoding, but the second problem is intrinsic to the "direct" approach. An alternative is to use an "inverse" representation. The idea with this representation is the opposite of the previous one: instead of starting with the members of the match set, combining them and then asking whether a combination is valid, the valid combinations under a given operator are found first, and then the subpatterns are checked for their presence in the match sets of the children. Which approach is best depends on the probabilistic behavior of the match sets.

A convenient representation of the inverse approach is one that keeps, for a given operator, a list of all the subpatterns that appear as a first child in a subpattern rooted by that operator. Then, for each one of these, it keeps a list of all the subpatterns that appear as a second child in a subpattern rooted by that operator and with first child the one chosen, and so on up to the arity of the operator. This representation makes it possible to "prune" unwanted combinations. This comes at the cost in space used in the representation. In an inverse representation the inner loop of Figure 3.3 can be implemented as follows:

```
set σ=∅;
let σ_1 ,..., σ_n be the match sets of the children;
for each ρ_1 that is a 1st child of a pattern rooted by op do
    if ρ_1 is in σ_1 do
        for each ρ_2 that is a 2st child of a pattern ρ rooted by op
          and with ρ_1 the 1st child of ρ do
            if ρ_2 is in σ_2 do
                ...
                for each ρ_n that is an n-th child of a pattern ρ rooted by op
                  and with ρ_1 ,..., ρ_{n-1} the 1st, 2nd ,..., n-1th children of ρ do
                    if ρ_n is in σ_n do
                        let ρ = Trans_n(op ,ρ_1, . . . , ρ_n)
                        if (ρ≠X)∧ρ is not in σ then
                            set σ=σ∪{ρ};
```

The inverse representation is used in several places in this dissertation, including the implementation described in Chapter 8.

Another possible representation organization is to combine both the direct and the inverse representations. Such a representation is suggested in [PuB87] where it is called a "curried dag". A curried dag uses a 3-argument hash function. The hash function returns a value that represents either a valid partial combination of subpatterns for the children, or an indication of a non-valid combination. The first argument to the hash function is the operator, and the second argument is the subpattern in consideration in the match set of the child. The third argument is *nil* to obtain the value for the first child, and is the value obtained from the $n-1$st invocation afterwards. The main advantages of the curried dag representation are its flexibility, allowing addition of patterns, and the way it generalizes to non-linear patterns (see Chapter 4). This makes it suitable for problems like the Knuth-Bendix completion algorithm [KnB70]. The curried dag representation may be slower, or faster, than both the direct and the inverse representations.

3.3. Bottom-Up Matching Using Match Set LB-fsa

The main difficulty in using the algorithm of the previous section in some application areas is that it may be quite slow to track all the reachable states of the (non-deterministic) LB-fsa to find the match sets. The alternative is to convert the non-deterministic LB-fsa into a deterministic one, and to track only one state, directly corresponding to the match set.

Definition 3.11 *The linear match set LB-fsa associated with a pattern set F is a complete deterministic LB-fsa A together with a labeling function L. The set of states of A is the set of match sets of F.*

The transition function is defined as follows. Let St_σ *be the state associated with the match set* σ. *Let Op be an n-ary operator, and let* $\sigma_1,\ldots,\sigma_n$ *be any n match sets, then* $Trans_n(St_{\sigma_1},\ldots,St_{\sigma_n}) \triangleq \{St_\sigma\}$, *where* σ *is the match set at the term* $Op(\rho_{\sigma_1},\ldots,\rho_{\sigma_n})$.

The labeling function L assigns σ_i *to each state* St_{σ_i}.

Since the labeling function is fixed, frequently a match set LB-fsa is identified with its B-fsa. The match set LB-fsa does what its name suggests:

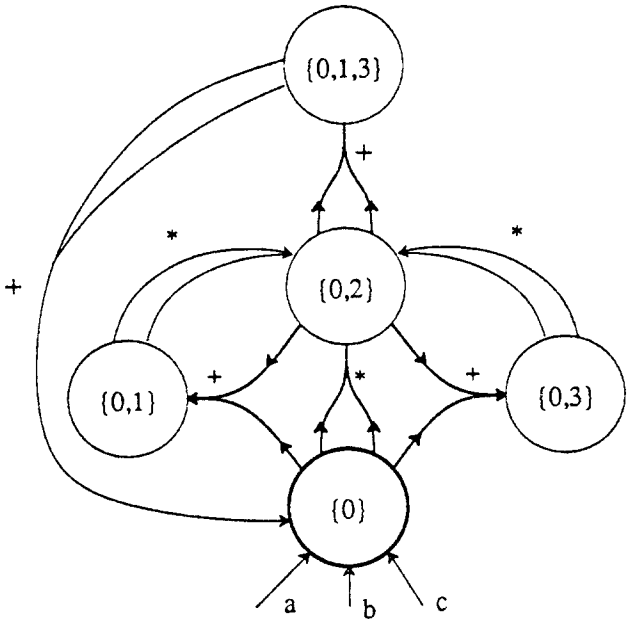
Proposition 3.12 *Let F be a pattern set and let A be the B-fsa of its match set LB-fsa. Then, for every tree T, the state at T under A is* St_σ *if and only if the label of the state associated with T is* σ.

Proof Structural induction on the trees, using the definition of match set B-fsa and Proposition 3.10 □

From Proposition 3.12 we get:

Corollary 3.2 *Let F be a pattern set, let A be its subpattern LB-fsa, and let A′ be its match set LB-fsa. Then, A′ is the result of converting a non-deterministic LB-fsa into a deterministic LB-fsa (Section 2.4.1)*

It is difficult to present examples of match set B-fsa because of the large number of edges present. Figure 3.6 shows part of the match set B-fsa corresponding to the pattern set of Figure 3.2. In this case, the match set B-fsa has one more state than the subpattern B-fsa, and many more non-default transitions. The new state is the one at the top.



Match Set B-fsa

Figure 3.6

Note that the B-fsa is identical to that of Figure 2.4, confirming that the match set B-fsa is the deterministic version of the subpattern B-fsa. The new matching algorithm just computes the state associated with a tree for a deterministic B-fsa.

```
procedure match-lMS(N:node)
    let N be a node in the subject;
    let N_1 ,..., N_n be the children of N;
    let Op be the label of N;
    for each N_i
        call match-lMS(N_i)
    let σ_i be the match set associated with the subtree rooted at N_i
    σ = Trans_n (Op ,St_ρ1,.....,St_ρn);
    σ is the match set associated with the subtree rooted at N;
```

Matching of Linear Match Sets

Figure 3.7

Figure 3.7 gives an algorithm for linear pattern matching based in the match set LB-fsa. The correctness of the procedure follows from the definition of match set LB-fsa.

Computing the Match Set LB-fsa

The basic approach to computing the match set B-fsa is to first compute the subpattern B-fsa and then use it to compute all the match sets in an algorithm essentially identical to the construction of a deterministic automaton from a non-deterministic one (Figure 2.3). Everything mentioned in the previous section on computing the match set applies here. The only difference is that it must be possible to find out efficiently if two match sets are identical or not. This probably rules out any type of "mixed" representation for the match sets at match set B-fsa construction time.

Representation of the match set LB-fsa

Given the (proportionally) large number of "non-default" transitions, some type of table representation seems the appropriate choice. These tables are not an efficient encoding because a match set B-fsa is not a generic DB-fsa. A very successful encoding technique is to use, instead of one i-dimensional table for each arity i, as many tables as operators in *Op*. Each one of these tables can then be encoded independently. Section 3.5 discusses the work of David Chase regarding how to compute efficiently, for each one of these tables, sub-tables that are identical and can be shared.

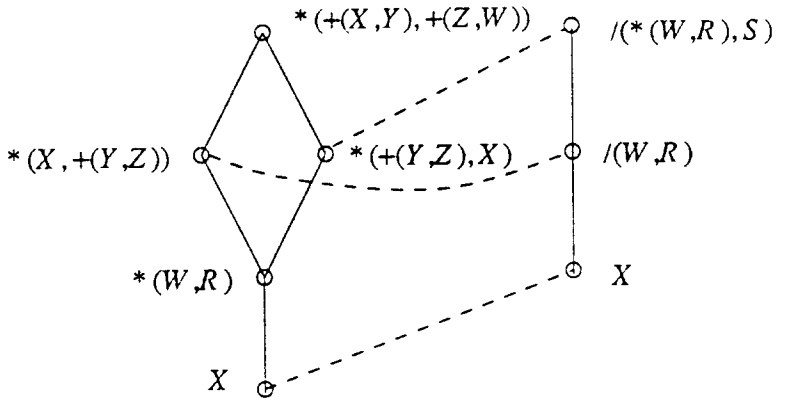
A drawback to the use of match set B-fsa instead of subpattern B-fsa is the potential increase in the size of the tables. Since the number of states in the match set B-fsa is identical to the number of possible match sets of the pattern set, the CTF property is sufficient, but not necessary, to guarantee a small number of states. Experience collected by Chase [Cha87], and the author (Chapter 8), indicates that, in practice, table size is not a problem.

3.4. The Subsumption Relationship

The key operation in the two algorithms presented in the two previous sections is, given an n-ary operator *op* and match sets $\sigma_1, \ldots, \sigma_n$ corresponding to trees $T_1, \ldots, T_n$, to obtain the match set σ of $op(T_1, \ldots, T_n)$. This operation is done at match time when using the subpattern LB-fsa and at table construction time when using the match set LB-fsa, and involves combining

subpatterns from the children under the operator of the parent. In this section we show how to use Proposition 3.10 together with the immediate subsumption graph (Def. 3.8) to perform this computation.

Recall the example of Figure 3.5. The left part of Figure 3.8 is the immediate subsumption graph for $*(+(a,b),+(c,d))$, the right part is the immediate subsumption graph for $/(*(e,f),g)$, and the broken lines in a thicker pen represent those pairs of patterns $<\rho_1,\rho_2>$ such that $+(\rho'_1,\rho'_2)$ is a subpattern, where ρ'_1 and ρ'_2 are patterns equivalent to ρ_1 and ρ_2 respectively except that no variable in ρ'_1 appears in ρ'_2.



$G_{>i}$ and Finding Match Sets

Figure 3.8

Let $T = op(T_1,T_2)$, and let σ_1 and σ_2 be the match sets of T_1 and T_2. We want to compute the match set of T, σ_T. The direct approach would be to consider all possible trees of the form $op(\rho_1,\rho_2)$ with $\rho_1 \in \sigma_1$ and $\rho_2 \in \sigma_2$ and to see if each tree is equivalent to a pattern $\rho \in \Pi_F$. If it is, then ρ is added to σ_T; if it is not, ρ is not added. This direct approach can be described as traversing the subsumption graphs for σ_1 and σ_2, considering all possible pairs of visited nodes, and testing if the pair of nodes in the two graphs is joined by a broken line. In Figure 3.8, this approach will test $15 = 5*3$ combinations to find the 3 subpatterns that correspond to the combinations marked by the thick broken lines[13].

Proposition 3.10 allows us to impose some order on the traversal of the two graphs so that some pairs of nodes are not visited. In Figure 3.8, since $+(*(+(Y,Z),X),/(*(W,R),S))$ is a subpattern, we know that any term subsumed by it that is a subpattern will be in the match set σ_T. And similarly for $+(*(X,+(Y,Z)),/(W,R))$. Thus, the traversal can be done recursively top-down starting at the root, using part (4) of Proposition 3.10 to perform the decomposition (that is, when finding the set $M(op(n_1,n_2))$ associated with a pair $<n_1,n_2>$, we have to consider all the nodes

[13] That is, the match set of $+(*(+(a,b),+(c,d)),/(*(e,f),g))$ consists of the patterns $+(*(X,+(Y,Z)),/(W,R))$, $+(*(+(Y,Z),X),/(*(W,R),S))$, and $+(X,Y)$ (not $+(X,X)$, since the two Xs are unrelated).

that are directly connected to n_1 and to n_2 in the left and right dag respectively), part (2) to "prune" down traversals (that is, if a pair $<n_1,n_2>$ is found to be connected. all the pairs $<n'_1,n'_2>$ where both n'_1 is below n_1 and n'_2 is below n_2 need not be tested), and part (3) to end the recursion at the leaves (if a pair of leaves are not connected, then the M set is empty).

The "traversal" approach described above can be used in different ways depending on the representation used for match sets, and on how efficiently it can be detected that a pair need not be visited. If match sets are represented explicitly by a list of their subpatterns, then we could keep a table containing, for each subpattern the list of subpatterns subsumed by it. Then, whenever a subpattern is found in the top-down traversal, all the subpatterns subsumed by it could be added to σ_T. This case has the additional advantage that the correctness of the approach does not depend on detecting all the nodes that don't have to be visited: visiting a node after it has already been included in σ_T is inefficient but still correct.

The situation is more complicated (in the general case) if a match set is represented by its set of representatives. The idea here would be just to add to σ_T the representatives as they are found in the top-down traversal. The problem is that in this case the correctness of the representation depends on not considering a node unless it is not subsumed by any other pattern in the final match set σ_T. In the example mentioned above, we must avoid reaching $+(X,Y)$ or otherwise it will be added incorrectly to the set of representatives of σ_T. Determining the situation correctly in all cases seems expensive, especially in those cases where the cost of testing whether a pair is connected or not is low.

Thus, it seems that the benefits of using the $G>_i$ depend on the pattern set and the particular application. In an application where it is statistically likely that the representatives of the two match sets will succeed in combining, using the $G>_i$ will be more efficient than a exhaustive enumeration of all the possibilities; in other cases that may not be the true.

Yet another alternative is to collect a set of patterns that will include all the representatives and, hopefully, only a few more, and then to remove them. This approach will pay off only when there is a very small number of candidate representatives, since discarding the duplicates is a quite expensive operation.

Simple and CTF pattern sets

The problems mentioned above can be solved efficiently if the pattern set is CTF. In that case part (4) of Proposition 3.10 can be strengthened as follows:

Proposition 3.13 *Let F be a pattern set over Op, let op be an operator in Op, let $T = op(T_1,\ldots,T_n)$ be a tree, let $\sigma_1,\ldots,\sigma_n$ be the match sets of $T_1,\ldots,T_n$ respectively, and let $\rho_1,\ldots,\rho_n$ be patterns equivalent to the meets of $\sigma_1,\ldots,\sigma_n$ respectively, where the variables are renamed, if necessary, so that $\{\rho_1,\ldots,\rho_n\}$ is variable-disjoint. Let $I_{\rho_i} = \{\rho_i{}^1,\ldots,\rho_i{}^{n_i}\}$ denote the immediate subsumption set of each ρ_i. If $op(\rho_{\sigma_1},\ldots,\rho_{\sigma_n})$ is not equivalent to a pattern in Π_F, then $M(op(\rho_{\sigma_1},\ldots,\rho_{\sigma_n})) \equiv \bigcup_{i=1}^{n}\bigcup_{j=1}^{n_i} M(op(\rho_1,\ldots,\rho_{i-1},\rho_i{}^j,\rho_{i+1},\ldots,\rho_n))$, and, there are i',j', $1\le i'\le n$, $1\le j'\le n_{i'}$ such that $\forall i\ 1\le i\le n\ \ \forall j\ 1\le j\le n_i\ \ (M(op(\rho_1,\ldots,\rho_{i-1},\rho_i{}^j,\rho_{i+1},\ldots,\rho_n))) \subset M(op(\rho_1,\ldots,\rho_{i'-1},\rho_{i'}{}^{j'},\rho_{i'+1},\ldots,\rho_n)))$.*

Before proving the proposition we need an auxiliary Lemma:

Lemma 3.1 *Let F be a (linear or non-linear) pattern set. Let ρ, ρ_1, ρ_2, ρ_1', and ρ_2' be patterns in F. Then, if both $\rho_1 \oplus \rho_2$ and $\rho_1' \oplus \rho_2'$ exist, and $\rho_1 \ge \rho_1'$ and $\rho_2 \ge \rho_2'$, then $\rho_1 \oplus \rho_2 \ge \rho_1' \oplus \rho_2'$.*

Proof By definition of $\oplus$ and the conditions of the lemma, $\rho_1 \oplus \rho_2 \ge \rho_1 \ge \rho_1'$ and

$\rho_1 \oplus \rho_2 \geq \rho_2 \geq \rho_2'$. The desired result follows by a new application of the definition of $\oplus$. □

Proof of Proposition 3.13 Once again, we provide the proof only for the case of $n = 2$. Let $M(op(\rho_{\sigma_1}, \rho_{\sigma_2}))$ be $\phi_1, \ldots, \phi_m$. Let ϕ be the meet of $\phi_1, \ldots, \phi_m$. Since F is CTF, $\phi \in \Pi_F$, and, by Lemma 3.1, $op(\rho_1, \rho_2) \geq \phi$. By definition of $>_i$, there must be some pattern ν of the form $op(\rho_1, \rho_2^i)$ or $op(\rho_1^i, \rho_2)$ that subsumes ψ. $M(\nu)$ is the desired one. □

Proposition 3.13 says that, if the subsumption graph of $op(\rho_1, \ldots, \rho_n)$ is explored in a top-down, breadth-first manner, the first successful node found in Π_F is the representative of the match set. The traversal can also be obtained through the simultaneous and coordinated traversal of the subsumption graphs of $\sigma_1, \ldots, \sigma_n$, as suggested in the previous section, never testing for the presence of a pair of nodes (denoting a pattern) in Π_F until all the pairs with both nodes above the current pair have been tested.

The breadth-first traversal can be enforced cheaply. If the combination is done by traversing through the subpatterns in each of the σ_i and then testing their combination, the subpatterns can be renumbered according to a topological sort under $>_i$. If the combination is done by using the children of the combining operator (as was suggested at the end of Section 3.3), the topological sort has to be applied to the order in which the children are kept.

For concreteness, assume an inverse representation of the subpattern B-fsa. Then the algorithm to combine the match sets could look like:

```
let σ1 ,..., σn be the match sets of the children,;
for each ρ1 that is a 1st child of a pattern rooted by op
 and in a topological order by >i  do
   if ρ1 is in σ1  do
      for each ρ2 that is a 2st child of a pattern ρ rooted by op
        with ρ1 the 1st child of ρ
        and in a topological order by >i  do
          if ρ2 is in σ2  do
              ...
             for each ρn that is an n-th child of a pattern ρ rooted by op
               and with ρ1, ..., ρn-1 the 1st, 2nd ,..., n-1-th children of ρ
               and in a topological order by >i  do
                  if ρn is in σn  do
                     let ρ1', ..., ρn' be patterns equivalent to ρ1, ..., ρn respectively
                      where {ρ1', ..., ρn'} is variable-disjoint
                     if there exists a pattern ρ ≡ op(ρ1', ..., ρn') in F then
                        return the match set induced by ρ;
```

There are other possible traversal orders that satisfy the "breadth-first" requirement.

If the pattern set is not only CTF but also simple, then the dags are actually just chains and the breadth-first traversal becomes just a linear search.

CTF subpattern B-fsa

The above technique is quite fast but, as stated, it only works for CTF pattern sets. The idea to modify it to deal with non-CTF pattern sets is quite simple. Given any pattern set F, obtain the pattern set that contains the meets of all the subpatterns of F (that is, the meets of all the match sets in F). Then construct the subpattern B-fsa for this new pattern set, which is clearly CTF. The new tree automaton suggested by this approach is called the CTF subpattern B-fsa.

The result is a pattern matching algorithm that will be slower than the one using a match set B-fsa, but with smaller table requirements. The comparison of the two table sizes is quite straightforward. The match set B-fsa has the same number of states as the CTF subpattern B-fsa

but contains many more transfer edges that need to be represented. Also, computing the tables for the CTF subpattern B-fsa is substantially faster.

Ken Rimey's Contribution

Ken Rimey [Rim85] independently has proposed yet another way of using the subsumes relation. In his paper, Rimey restricts his attention to binary trees. Using the terminology presented in this dissertation, his proposal corresponds to using a match set B-fsa where "half" of the transfer edges are missing. For each match set σ, and each operator op, the modified B-fsa stores directly the entry corresponding to $op(\sigma,\sigma')$ only for a few states σ'. The values for the other entries are obtained by decomposing them using $G >_i$.

Rimey's proposal is more expensive in space than the CTF subpattern B-fsa, but faster at matching time. On the other hand, the size of the tables is of the same order as a match set B-fsa, which, for the case of two arguments, is quadratic in the number of match sets. Of course, as in the CTF subpattern B-fsa, this last one can be exponential in the size of the pattern set.

3.5. David Chase's Contribution

David Chase has done independent research in linear N-pattern matching. This section describes his work briefly; see [Cha87] for more details.

Chase's pattern matcher is essentially a match set B-fsa represented so that for each n-ary operator there is an n-dimensional table. In addition, identical n-1 dimensional sub-tables are found and shared. In the frequent case of a binary operator, this means finding identical rows and columns and sharing them. For $n=2$, the process can be visualized as one of "folding" the array. The folding function, also called a **restrictor**, is a mapping associated with an n-ary operator and a i $(1 \le i \le n)$ child position that maps from the set of match sets to an $n-1$-dimensional subtable. The importance of Chase's contribution lies in that the folding is found in the same closure operation that produces the collection of match sets. For large pattern sets, his technique is substantially faster than an implementation that would compute the automaton first and then find the folding. This speedup is particularly significant when there is a large B-fsa where the unfolded representation may exceed virtual memory constraints.

Recall that σ_t denotes the match set associated with t. If $op \in Op$ has arity n, and $1 \le i \le n$, let $P_{op,i}$ denote the set of all patterns in Π_F that are equivalent to the i-th child of a subpattern in Π_F rooted by op. If σ is a match set, let $R_{\sigma,op,i}$ denote $\sigma \cap P_{op,i}$. If $T=op(T_1,\ldots,T_n)$, then part (1) of Proposition 3.10 states that σ_T is the set consisting of X and all those patterns in Π_F equivalent to $op(t_1,\ldots,t_n)$ where $t_i \in \sigma_{T_i}$. It follows that

Proposition 3.14 *If $op \in Op$ with arity n, and $T_1,\ldots,T_n$ are trees over Op, then $\sigma_{op(T_1,\ldots,T_n)}$ is the set containing X and all the patterns in Π_F equivalent to $op(t_1,\ldots,t_n)$ with $t_i \in R_{\sigma_{T_i},op,i}$.*

The advantage of this observation is that it provides a "natural" restrictor: for each n-ary operator op, and each integer i, $1 \le i \le n$, the restrictor maps a match set σ into $R_{\sigma,op,i}$. Chase shows [Cha87] that this folding is optimal: that is, the size of the folded array is as small as possible. Chase's algorithm is described in Figure 3.9, for the case where all operators are either nullary or binary.

```
procedure Chases-Compute-B-fsa
    Construct a curried representation of the subpattern B-fsa;
    for each op ∈ Op do
        let n be the arity of op
        for each i, 1≤i≤n do
            compute the set P_{op,i}.
    set last_iteration=0
    for each op of arity 0 do
        compute its match set σ;
        set index=add(σ);
        set B-fsa[op].0ary=index; /* index represents the match set σ */
    while there is some R_{σ,op,i} with mark[R_{σ,op,i}]=last_iteration do
        for each op ∈ Op do
            let n be its arity
            for any selection of R_{σ,op,i}, 1≤i≤n
              where at least one of them has mark[R_{σ,op,i}]=last_iteration, do
                set σ'=∅
                for each t_i selected from R_{σ,op,i} do
                    let {t_1', ..., t_n'} be a variable-disjoint set
                      equivalent to {t_1, ..., t_n};
                    consult the subpattern B-fsa to determine if op(t_1', ..., t_n')
                    is equivalent to a pattern in Π_F, and if so, add it to σ'
                comment now σ' is the match set corresponding to the collection R_{σ,op,i}
                set index=add(σ');
                set B-fsa[op].Nary.transfer[R_{σ,op,1}, ..., R_{σ,op,n}]=index;
        set last_iteration = last_iteration + 1;
    for each op ∈ Op with arity >1 do
        for each i 1≤i≤n do
            set B-fsa[op].Nary.restrictor[i]=make-map(R_{σ,op,i});

procedure add(σ) returns index into SetOfAllMatchSets
    let σ be a match set
    let index be the index of σ in SetOfAllMatchSets;
    if σ is new then
        for each op ∈ OP do
            let n be the arity of op
            for each i, 1≤i≤n do
                compute R_{σ,op,i};
                if R_{σ,op,i} had not been generated before then
                    set mark[R_{σ,op,i}]=last_iteration;
    return index;
```

Chase's Algorithm

Figure 3.9

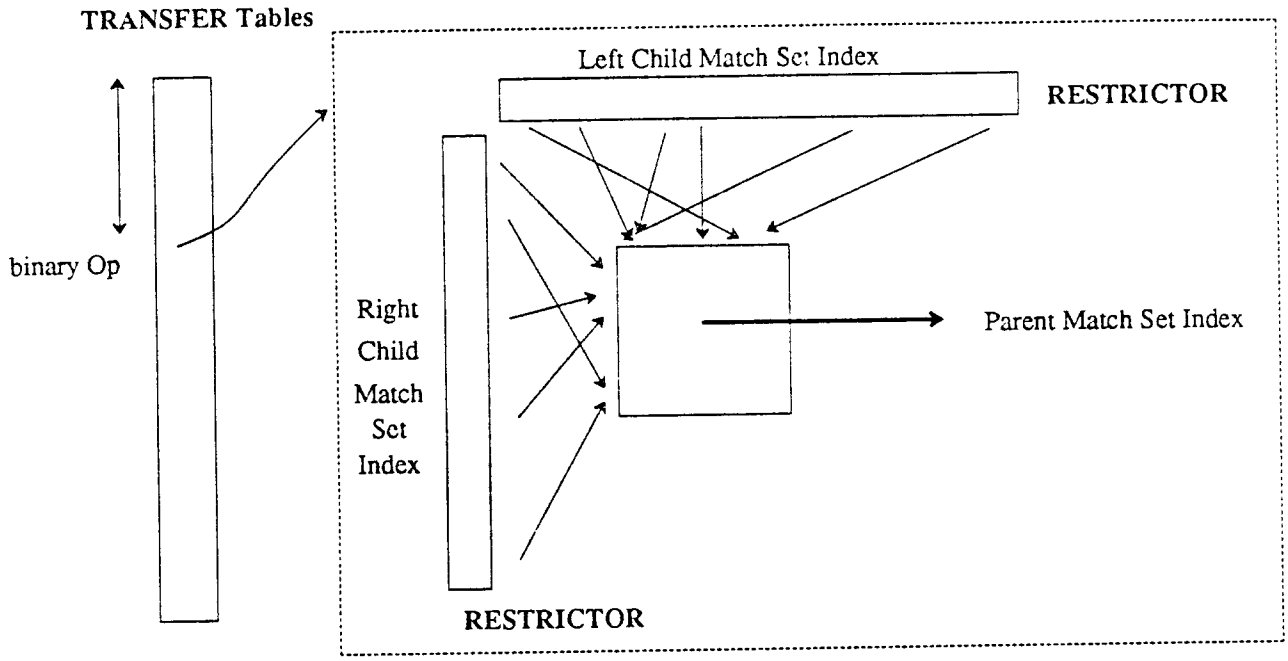
The result of the algorithm is the match-set B-fsa in folded representation; the information associated with each operator *op* is denoted by $B\text{-}fsa[op]$; if *op* is an 0-ary operator, the information is simply its corresponding match set ($B\text{-}fsa[op].0ary$); if *op* is an $n>0$-ary operator, the information is a structure ($B\text{-}fsa[op].Nary$) with two fields, *transfer*, which contains the folded transfer arrays for *op*, and, for each i with $1 \le i \le n$, *restrictor* which is computed from the sets $R_{\sigma,op,i}$ by applying the (here undefined) function *make-map*. The B-fsa obtained by the algorithm is organized as shown in Figure 3.10, which only shows the situation for binary operators: associated with each binary operator there are two restrictor arrays which map indices representing match sets into indices representing the $R_{\sigma,op,i}$, which are then used to fetch the index of the match set of the node rooted with the binary operator from the folded transfer table.

The algorithm of Figure 3.9 computes all the match sets σ and $R_{\sigma,op,i}$ through a closure technique. The σ sets are represented by indices into a set named "SetOfAllMatchSets", and are "marked" to indicate that all they have already been considered in the closure algorithm. Since the new states are always computed by first finding the set $R_{\sigma,op,i}$, there is no need to compute or store the complete match set B-fsa, only the much smaller transition tables indexed by $R_{\sigma,op,i}$.

Another factor in the success of Chase's algorithm is that many of the structures involved can be represented using bit vectors:

- Match Set. The match set is represented as a bit vector over all the patterns in Π_F.
- $P_{op,i}$. The patterns that can appear per position and operator are also represented as a bit vector over the same base set. This means that the most frequent operation of the algorithm, computing the sets $R_{\sigma,op,i}$ can be done very efficiently.
- $R_{\sigma,op,i}$. All the (distinct) intersections of a match set and a $P_{op,i}$ are stored in a set of bit vectors.

The final shape of the generated tables is shown in Figure 3.10. Final table sizes are, for most applications, very small. The interested reader should consult [Cha87] for some comparisons in table generation time and table generated size.



Folding Rows and Columns in a Match Set B-fsa

Figure 3.10

Improvements to the Technique

[Cha87] misses a simple but very useful consideration for reducing table size. In most applications after finding the folding, most of the table space is spent describing the folding, in the restrictor arrays. It is also common that patterns have the same subpatterns. When these two conditions occur together, for instance in the applications of Chapter 8, it is possible to obtain big savings in table size by sharing identical restrictor arrays. Finding the identical restrictors is quite simple: it can be done by just checking for identical $P_{op,i}$.

It is also tempting to try to use ideas described in Section 3.4 to speed up the inner loop of the table construction algorithm. In practice it seems that there would be implementation problems to make it run faster than the current "direct" approach.

3.6. The Influence of an Input Set

Up to this point this chapter has assumed that PATTERN MATCHING is solved relative to the input set L_{Op}. That is, we want to find the patterns matching for all possible trees over the set of operators Op. In some cases we have additional knowledge about the possible input trees that can be used profitably. In particular, we may know that the set of input trees is a recognizable set L (Definition 2.5). This additional information can be employed using the following general technique:

Proposition 3.15 *Let A and B be two B-fsa over some operator set Op. There is an algorithm that will compute, for each state St_A in A, the set of all those states St_B in B such that there is a tree T labeled with state St_B in B that would have been labeled with state St_A in A.*
Proof The set of states associated with each node in A is finite and can be computed through iteration on the nodes of A. □

The question of interest is the following:

Proposition 3.16 *Let L be a set of trees over Op, and let F be a set of patterns over Op. There exists an algorithm that will determine which match sets of F correspond to subtrees of trees in L.*
Proof Let A_F be the B-fsa that computes the match sets for F, and let A_L be the B-fsa that recognizes membership in L. By Proposition 3.15 there is an algorithm that will associate sets of states from A_F with the states of A_L. A "good" state in A_F is one that corresponds to a match set of F containing at least one subpattern in Π_F present in F. There is a tree in L with a subtree matching some pattern in F if and only if there is a state in A_L that reaches (in the B-fsa) a final state and that has been labeled with a set containing at least one good state in A_F. □

This provides directly a solution to:

Corollary 3.3 *Let L be a set of trees over Op, and let F be a set of patterns over Op. There exists an algorithm that will determine whether there exists a tree $T \in L$ such that some pattern in F will match at some node of T.*

The algorithm of Proposition 3.16 can be used to reduce the table size requirements for solving pattern matching, while that of Corollary 3.3 can be used to detect specification inconsistencies.

3.7. Related Work

There are two main approaches to pattern matching: those based on a "top-down" traversal of the subject tree, and those based on a "bottom-up" traversal of the subject tree. This chapter has used only a bottom-up approach.

Top-Down Pattern Matching

The most straightforward algorithms for pattern matching are based on top-down traversals. Unfortunately they are quite inefficient. The main advantage of a bottom-up traversal over a top-down traversal is that it aggregates the information in a natural way from children to their parents.

The simplest method for matching a set of patterns against a single subject is to decompose the problem into several independent problems of matching a single pattern against a single subject. Likewise, the simplest approach to matching a single subject against a single pattern is to reduce the problem to the case where the pattern is forced to match at the root of the subject, and to repeat the problem for all the nodes in the tree.

The approach outlined above is very easy to implement but, except in the most simple applications, is impractical. A better approach is presented in [KMR72]. The technique employed maps the problem of tree pattern matching into string pattern matching by using as patterns the strings obtained when doing a preorder traversal of the tree patterns. Since this mapping is not unambiguous, whenever any such string is found, its origin in the subject tree is marked. Any node with "enough" marks is matched. The string matching problem is solved using an automata that keeps track of all the different strings at once (using its finite state memory).

This algorithm works fairly well when there is only one pattern to consider. When there are several patterns, it is necessary to keep independent markers for each tree pattern. There are

several ways to do this. The simplest one is to keep different locations for each counter, increment the locations, and check the values for all of them at the second visit to the node in the preorder traversal. Depending on the density of "hits" (of which currently the author has no experimental data) it may be better to test after each increment. Yet another alternative would be to use a heap of increment requests. In general, it appears that top-down pattern matching will be slow if there are many patterns.

[HoO82] and [KMR72] elaborate on these ideas. The biggest advantage of top-down pattern matching is the reduced size of the tables created. The price for this reduction in table size is an increase in the pattern matching time. This is particularly true when the bottom-up algorithm uses an implicit representation of the match set. Which technique to use will depend on the particular application. See Chapter 8 for some performance numbers for the REACHABILITY problem where pattern matching is employed.

Bottom-Up Pattern Matching

[HoO82] is probably the best known reference to bottom-up algorithms for linear N-pattern matching. The bottom-up algorithm for general pattern sets presented there is based on match sets, represented in a table form. Most of the emphasis of Hoffman and O'Donnell [HoO82] is on simple pattern sets (Definition 3.7), disregarding other weaker constraints.

[Kro75] is an excellent, but little known, dissertation. The emphasis of the author is on CTF pattern sets. The trees that the author deals with are slightly different from those defined in this dissertation: they are defined as labelings on tree domains, and their operators do not have a fixed arity. This probably influenced Kron's recognizer, which is based on "orthogonal tree automata". The dissertation contains equivalent material to that presented in the bottom-up part of [HoO82], and also additional material on rewrite systems. It lacks an analysis of the space complexity of the generated automata, but orthogonal tree automata seem good for sharing similar entries, at the expense of a small reduction in matching speed. It is important to note that the notion of a CTF pattern set is far more important than the notion of a simple pattern set introduced by Hoffman and O'Donnell (Proposition 3.8).

3.8. Summary of the Algorithms in this Chapter

The research presented in this chapter contains several contributions to the theory of bottom-up linear N-pattern matching. The proof that simple pattern sets are CTF provides a better understanding of the results of both [HoO82] and [Kro75]. The major contribution is the understanding of the role of $G >_t$. Proposition 3.10 is used in Rimey's algorithm and in the CTF subpattern B-fsa algorithm (Section 3.4).

The chapter also presents a unified approach, based in the notion of B-fsa, for five bottom-up algorithms. The flexibility of the model shows its applicability.

The CTF subpattern B-fsa algorithm is interesting in its own right, especially in application areas with a large number of patterns. With the advent of D. Chase's algorithm, many applications can use a shared table representation of a match set B-fsa with reasonable table sizes, but some applications may need a method that uses a representation guaranteed to produce tables with table size linear in the number of states.

Figure 3.11 shows the five algorithms mentioned here, together with an indication of their situation in a table-size × matching-speed space. The actual values for size and speed are strongly dependent on the properties of the pattern sets. Thus, deciding among the algorithms requires an analysis of the intended application.

Algorithm	Table Size	Matching Speed
Subpattern B-fsa	Edges are linear in # states States are linear in description size	Depends on size of match set
CTF Subpattern B-fsa	Edges are linear in # states States may be exponential in descr. size	Depends on depth of $G >_i$
Rimey's	Edges are quadratic in # states States may be exponential in descr. size	Depends on depth of $G >_i$
Chase's	Edges are quadratic in # states (but there is folding) States may be exponential in descr. size	~3 table lookups per tree node
Match set B-fsa	Edges are quadratic in # states States may be exponential in descr. size	1 table lookup per tree node

Overview of Algorithms for Linear Pattern Matching

Figure 3.11

Chapter 4 explores how to perform non-linear pattern matching using variations on these ideas.

CHAPTER 4

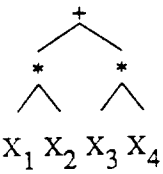
Matching Non-Linear N-Patterns

Divide et impera

[cited by Machiavelli]

In this chapter we study the problem of pattern matching for non-linear pattern sets. The approach taken is to regard a non-linear pattern as a linear pattern extended with a predicate which will evaluate to true if the non-linear conditions are satisfied. For example, the non-linear pattern

corresponds to the linear pattern



together with a predicate that tests whether the trees matched at positions $1 \cdot 1$ and $2 \cdot 1$, and those at $1 \cdot 2$ and $2 \cdot 2$ are equal. Such a predicate is denoted in this chapter as $(1 \cdot 1 = 2 \cdot 1) \wedge (1 \cdot 2 = 2 \cdot 2)$. The linear pattern associated with a non-linear pattern is called its ***structure***; the predicate its ***binding predicate***. The structure is unique up to pattern equivalence, but the binding predicate is not: another binding predicate for the above pattern is $1 = 2$, where 1 and 2 are positions.

All the binding predicates that we consider are like the ones shown above: if ρ is the pattern, a binding predicate for ρ has the form $\bigwedge_i p_i = q_i$, where, for each i, p_i and q_i are positions in ρ such that $\rho_{@p_i} = \rho_{@q_i}$. A binding predicate is evaluated by comparing the trees that the match of the structure assigns to the positions mentioned in the binding predicate. The actual comparison may involve a tree comparison if a traditional representation of the subject tree is used, or a simple pointer comparison if the subject tree is represented as a computation dag.

The view of non-linear pattern matching as linear pattern matching together with predicate testing leads to several algorithms for non-linear pattern matching that are modifications of algorithms in Chapter 3. The simplest of these algorithms is presented in Section 4.1.2 and uses a subpattern LB-fsa to find subtrees of the subject tree that match the structural patterns; then these "candidates" are tested with binding predicates to find the real matches. This algorithm is quite straightforward and does not use any knowledge specific to non-linear patterns; it could be used equally well if the binding predicate were replaced by another (more general) semantic predicate.

In Chapter 3 we showed that there is a big reduction in matching time if a match set LB-fsa is used instead of a subpattern LB-fsa. In Section 4.1.3 we show how to apply these ideas to non-linear pattern matching. The resulting algorithm is more complex than the corresponding one in Chapter 3 due to some characteristics of the notion of the structure of a pattern.

The algorithms of Sections 4.1.2 and 4.1.3 require testing collections of binding predicates. Section 4.1.4 shows how to select, for a given pattern, a binding predicate with as few tests as possible. This predicate is the one that compares subtrees as high as possible. For the pattern at the beginning of this introduction, $1=2$ is the best predicate. Section 4.1.4 also contains some analysis of how to evaluate efficiently binding predicates that share some sub-predicates.

All the algorithms for non-linear pattern matching in this chapter characterize the subtrees of the subject tree with two parts: one of a finite number of states, and additional information referring to the subject tree. In the algorithms of Sections 4.1.2 and 4.1.3 the state is either a list of structures or a pattern equivalent to their meet, and the additional information is a pointer to the node being visited. Section 4.2 provides a definition that enrichs the notion of state. A ***p-pattern*** is a linear pattern extended with a collection of equalities and inequalities between its variables. For example, $<*(+(X_1,X_2),*(X_3,X_4)),\ \{1\cdot 1=1\cdot 2, 2\cdot 1\neq 2\cdot 2\}>$ is a p-pattern. P-patterns are used to record information that is available from the evaluation of binding predicates. If T is a tree of the form $op(T_1,\ldots,T_n)$, knowing that p-patterns $\phi_1,\ldots,\phi_n$ match at $T_1,\ldots,T_n$ may make it possible to replace the binding predicate at T by a simpler predicate. It is possible to use p-patterns to obtain several different improved pattern matching algorithms; Section 4.3 shows one particular alternative based on a variation on a deterministic B-fsa.

Section 4.4 presents a pattern matching algorithm by Purdom and Brown [PuB87]. Their algorithm suggests another approach to extending the information associated with the node in the subject tree at matching time: extend the dynamic information instead of extending the state. Specifically, the idea is to keep a list of all the subterms of the node that may be used in binding predicates of this node and its ancestors. The basic premise of this approach is that extracting these subterms is an expensive operation and that it may be cheaper to extract them when the node is being visited and to propagate them up the tree. Although most tree representations do not justify this premise, Section 4.4 briefly analyzes this approach.

Unlike other results in this dissertation, none of the algorithms presented in this chapter have been implemented, and we cannot provide definitive answers on their applicability.

4.1. Non-Linear Matching = Linear Matching + Binding Predicate

This section presents the simplest algorithms for non-linear pattern matching. They are based on the idea of applying the algorithms for linear pattern matching of Chapter 3 to the ''structures'' of the original pattern set.

4.1.1. Basic Definitions

The central definitions are those of the structure of a pattern and its binding predicate.

Definition 4.1 *The **structure** of a pattern ρ is a pattern in which each occurrence of a variable in ρ is replaced by a new variable. A **binding predicate** of a pattern ρ is a partial predicate $P_\rho(T)$ on trees such that, if a structure of ρ matches T, then ρ matches at T if and only if $P_\rho(T)$ is true. If a structure of ρ does not match T, P_ρ is undefined.*

The above definition of binding predicate is well-defined because all the structures of a pattern are equivalent. As mentioned previously, all the binding predicates of a pattern ρ considered in this chapter have the form: $\bigwedge_i p_i = q_i$, where, for each i, p_i and q_i are positions in ρ such that $\rho_{@p_i}=\rho_{@q_i}$.

As in Chapter 3, we will use several $\equiv$-reduced sets of patterns to simplify later definitions.

Definition 4.2 *If F is a pattern set, Z_F is a $\equiv$-reduction of the set of all patterns ζ which are structures of a pattern $\rho \in F$. Z_F is a pattern set; ΠZ_F is an equivalent denotation for Π_{Z_F}.*

We define Σ_F as an $\equiv$-reduction of the set of all patterns ρ_σ that are meets of match sets σ in F, and $Z\Sigma_F$ as an $\equiv$-reduction of the set of all structures of patterns in Σ_F.

For any pattern $\rho \in \Pi_F$, ζ_ρ is the pattern in ΠZ_F which is a structure of ρ. If σ is a match set of F, then ζ_σ, is the pattern in $Z\Sigma_F$ which is a structure of the meet of σ.

The next proposition lists some properties of the definition of structure. Note that (4) and (5) show that the notion of structural match set as defined above is different from the meet of the structures of all the subpatterns in the match set.

Proposition 4.1 *Let F be a non-linear pattern set. Let ρ, ρ_1, and ρ_2 be patterns in F. The following properties are true:*

(1) *If $\rho_1 \geq \rho_2$, then $\zeta_{\rho_1} \geq \zeta_{\rho_2}$.*

(2) $\zeta_\rho \oplus \rho = \rho$.

(3) *If $\rho_1 \oplus \rho_2$ exists, then $\zeta_{\rho_1} \oplus \zeta_{\rho_2}$ also exists and $\zeta_{\rho_1 \oplus \rho_2} \geq \zeta_{\rho_1} \oplus \zeta_{\rho_2}$.*

(4) *There are cases where the above implication may be strict.*

(5) *$\zeta_{\rho_1} \oplus \zeta_{\rho_2}$ may exist even if $\rho_1 \oplus \rho_2$ does not.*

Proofs

(1) and (2) are straightforward.

(3). By definition of $\oplus$ and (1), $\zeta_{\rho_1 \oplus \rho_2} \geq \zeta_{\rho_1}$ and $\zeta_{\rho_1 \oplus \rho_2} \geq \zeta_{\rho_2}$. A further application of the definition of $\oplus$ yields the desired result.

(4). Proof by example. Let ρ_1 be $+(X,X)$, and let ρ_2 be $+(*(X,Y),Z)$. $\rho_1 \oplus \rho_2 = +(*(X,Y),*(X,Y))$, $\zeta_{\rho_1 \oplus \rho_2} = +(*(X_1,X_2),*(X_3,X_4))$, but $\zeta_{\rho_1} \oplus \zeta_{\rho_2} = +(*(X_1,X_2),X_3)$.

(5). Proof by example. Let ρ_1 and ρ_2 be $*(+(X,Y),X)$ and $*(X,+(X,Y))$. □

4.1.2. A Simple Subpattern Matching Algorithm

The first algorithm for non-linear pattern matching presented in this chapter is also the simplest. The algorithm uses a subpattern LB-fsa for the set of the structures of the original pattern set. The LB-fsa is used to find the structural subpatterns matching at each node of the subject tree. Once any one such structure is found it is necessary to determine to what original pattern it corresponds. The notion of discrimination set encodes this information.

Definition 4.3 *Let F be a non-linear pattern set and let $\psi \in \Pi_F$. A discrimination set $DS(\zeta_\psi)$ of ζ_ψ is a collection of pairs $<\rho, P_\rho>$ such that $\rho \in \Pi_F$, $\zeta_\psi = \zeta_\rho$ and P_ρ is a binding predicate for ρ.*

Proposition 4.2 *Let F be a non-linear pattern set over Op and let T be a tree over Op. The algorithm of Figure 4.1 correctly computes the (explicit) match sets over F for every subtree t of T.*

Proof Straightforward. □

```
find the set of structures matching at every subtree t of T using
  the subpattern B-fsa of Z_F.
let DS(ζ) be the discrimination set of ζ;
for each subtree t of T do
    for each structure ζ in the structural match set do
        set σ(t)=∅;
        for each <ρ,P_ρ> ∈ DS(ζ) do
            if P_ρ(t) then
                σ(t)=σ(t)∪ρ;
```

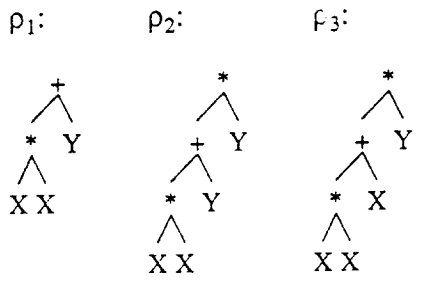
Matching Using Structural Subpatterns

Figure 4.1

The above algorithm is correct but slow. All the problems associated with keeping track of all the subpatterns explicitly are compounded by having to test a (potentially) large number of binding predicates for each subpattern.

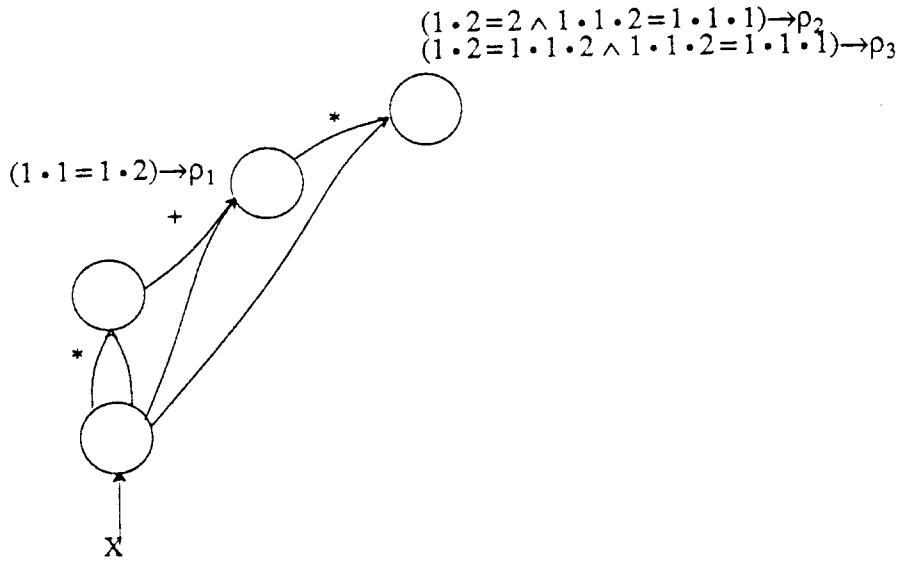
Normally, we assume that we are only interested in the **patterns** of F, not in its **subpatterns**. The notion of discrimination set can be changed to acknowledge this assumption by including only those patterns ρ that are members of F. This would lead to a somewhat faster pattern matching algorithm.

Figure 4.2 contains an example of a non-linear pattern set, while Figure 4.3 shows the B-fsa representing the computation of the structural subpatterns for that pattern set and the discrimination sets associated with each structural subpattern.



Example of a Pattern Set

Figure 4.2



B-fsa Used for Matching Using Structural Subpatterns

Figure 4.3

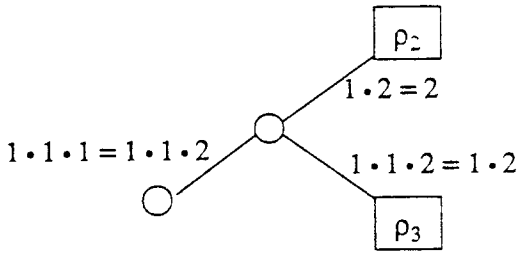
This example provides a good motivation for the notion of a ***discrimination tree***. The discrimination set for some structural subpatterns (as in the case of $*(+(*(X_1,X_2),X_3),X_4)$ above) may contain binding predicates that duplicate equality tests (such as $1 \cdot 1 \cdot 2 = 1 \cdot 1 \cdot 1$ above). A discrimination tree helps to share the result of those tests. There are two types of discrimination trees, one for the algorithm of this subsection and the other for the one shown below in Section 4.1.3.

Definition 4.4 *Let ρ be a pattern. An* ***all-discrimination tree*** *for ρ is an ordered tree in which each edge is labeled with either "$p_1 = p_2$", for p_1 and p_2 positions in ρ, or with "**true**", and in which the leaves are labeled. If DT is an all-discrimination tree for ρ, and ρ matches at T, the sequence of* **valid labels** *for DT and T is the ordered sequence of labels of those leaves reached in an ordered depth first search of DT, such that all the predicates in the path from the root to the leaf evaluate to **true** at T.*

A ***first-discrimination tree*** *for ρ is an ordered binary tree in which each edge is labeled with either **true** or **false**, each internal node is labeled with "$p_1 = p_2$", for p_1 and p_2 positions in ρ, and each leave is labeled. If DT is a first-discrimination tree for ρ, and ρ matches at T, the* **valid label** *for DT and T is the label (if any) of the leaf that is found by traversing DT, starting at the root, evaluating at each internal node reached its predicate and selecting one of the two outgoing paths from the node depending on the outcome of the test.*

Let ρ be a pattern. Given a sequence of predicates S, each one a conjunction of equalities between positions in ρ, an all-discrimination tree for ρ, DT, is **correct** *if for every tree T at which ρ matches, the set of valid labels for DT and ρ equals the set of predicates of S true at T. A first-discrimination tree DT for ρ is* **correct** *if for every tree T at which ρ matches, the first valid label for DT and ρ is the first predicate in S that evaluates to **true** for T.*

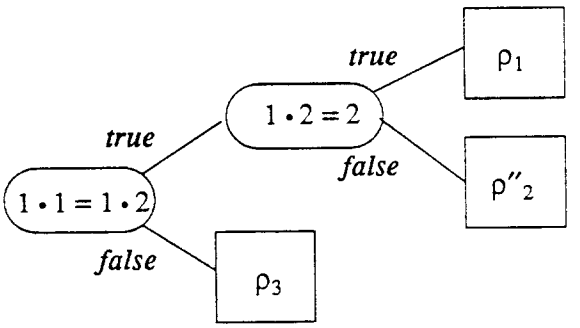
All-discrimination trees can be used to compute efficiently the set of all the valid predicates for a discrimination set of a structure. For example, a discrimination tree for the patterns ρ_2 and ρ_3 of Figure 4.2 is shown in Figure 4.4.



Example of an All-Discrimination Tree

Figure 4.4

An example of a first-discrimination tree, for a different set of patterns is shown in Figure 4.5. This discrimination tree will be used later in this section, is:



A First Example of a First-Discrimination Tree

Figure 4.5

The cost of a discrimination tree can be defined (for example) as the number of predicates (edges) in the tree. Other alternatives could include the expected number of predicates that have to be tested for some input distribution. Algorithms to compute minimum cost correct discrimination trees have not been developed, but Section 4.1.4 below contains some related results.

4.1.3. A Simple Match Set Pattern Matching Algorithm

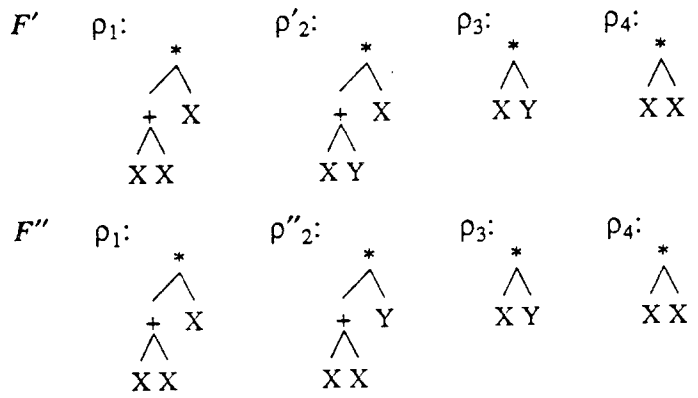
Point (5) in Proposition 4.1 indicates that structural match sets are not closed under $\oplus$. Hence, the algorithm to use structural match sets uses as state information the **closure** of $Z\Sigma_F$ under $\oplus$. Let this set be denoted by $Z\Sigma_F{}^{\oplus}$. It is possible to define a notion of discrimination set similar to the one in structural subpatterns for the members of $Z\Sigma_F{}^{\oplus}$.

Definition 4.5 *Let F be a non-linear pattern set over Op. If $\gamma \in Z\Sigma_F{}^{\oplus}$ a discrimination set $DS(\gamma)$ of γ is a collection of pairs $<\sigma, P_\sigma>$, where σ is a match set in F, P_σ is a binding predicate associated with σ, and σ satisfies the following two properties:*

(i) $\gamma \geq \zeta_\sigma$, *and*

(ii) there does not exist a match set ϕ_1 in F such that $\gamma \geq \zeta_{\phi_1}$ and $\phi_1 \oplus \sigma \equiv \phi_2$ with $\zeta_{\phi_2} > \gamma$.

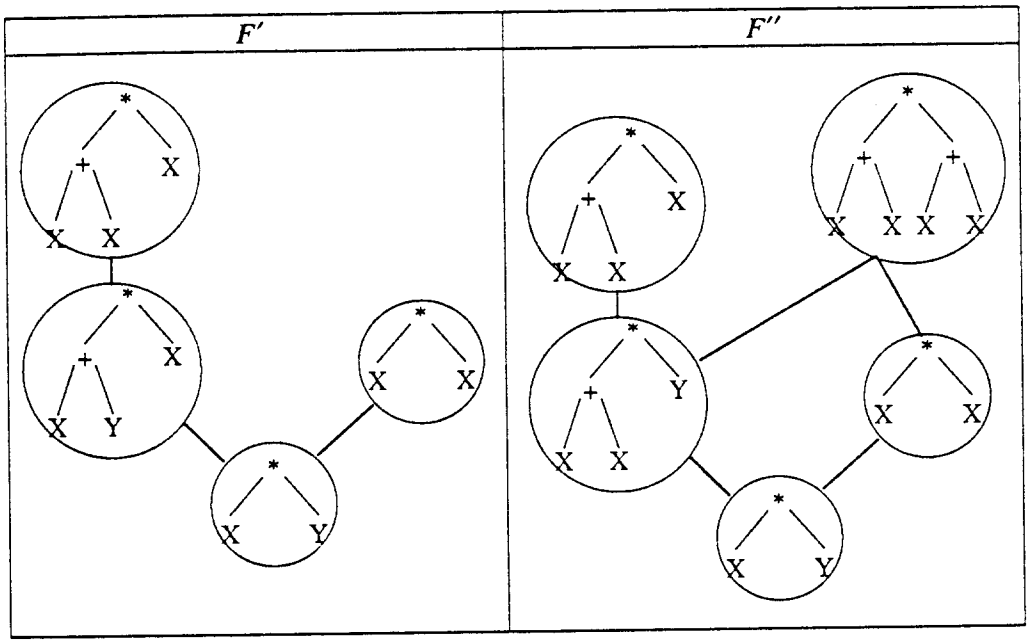
Intuitively, the definition of $DS(\gamma)$ is trying to capture all the possible match sets that could have γ representing their structure. Condition (ii) is intended to eliminate those structural match sets whose membership in $DS(\gamma)$ would imply that a member $Z\Sigma_F{}^{\oplus}$ larger than γ matches at the node. As an example of the role of (ii) consider the two pattern sets F' and F'' of Figure 4.6.



A Pattern Set for Match Set Pattern Matching

Figure 4.6

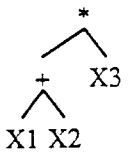
The subsumes relations for Σ_F in the two examples are shown in Figure 4.7.



Different Subsumes Relations

Figure 4.7

Now consider the structural match set represented by a pattern γ



and its discrimination set in each case. For the pattern set F' the match sets in the discrimination set of γ are $*(+(X,X),X)$, $*(+(X,X),Y)$, $*(X,X)$, and $*(X,Y)$ but, for the pattern set F'', the match sets for γ are $*(+(X,X),X)$, $*(+(X,X),Y)$, and $*(X,Y)$. $*(X,X)$ does not appear in the second case because, were that pattern a possibility, the structural match set would have been $*(+(X_1,X_2),+(X_3,X_4))$.

Discrimination sets are used in a pattern matching algorithm based on match set B-fsa.

Proposition 4.3 *Let F be a non-linear pattern set over Op, and let T be a tree over Op. The algorithm of Figure 4.8 correctly computes the match sets over F for every subtree t of T.*
Proof The correctness of the algorithm follows from the definition of discrimination set of γ. Condition (i) includes all the match sets of interest. Condition (ii) discards those match sets that would have required a structural match set different from γ. □

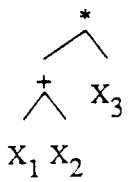
```
let B be a (deterministic) match set B-fsa for ZΣ_F^⊕;
assign to each subtree t of T a pattern in ZΣ_F^⊕ using B;
for each subtree t of T do
    let γ be the pattern in ZΣ_F^⊕ assigned to t;
    let DS(γ) be the discrimination set of γ;
    for each <σ,P_σ> ∈ DS(γ) in ≥ order do
        if P_ρ(t) then
            let the match set of t be σ;
```

Matching Using Structural Match Sets

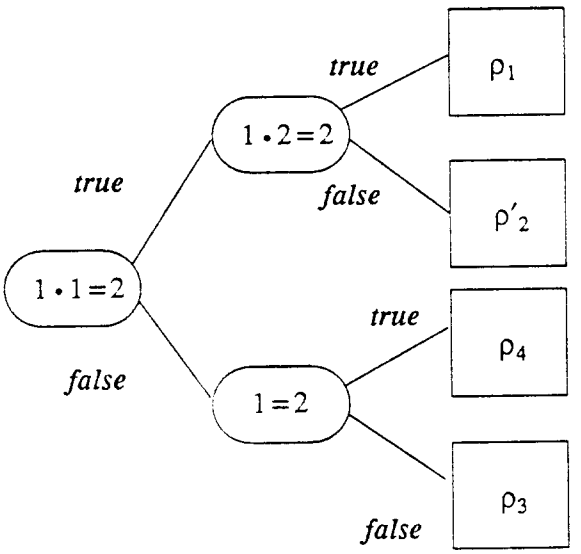
Figure 4.8

The above algorithm assigns to each structural match set a list of match sets whose binding predicates then are evaluated sequentially to find the largest valid one. The evaluation of this sequence may contain redundancies. An alternative is to associate with each structural match set, a first-discrimination tree.

For the pattern set F'' of Figure 4.6, a first-discrimination tree for



is the one shown previously in Figure 4.5, while the one for F' is



4.1.4. Optimal Binding Predicates

In general there is more than one valid binding predicate for a given pattern. Some of them may be more expensive, that is, perform more tests, than others. The following proposition tells how to find one with minimum cost.

Proposition 4.4 *Let* ρ *be a pattern. Let* $S(\rho)$ *be the class of binding predicates for* ρ *of the form* $\bigwedge_i p_i = q_i$, *where, for each* i, p_i *and* q_i *are positions in* ρ *such that* $\rho_{@p_i} = \rho_{@q_i}$. *There is an algorithm to find a member of* S *with a minimum number of equality tests.*

Proof Let $C(\rho)$ be the partition of the positions in ρ so that two positions are in the same block if they correspond to identical subtrees in ρ. Define a relation $\geq$ between the blocks of $C(\rho)$ by $P_i \geq P_j$ if and only if a position in P_i is an ancestor (a prefix) of a position in P_j, $i \neq j$. The relation $\geq$ is well defined, and is a partial order. $\geq$ is well defined because otherwise there would be tree positions p, q, p', and q' such that p is ancestor of q, and q' is ancestor of p', and with $\rho_{@p} \equiv \rho_{@p'}$ and $\rho_{@q} \equiv \rho_{@q'}$, which leads to a subtree being inside itself, which is impossible for finite trees. The proof of $\geq$ being a partial order is straightforward and is left to the reader.

The algorithm needs some additional concepts. For each block P_i in $C(\rho)$, we consider a complete graph, $G(P_i)$, with nodes the positions in the block; each edge represents a predicate on positions of ρ. An edge $<p,q>$ is a **descendant** of another edge $<p',q'>$ if there is a position s such that $p = p'/\!/s$ and $q = q'/\!/s$. Two edges are **connected** if they are of the form $<p_1,p_2>$ and $<p_2,p_3>$; their **connection** is $<p_1,p_3>$.

These last two relations satisfy that, if an edge is true, all its descendants are true; and, if two connected edges are true, their connection is true. Given that a set of edges (for some of the complete graphs mentioned above) are known to be true, applying repeatedly the two propositions above is known as "taking" the closure of the graph under connection and descendant.

Let E' be the set of edges in the complete graphs returned by an invocation to the algorithm of Figure 4.9. To prove that $\bigwedge_{<p_i,q_i> \in E'} p_i = q_i$, has minimum cost we first prove that an optimal binding predicate is a possible output of the algorithm, which follows from the following considerations.

- All the edges in E' must belong to the graph of some block in $C(\rho)$. Otherwise we perform either a trivially true comparison – which can be removed – or a trivially false comparison – an incorrect predicate – or a comparison that can be made to fail in trees where ρ matches.
- E' contains no edges that can be obtained from the others by applying transitivity or the descendant relation (see above).
- The transitive and descendant closure of E' contains all the edges in the graphs of the blocks of $C(\rho)$.

Finally, all possible outcomes of the algorithm of Figure 4.9 have the same number of edges. This follows because all the topological orderings of the blocks of $C(\rho)$ produce the same result, and because all minimum spanning trees have the same number of edges. □

```
let P_1, ..., P_n be an ordering of the blocks of C(ρ) following ≥;
let G(P_i) be the complete graph with P_i as nodes;
mark all the edges in each G(P_i) as "unknown";
E = ∅;
for each i from 1 to n do
    choose a minimum spanning tree MST for the "unknown"edges of G(P_i);
    E = E ∪ all the edges of MST;
    mark all the edges of G(P_i) as "known";
    comment now compute the closure under connection and descendant
    repeat
        if a " known " edge e has an unknown descendant e' then
            mark e' as "known";
        if two " known " edges e_1 and e_2 are connected and
          their connection is an "unknown" edge e' then
            mark e' as "known";
    until there are no more changes;
```

Algorithm to Find Optimal Binding Predicates of Patterns

Figure 4.9

Another algorithm of interest is how to compute, given a discrimination set, a discrimination tree for it. Ideally we would like the discrimination tree to minimize some cost function like the number of edges in the tree. We do not have a solution to that problem, but the algorithm presented in the proof of the following proposition contains a heuristic that tries to attain the goal. The "goodness" of the heuristic cannot be measured until after it is implemented and compared with the optimal, maybe found with some type of exhaustive search.

Proposition 4.5 *Let F be a non-linear pattern set, and let $\zeta \in Z\Sigma_F$. There is an algorithm that will select a first-discrimination tree implementing a discrimination set for ζ.*

Let F be a non-linear pattern set, and let $\zeta \in Z_F$. There is an algorithm that will select an all-discrimination tree implementing a discrimination set for ζ.

Proof We solve the problem involving a match set represented by some pattern ρ; the problem for structural subpatterns is similar. Let S be the subset of Σ_F that appear in any discrimination set for ρ (Definition 4.5). For every pair of positions in ρ, let $count(p_1,p_2)$ be the number of patterns in S in which the subtrees at p_1 and p_2 are identical. Now use the algorithm of Figure 4.10.

The correctness of the algorithm follows from its use of the algorithm of Figure 4.9 to compute the individual binding predicates for each match set. The use of $count(p_1,p_2)$ attempts to share equality tests between the different binding predicates, and the sort tries to put the queries in the right order.

This heuristic is clearly non-optimal. Other heuristics could be used. For instance, one could try to select the minimum spanning tree so as to choose previously used equality tests if possible. □

```
choose one ordering of S following ≥ and let it be {σ1, . . . , σn };
for each σi do
    let Ei be the set of edges <p1,p2> returned by the algorithm of Figure 4.9,
     modified so that the minimum spanning tree is chosen using first edges with highest count();
let E be ⋃_{i=1}^{n} Ei ;
for i from n downto 1 do
    move to the front of E those edges in Ei ;
    comment the move is supposed to preserve the previous order;
for i from 1 to n do
    lay a path using Ei sharing as much as possible with
     the previous paths;
```

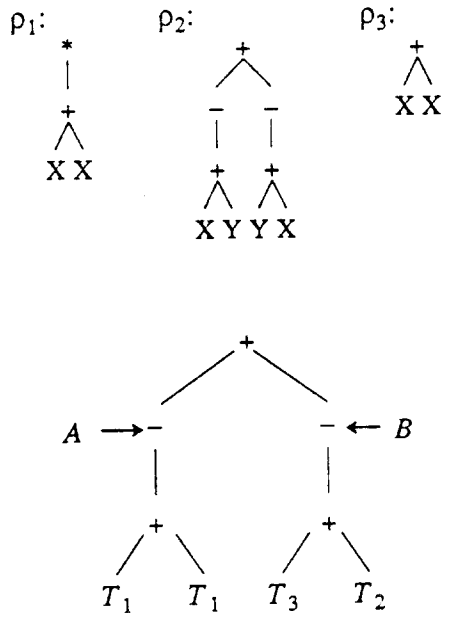
Finding a Discrimination Tree

Figure 4.10

4.2. P-Patterns

The idea behind the notion of p-patterns introduced in this section is to evaluate the binding predicates of the children **before** computing the state of the parent and to extend the linear patterns used to represent states in the previous sections with equalities and inequalities to record the results of these tests.

As a motivation, consider the pattern set composed of the two patterns at the top of Figure 4.11, and the subject tree at the bottom of the figure. Assume that T_1, T_2, and T_3 are all different subtrees.



P-patterns are Better than Patterns

Figure 4.11

The children of the subtrees marked A and B have match sets represented by $+(X,X)$ and $+(Y,Z)$, respectively. Accordingly, at some point we have to test binding predicates and determine whether T_1 and T_2 and also T_1 and T_3 are equal so as to determine if we have found ρ_2. If we encode the state information as members of $Z\Sigma_F^{\oplus}$, nodes A and B will be characterized as $-(+(X,Y))$, which contains no information about the equality between X and Y, and the binding predicate (for ρ_2) at the root would be $(1\cdot1\cdot1=2\cdot1\cdot2)\wedge(1\cdot1\cdot2=2\cdot1\cdot1)$. Instead, if the test of the binding predicates at the children of A and B is done **before** computing the states of A and B, **and** the result of the test is incorporated into states $<-(+(X,Y)),\{1\cdot1=1\cdot2\}>$ for A and $<-(+(X,Y)),\{1\cdot1\neq1\cdot2\}>$ for B, the binding predicate at the root for ρ_2 is equivalent to ***false***.

Definition 4.6 *Let **F** be a non-linear pattern set over Op. A **p-pattern** over **F** is a pair $<\zeta,E>$, where ζ is a linear pattern and E is a conjunction of equalities and inequalities between locations of variables in ζ. A p-pattern $<\zeta,E>$ **matches** at some term T if ζ matches at T with assignment α, and all the equations in E are satisfied by α.*

ζ in Definition 4.6 will be either a member of Z_F or a member of $Z\Sigma_F^{\oplus}$. If the set E is empty, we obtain the notions of structures that we used in the previous section.

P-patterns are used to represent state information. Thus, we want to find how to compute the new states given the previous ones. For this, we need the notion of an incremental binding predicate.

Definition 4.7 *Let F be a non-linear pattern set F over Op. Let ϕ be a p-pattern over F. The incremental binding predicate for a pattern $\rho \in F$ satisfying ϕ is a partial predicate, $P_{\rho,\phi}(T)$, such that, if ζ_ρ matches at T, and ϕ matches at T, then $P_{\rho,\phi}(T)$ is true if and only if ρ matches at T.*

Definition 4.7 is a slight generalization of what is needed as the equalities and inequalities in ϕ may span across different children of its structure.

A good incremental binding predicate can be computed.

Proposition 4.6 *Let F be a non-linear pattern set over Op. Let ϕ be a p-pattern over F, and $\rho \in \Pi_F$. Let S be the class of incremental binding predicates for ρ satisfying ϕ of the form $\bigwedge_i p_i = q_i$, where, for each i, p_i and q_i are positions in ρ such that $\rho_{@p_i} = \rho_{@q_i}$. There is an algorithm to compute a member of S with the smallest number of comparisons.*

Proof The algorithm used is a variation of the one in Proposition 4.4. Let ϕ be $<\phi', E>$. If ρ cannot be unified with ϕ' then the desired predicate is *false*. If it can, let f be their most general unifier (Definition 2.10).

Let $C(f(\rho))$ be the partition of the positions in $f(\rho)$ such that two positions are in the same block if they correspond to identical subtrees of $f(\rho)$. If there is an inequality in E between positions p and q and those positions are in the same block in $C(f(\rho))$ the desired predicate is (again) *false*.

Otherwise, apply the algorithm of Figure 4.9 modified so that instead of initially "marking" all the edges in $G(P_i)$ as "unknown", an edge $<p,q>$ is marked "known" if $(p = q) \in E$, and "unknown" otherwise, and after this the graphs are closed under *connection* and *descendant*. The rest of the algorithm stays the same. □

The other notion needed for obtaining a pattern matching algorithm is:

Definition 4.8 *Let F be a (non-linear) pattern set. Let $\phi = <\zeta, E>$ be a p-pattern of F. A* **discrimination set** *of ϕ, $DS(\phi)$, is a collection of pairs $<P_\sigma, \sigma>$ where σ is a match set in F satisfying (i) $\gamma \geq \zeta_\sigma$, (ii) there does not exist a match set ϕ_1 in F such that $\gamma \geq \zeta_{\phi_1}$ and $(\phi_1 \oplus \sigma) = \gamma_2$ with $\zeta_{\gamma_2} > \gamma$, and, and P_σ is an incremental binding predicate associated with σ such that P_σ is not equivalent to false.*

Finally, a discrimination tree can be obtained using the algorithm of Proposition 4.5.

4.3. A Match Set Algorithm Using P-Patterns

P-patterns can be used in several different ways to implement matching algorithms. One alternative is to construct a faster version of the match set pattern matcher described in Section 4.1.3; this is the alternative explored in this section. Another, less attractive alternative, would be to extend the algorithm of Section 4.1.2; this alternative is not explored in this chapter.

The idea followed in this section is to modify the algorithm of Section 4.1.3 so that, after testing some binding predicate the information gained by the evaluation is not thrown away but it is encoded in p-patterns which are then, in turn, used to simplify the later evaluation of other binding predicates.

The computation of the states associated with the nodes of the tree is no longer just a LB-fsa. There are two functions evaluated sequentially at each node of the input tree. The first function looks like

$$F_1 : Op \times p\text{-}pattern_1 \times \cdots \times p\text{-}pattern_n \rightarrow p\text{-}pattern$$

while the second takes the p-pattern produced by the first one and uses its discrimination set and the actual tree to compute the p-pattern associated with the node, that is,

$$F_2 : p\text{-}pattern \times T \rightarrow p\text{-}pattern$$

There are variations within this approach. One may always compute F_2, or only do it when there is some match set in the discrimination set that includes a pattern (as opposed to a tree that appears only as a subpattern of a pattern) in $\boldsymbol{F}$. The latter approach is a better one since it postpones testing the binding predicates until necessary, and some predicates may never be tested as their applicability is discarded due to structural reasons.

The matching algorithm can be described as follows:

procedure *find_state* (T: tree) ***return*** a state (a p-pattern)
    let $T = op(T_1, \ldots, T_n)$;
    for *i* ***from*** 1 ***to*** *n* ***do***
        ϕ_i=*find_state* (T_i);
    $\gamma = F_1(op, \phi_1, \ldots, \phi_n)$;
    $\phi = F_2(\gamma, T)$;
    comment depending on the policy followed and the discrimination set of γ, F_2 may just return γ;
    return γ;

$F_1(op, \phi_1, \ldots, \phi_n)$ is a simple computation:

Proposition 4.7 *Let F be a (non-linear) pattern set over Op. Let op be an n-ary operator in Op, and let $\phi_1, \ldots, \phi_n$ be p-patterns in F, with $\phi_i = \langle \zeta_i, E_i \rangle$. Let ζ be the largest (relative to $\geq$) member of $Z\Sigma_F^{\oplus}$ that matches at $op(\zeta_1, \ldots, \zeta_n)$. There is an algorithm that will compute the collection E of equalities and inequalities between variables in ζ such that for every tree $T = op(T_1, \ldots, T_n)$, ϕ_i matches at T_i for $1 \leq i \leq n$ if and only if $\langle \zeta, E \rangle$ matches at T.*

Proof Assume, without loss of generality, that the sets of variables of ϕ_i, for $1 \leq i \leq n$, are disjoint. Let f be the matching assignment for ζ and $op(\zeta_1, \ldots, \zeta_n)$. For each two variables X_1 and X_2 in ζ with positions p_1 and p_2, $p_1 = p_2$ is in E if the function $eq(f(X_1), f(X_2))$, defined below, returns *true*; $p_1 \neq p_2$ is in E if the function $ne(f(X_1), f(X_2))$, defined below, returns *true*.

$$eq(Op(X_1, \ldots, X_n), Op(X'_1, \ldots, X'_n)) \triangleq Op = Op' \wedge eq(X_1, X'_1) \wedge \cdots \wedge eq(X_n, X'_n)$$
$$eq(X, X') \triangleq X = X' \wedge (\exists i)(X = X' \in E_i)$$

$$ne(Op(X_1, \ldots, X_n), Op(X'_1, \ldots, X'_n)) \triangleq Op \neq Op' \vee ne(X_1, X'_1) \vee \cdots \vee ne(X_n, X'_n)$$
$$ne(X, X') \triangleq (\exists i)(X \not\equiv X' \in E_i)$$

The correctness of the algorithm is left to the reader. □

$F_2(\phi, T)$ are the equalities and inequalities known when a leaf in the discrimination tree of ϕ is reached (that is, equalities of those internal nodes whose ***true*** branch has been taken, inequalities of those whose ***false*** branch has been taken).

To implement the algorithm efficiently it is necessary to precompute F_1 and to encode $F_2(\phi, T)$ for all useful ϕ. This means finding all the "useful" p-patterns. This can be done using a closure algorithm. In the algorithm, U contains the p-patterns in the image of F_2, (those used as states of the nodes of the tree) while V contains the p-patterns in the image of F_1, (those used as starting points for F_2).

```
U={<X,∅>};
V=∅;
repeat
    for each op ∈ Op do
        let n be the arity of op;
        for each φ1, ..., φn in U do
            φ=F1(op,φ1, ..., φn);
            V=V∪{φ};
            U=U∪{ all those φ obtained by evaluating F2(φ,T)};
until no more changes in U or V.
```

Finding all P-patterns

Figure 4.12

The algorithm in Figure 4.12 generates all the p-patterns that can be obtained using F_2, but does not take into consideration that some of them are useless, in the sense that some of the equalities and inequalities carried in the p-pattern are not used. Removing these p-patterns would lead to smaller tables (but no faster matching algorithms). This could be done using another closure algorithm to propagate useful equations backwards from the match sets containing at least one pattern (as opposed to a subpattern) of F.

Some equalities and inequalities can be determined useless by simple inspection. If an equality or inequality involves variables where at least one of the them appears only once in all patterns (representing match sets), then that equality or inequality can be dropped safely from the p-pattern.

Consider the pattern set containing the patterns ρ_1 and ρ_2 of Figure 4.11.

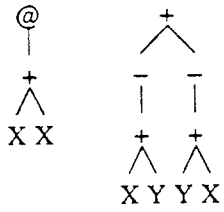
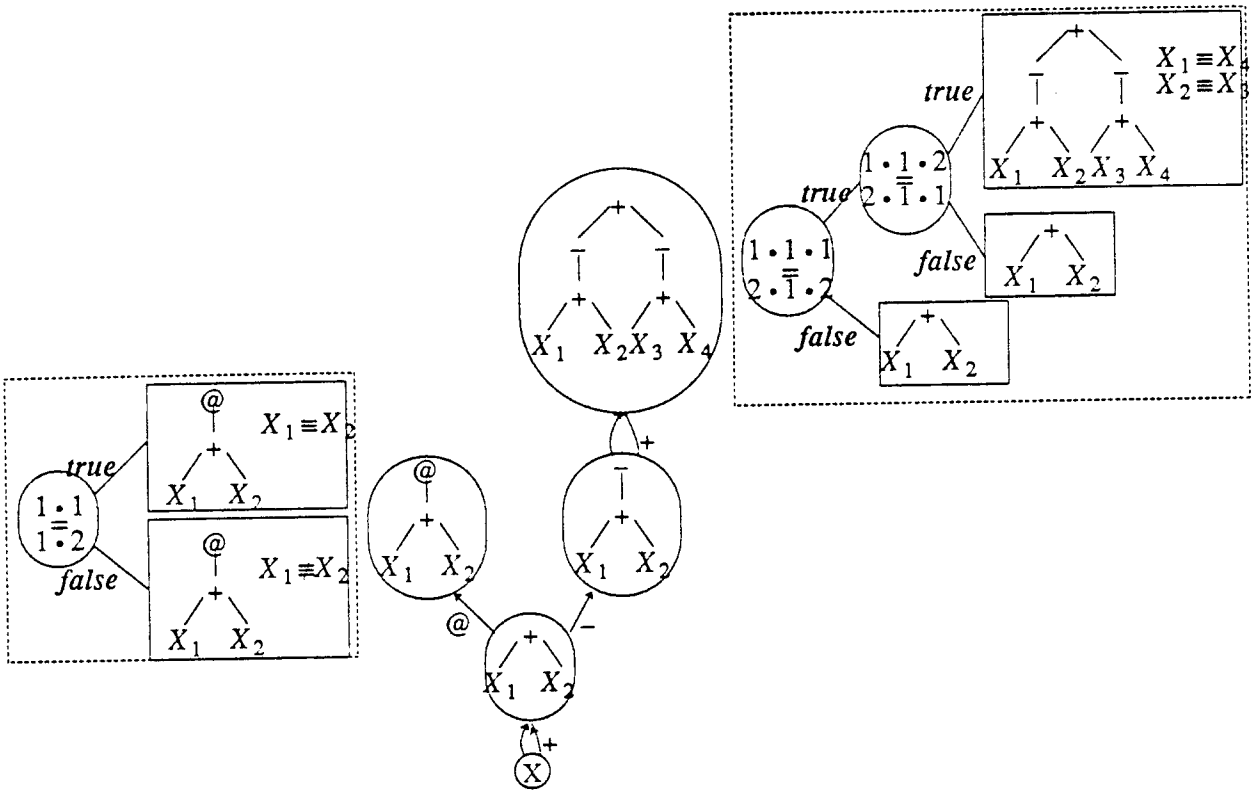


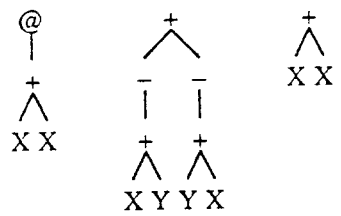
Figure 4.13 shows some of the most interesting parts of the two functions F_1 and F_2 for this pattern set. F_2 is represented by the first-discrimination trees inside the dotted boxes, and F_1 is represented by the other edges.



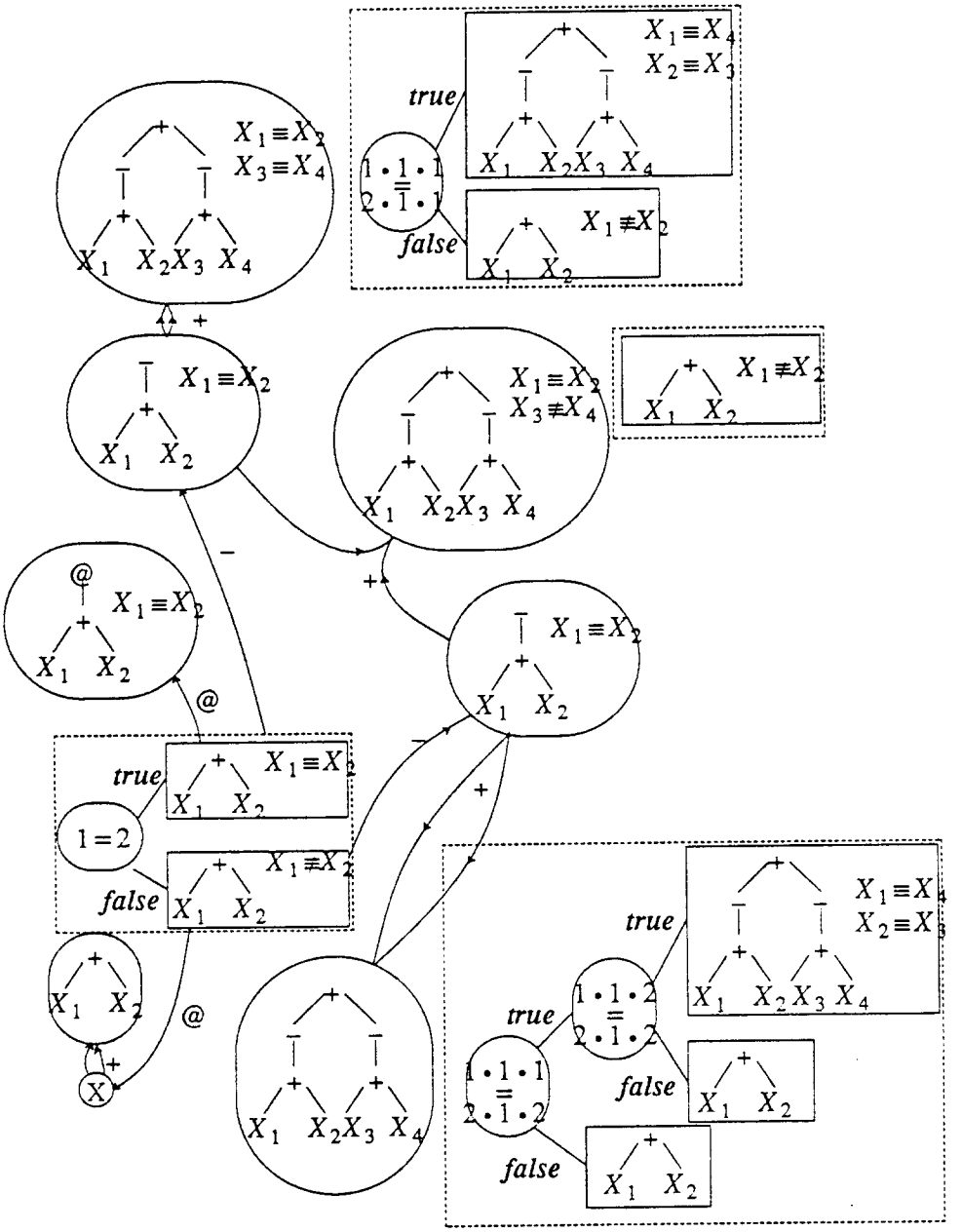
First Example of F_1 and F_2

Figure 4.13

Restoring the pattern $\rho_3 = +(X, X)$ to the pattern set changes things because it forces some tests to be performed earlier. The pattern set is:



and Figure 4.14 contains the most interesting parts of the new functions.



Second Example of F_1 and F_2

Figure 4.14

4.4. Explicit Term Representations and the Work of Purdom and Brown

The previous sections have assumed that the binding predicates have a single input: the subtree in the subject tree being analyzed. Thus, a reference to a position implies the extraction of a subtree. All uses of the same position in a discriminating tree may share the extraction cost, but references to the same position in different discriminating trees imply repeated extractions. The cost of these extractions depend on the representation of the tree. An alternative approach is to extend the state information that is manipulated at pattern matching time to include an explicit representation of the subterms needed for the evaluation of the binding predicate. The subterms can now be extracted when the extraction cost is minimum, at the point of the matching algorithm where the node representing the subterm is being visited, and they can be propagated from a node to its ancestors for as long as they are useful and some ancestor of the node might need them to evaluate a binding predicate.

Whether an explicit representation of the terms is cost-effective depends on the cost to store, retrieve, and propagate the representation of the subterms and its comparison against the cost to extract the subterms from the subject tree. It seems likely that the explicit representation is not the best choice for most tree representations. This section presents one algorithm based on an explicit term representation developed by Purdom and Brown [PuB87], and discusses briefly the problems involved in extending it.

The main issue in an algorithm involving an explicit term representation is determining what subterms have to be kept explicit, and where they should be placed for access by the binding predicates and for propagation. The algorithm by Purdom and Brown is intended for a Knuth-Bendix completion algorithm [KnB70] and, as such, places heavy emphasis on the ability to increase the pattern set dynamically[14]. The states used are explicit representations of match sets (which are, as we know, a special type of p-patterns). Associated with each subpattern in the match set there is a list of pointers to the subtrees in the subject tree that correspond to the variables in the subpattern. The encoding of the computation of the new subpatterns uses a "curried dag" (Section 3.2.2).

The binding predicate of [PuB87] belongs to the same class that we have used before in this chapter: $\bigwedge_i p_i = q_i$, where, for each i, p_i and q_i are positions in ρ such that $\rho_{@p_i} = \rho_{@q_i}$. More precisely:

Definition 4.9 *Let F be a non-linear pattern set, and let $\rho \in \Pi_F$. The slow incremental binding predicate of ρ is the predicate of the form $\bigwedge_i p_i = q_i$, where, for each i, p_i and q_i are positions in ρ labeled with the same variable and descendants of different children of the root of ρ.*

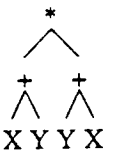
It is pretty straightforward that the slow incremental binding predicate is actually an incremental binding predicate as defined in Definition 4.7. We call this predicate "slow" because it may use more queries than the optimal binding predicates presented previously.

The main advantage of using the slow binding predicate is one of simplicity: no special effort is needed to determine what subterms to encode in the explicit representation and where to encode them. Simply, pointers to all the variables are kept in the (left-to-right) order in which

[14] Since the research reported in this dissertation does not consider changing pattern sets, straight comparisons are not directly meaningful. Another way to tackle the problem of "changing" pattern sets is to use a fast algorithm with the original pattern set and a patch for the updates. When the updates are beyond some threshold, then the complete new set can be analyzed to obtain new tables for the fast algorithm. In a multi-processor situation (which will be quite common in the near future, it seems), computing the new tables can be done concurrently while using the previous ones.

they appear in a linear array. The binding predicate for a pattern with an n-ary operator at the root uses n such arrays for its evaluation. If the predicate is **true**, the propagation algorithm combines the n arrays into a new array corresponding to the variables of the pattern.

Purdom and Brown show, in [PuB87], how to combine the two operations, testing the predicate and propagating values, into a single algorithm that uses a compact representation of the binding predicate. Figure 4.15 shows an example of the constructions used by the algorithm. The top row shows a non-linear pattern. The bottom row shows different versions of its incremental binding predicate. The slow incremental binding predicate is at the left. In the middle, the predicate and the propagation are described through four equations between variables in the left, and right children of the pattern, denoted as $left.V_i$ and $right.V_i$, and variables in the pattern itself, denoted as $self.V_i$. These equations are described in a compact form at the right by simply listing the slots in the arrays corresponding to the variables.



$(1 \cdot 1 = 2 \cdot 2)(1 \cdot 2 = 2 \cdot 1)$

$self.V_1 = left.V_1$ 1 2
$self.V_2 = left.V_2$ 2 1
$self.V_2 = right.V_1$
$self.V_1 = right.V_2$

Example of a Slow Incremental Binding Predicate

Figure 4.15

The compact representation at the right of the second row is called a "substitution map" in [PuB87]. See that paper for further details.

Extensions

If we compare the implementation of pattern matching presented in [PuB87] with one using the ideas of Section 4.1.3, we can note the following sources of inefficiency:

(1) The use of explicit match sets instead of implicit match sets.

(2) The use of non-optimal binding predicates.

(3) Somewhat slow testing of the binding predicate, as it has to be interpreted from the substitution maps.

The last point can be seen as just a detail of the implementation: the propagation and testing could have been done by direct encoding into a sequence of instructions, thus yielding a faster representation but also a probably larger table. The first two points are more important.

For any pattern matching algorithm, the goal of subterm represesentation is to assign a collection of subterms to a state at table construction time. The solving-time values of these subterms are then used either to evaluate the binding predicate or to compute the values of subterms

associated with nodes higher in the subject tree. The collection of subterms of interest can be found using a closure algorithm that starts by assigning to a state those subterms used by its binding predicate and then repeats computing new subterms until no more need to be found. Since we assume that the subterms can only be extracted when the algorithm is visiting the root of the node, any subterm that is needed and is not the root comes from a node lower in the tree; these are the subterms used in the closure.

An additional problem appears with algorithms involving match sets. In these algorithms, unlike those involving subpatterns, a given state may be reached in more than one "way". The problem is to decide where to place the subterms so that they are accessible to the binding predicates in a way independent of how the state was reached. For example, consider the pattern set

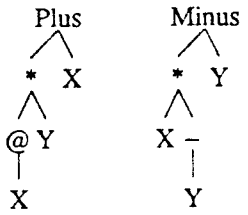
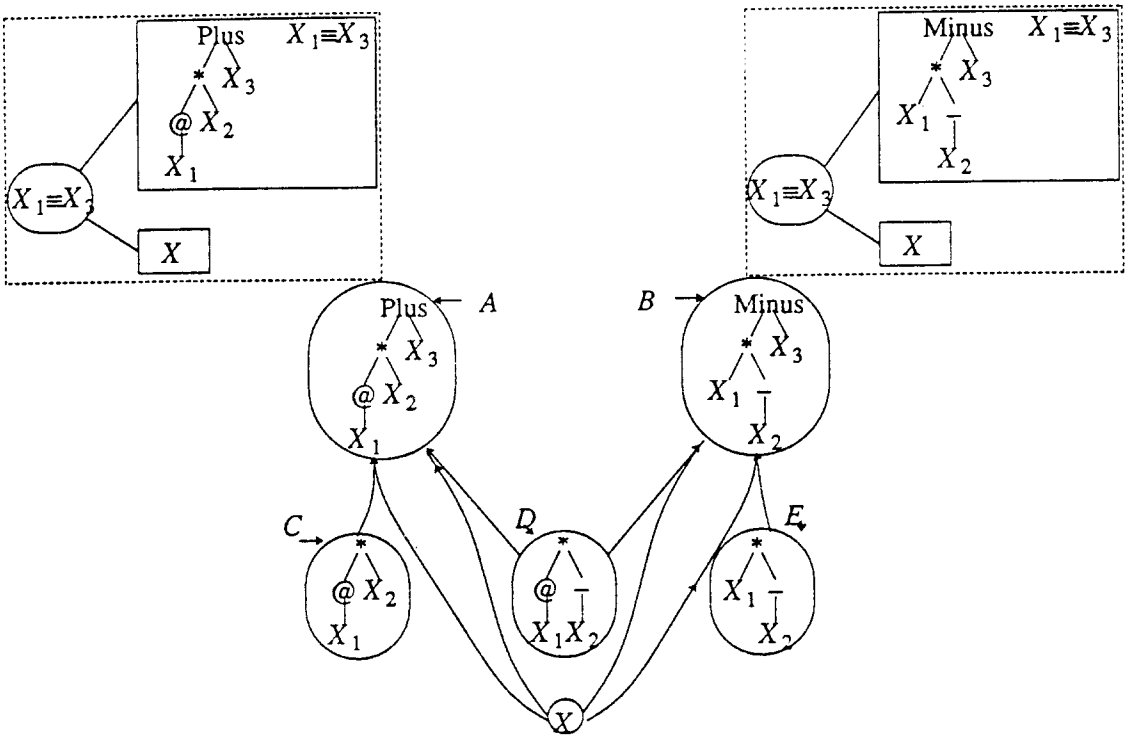


Figure 4.16 shows part of the functions F_1 and F_2 of Section 4.3 for this pattern set.



Example of Alignment Problems with Representation

Figure 4.16

The discrimination tree marked "A" requires the presence of the subterm of position $1 \cdot 1 \cdot 1$, while that at "B" requires $1 \cdot 2 \cdot 1$. The first value may come from either "C" or "D", while the second may come from either "D" or "E". The question is how to lay out the subterm representation so that "C", "D", and "E", all can place the pointers to the subterms in a location that can then be used by "A" and "B" disregarding how it got there. The answer is that either "E" or "C" will have to deal with a subterm representation that, from its standpoint, has a "hole". "C", "D", and "E" could use $[1 \cdot 1, 2 \cdot 1]$ as a representation. "C" would only store into the first slot and leave the second unmodified, "E" would only store into the second and leave the first unmodified, and "D" would store into both slots.

Choosing the slot assignment to minimize the number of "holes" might be non-trivial. If the minimization is ignored, which is a safe assumption in most cases, a valid slot assignment can be chosen by considering all the possible subterms required and leaving space for all of them.

4.5. Related Work

Top-Down Pattern Matching

As in the case of linear pattern matching, top-down algorithms are somewhat simpler, but slower, than bottom-up algorithms.

A very simple algorithm for non-linear N-pattern matching based on a top-down traversal is described in [Sny82]. The algorithm starts by detecting common sub-expressions in the subject tree. The algorithm then maintains a list of partial matchings, where a partial matching is composed of a pattern, a node in the subject tree, a (maybe incomplete) substitution, and an indication of what portion of the pattern has already been considered so far. This list is updated as the subject tree is traversed.

The algorithm requires traversing two lists at each node of the subject tree: the list of "active" partial patterns to enlarge, and the list of "possible" patterns to start. This makes the algorithm quite slow. There is no precomputation at all, and the list of patterns is searched always to determine which new patterns have to be added. It might be possible to pre-analyze the set of input patterns so that it is not necessary to traverse the whole list over and over again; probably some of the ideas in [KMR72] could be used. But it is not clear how to carry the idea further and get rid of the traversal of the list of active partial patterns.

Splitting Using the Number of Parents

The previous sections have emphasized how to do pattern matching using equality tests on a subject where common subexpressions have been detected and encoded in the form of a computation dag. One piece of information present in the computation dag that none of the previous algorithms has used explicitly is the number of parents of a node. In particular, if the number of parents of a node is just one, then the subterm described by the node is not being used in more than a single place in the subject. This information could be used to discard from consideration as members of the match set of the node all those patterns that would require the subterm to appear more than once.

The main advantage of such an approach would be that if a portion of the subject does not contain any common sub-expressions no tests for them would be considered, since the matching algorithm assigns to the nodes only states corresponding to subterms with a single parent. A state including binding predicates would be reached only when a subterm in the subject has the minimum number of parents.

This idea is not explored any further. Probably the most practical application of this idea would limit itself to differentiating subject nodes with one parent from those with more than one and likewise with the states. Such an approach could make up for the additional cost of checking the number of parents of the node.

In theory the algorithm is not fully satisfactory since there is a potential for a combinatorial explosion in the number of states. Also the algorithm does not attain the goal of doing pattern matching on a cost (including evaluation of binding predicates) linear on the number of edges of the original subject dag[15]. In some sense, this algorithm is just one method – not particularly clean – of making context present in the subject available to the matching algorithm. A cleaner solution could be a two-pass algorithm that would, first, in a top-down phase, transmit context information down the subject, and then, in a bottom-up phase, would collect it and find all the matches neatly.

4.6. Summary of the Algorithms in this Chapter

This chapter has several new contributions to the theory of non-linear pattern matching. The key notion is the equation ***non-linear matching = linear matching + binding test.*** The chapter has explored in some detail its implications, obtaining new algorithms that speed up the matching process by increasing the size of the tables representing the matching functions. In

[15] An example is left to the reader.

addition, the results on optimal binding predicates allow further reduction in the matching cost.

A nice property of the algorithms presented here is that the extra power of the non-linear matching is used (for the most part) only on those patterns that need it. If a linear pattern set is used, the algorithms degenerate to the linear pattern case. The only exception is with a naive implementation of explicit subterm representations using a substitution map, but careful encoding can avoid this problem.

This chapter has explored a moderate number of approaches to non-linear pattern matching. Lack of time has prevented the author from following the next obvious step: select a specific application for non-linear pattern matching, implement several of the algorithms, and measure their performance. This is a topic for future research.

CHAPTER 5

Bottom-Up Rewrite Systems

Everything that goes up must go down

[after Newton]

Recall the definition of REACHABILITY:

Definition 2.28 *Let R be a rewrite system over Op, and let L_{in} and L_{out} be two sets of trees over Op. The REACHABILITY problem for R, L_{in}, and L_{out} is, given $T \in L_{in}$ and $T' \in L_{out}$, to determine whether there is a rewrite sequence τ for R applicable at T such that $\tau(T) = T'$, and, if so, to produce one such τ.*

*If L_{out} is a singleton $\{G\}$, then the REACHABILITY problem is called the **fixed goal** REACHABILITY, and G is called the **goal**.*

In this chapter we study mostly the fixed-goal REACHABILITY problem for a particular class of rewrite systems. For the most part, the input language, L_{in} is restricted to be the set of all trees L_{Op}, although Section 5.5 answers some questions relative to any set $L_{in} \in$ RECOG. Variable-goal REACHABILITY is studied in Section 5.3.1. All rewrite rules contain only linear N-patterns.

REACHABILITY, even fixed goal REACHABILITY, is unsolvable in general, since it can model the HALTING problem, but it can be solved for special classes of rewrite systems. This chapter defines two classes of rewrite systems. **Bottom-up rewrite systems** (BURS) have an algorithm for solving REACHABILITY; **finite bottom-up rewrite systems** (finite BURS) have a very efficient algorithm for solving fixed goal REACHABILITY in time linear in the size of the input tree. Finite BURS have an important practical value: Chapter 6 shows how they can be used to solve locally optimal instruction selection, and Section 7.2 shows how they can be used to define tree languages and a generalized notion of homomorphism.

The algorithms to solve variable-goal REACHABILITY and fixed-goal REACHABILITY are based on several notions of state to be associated with the nodes of the input tree. All the notions satisfy two basic requirements:

(STATE-1) The collection of states associated with the nodes contains enough information to characterize all the "interesting" rewrite sequences that are applicable to the input tree.

(STATE-2) The states can be computed in a bottom-up pass over the input tree.

In all cases, the "interesting" rewrite sequences are restricted to be in a bottom-up **normal form**: all rewrite applications are done as low in the tree as possible. All non-looping rewrite sequences can be reordered so that they are in bottom-up normal form. A rewrite sequence for some input tree T in normal form assigns to each node N in T a **local rewrite sequence**: a sequence composed of the rewrite applications that cannot be done "below" N and do not need the result of applications to nodes "above" N. The class BURS contains all those rewrite systems for which there is a positive integer k such that all local rewrite sequences have at most k rewrite applications. Section 5.1 defines these notions. There are several useful subclasses of BURS. Many of them are based on the idea of **reduction systems**: rewrite systems in which the rules always "reduce" the size of the input tree. Reduction systems are presented in Section 5.2.

We use three notions of state in this chapter: proto-states, local rewrite graphs (LR graphs), and uniquely invertible local rewrite graphs (UI LR graphs). The first notion is used to solve variable-goal REACHABILITY in BURS, the last two are used for fixed-goal REACHABILITY in finite BURS.

The "interesting" rewrite sequences considered in the **proto-state** associated with a node are all the normal form rewrite sequences applicable to the subtree of the input tree rooted by the node. Proto-states encode the result of applying these rewrite sequences to the input tree and allow a three-pass algorithm to solve variable-goal REACHABILITY. The proto-states associated with the nodes of an input tree can be computed in a bottom-up pass; then they can be consulted by a top-down traversal to find (if it exists) a local rewrite sequence associated with each node, which, finally, can be collected in a final bottom-up pass to obtain a rewrite sequence transforming the input tree into the output tree.

Proto-states can contain an unbounded amount of information, which leads to inefficient algorithms. This is not necessary in many applications. In the notion of a ***local rewrite graph*** (LR graph) used for fixed goal REACHABILITY (Section 5.3), the "interesting" rewrite sequences considered are all those normal form rewrite sequences that are applicable[16] to the subtree of the input tree rooted by the node and also can lead to the goal tree. The information stored in the state is the effect of the local rewrite sequences on members of an **extended pattern set.** In general, this set can be infinite; if it is finite, the rewrite system is said to be finite BURS, and fixed-goal REACHABILITY can be solved very efficiently.

LR graphs contain more information than is needed to solve fixed-goal REACHABILITY; the related notion of a **uniquely invertible** LR graph (UI LR graph) leads to a smaller number of graphs (where each graph corresponds to a state). This notion is obtained by restricting further the set of "interesting" rewrite sequences. UI LR graphs are introduced in Section 5.4. Section 5.5 discusses some modifications to the algorithms and definitions that can be made when the input set of interest is some recognizable tree language. The implementation considerations of the algorithm are discussed in Section 5.6.

The chapter concludes with a brief summary of related work. References to other related work can also be found in Chapter 6.

5.1. Normal Forms

The basic notion in BURS theory is that of a bottom-up normal form. Due to Proposition 2.14 we can ignore looping rewrite sequences.

Definition 5.1 *Let τ be a valid rewrite sequence without loops. τ is in **normal form** at ε if it is of the form $\tau_1 \cdots \tau_n \tau_0$, and*

(1) For all i, $1 \le i \le n$, all rewrite rule applications in τ_i are at positions that are descendants of i; and,

(2) There is no rewrite sequence τ', equivalent to τ, and of the form $\tau_1 \cdots \tau_n \tau'_0$, where τ'_0 is a permutation of τ_0 starting with a rewrite application of the form $<r, k \| p>$ for some k, $1 \le k \le n$, and some position p.

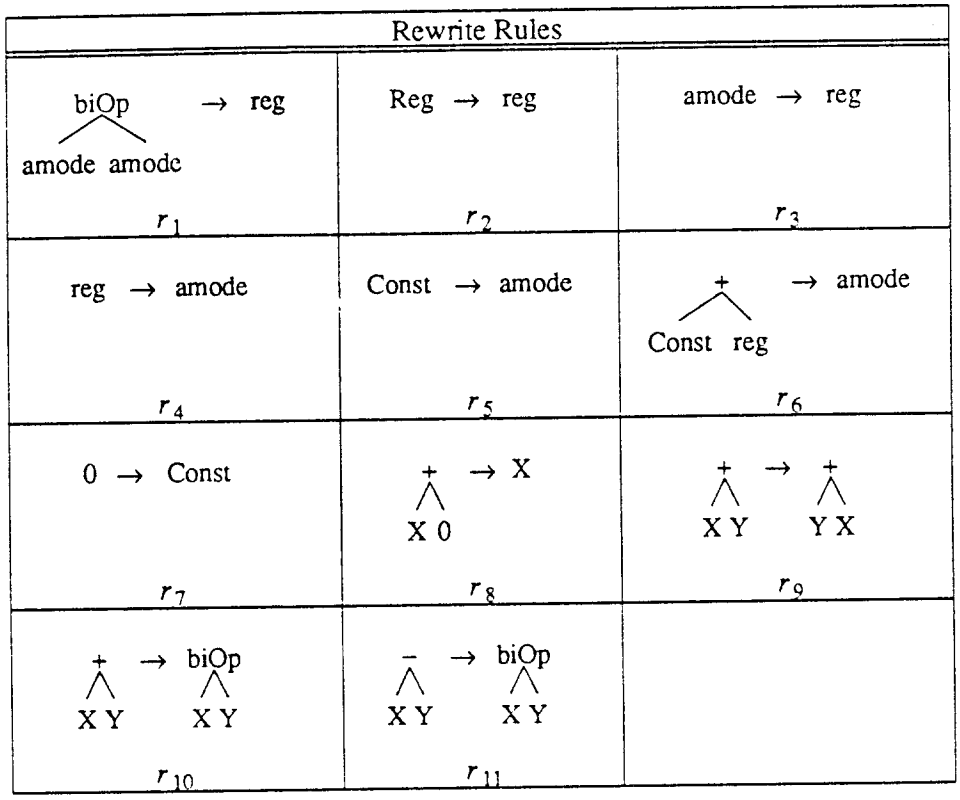
*τ is in **normal form** (everywhere) if it is in normal form at ε, and,*

(3) for $1 \le i \le n$, $(\tau_i)_{@i}$ is in normal form.

---

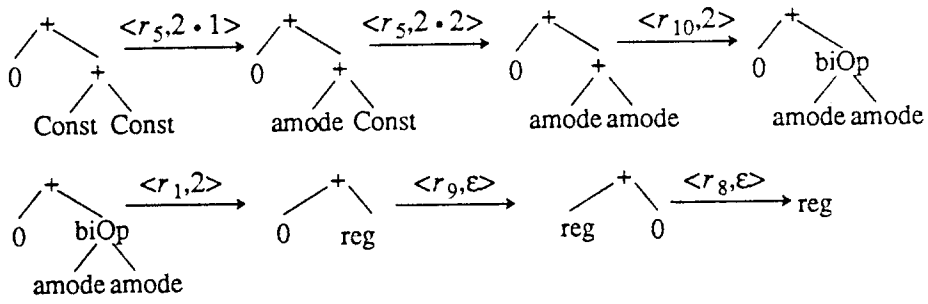
[16] Actually, just a subset of them; see Section 5.3.2 for details.

As an example, the rewrite sequence shown in Figure 5.2 is a valid rewrite sequence for the rewrite system of Figure 5.1. It is applicable to +(0, +(*Const*, *Const*)), and is in normal form.



Example of a Rewrite System

Figure 5.1



A Normal-Form Rewrite Sequence

Figure 5.2

Proposition 5.1 *Let R be a rewrite system and let ϕ be a valid non-looping rewrite sequence for R. Then there exists a permutation of ϕ in normal form.*

Proof We first construct a permutation of ϕ in normal form at ε.

We construct a series of rewrite sequences $\phi^0,\phi^1,\ldots$ such that for each ϕ^i in the series, ϕ^i is a permutation of ϕ, and

(*) ϕ^i is of the form ${\phi^i}_1 \cdots {\phi^i}_n {\phi^i}_0$ satisfying part (1) of Def. 5.1, and with $length({\phi^i}_1 \cdots {\phi^i}_n) = i$.

ϕ^0 is defined to be ϕ. ϕ^0 is clearly a permutation of ϕ, and ϕ^0 satisfies (*) with ${\phi^0}_j = \varepsilon$ for j, $1 \le j \le n$.

For $i \ge 0$, ϕ^{i+1} is constructed from ϕ^i. If ϕ^i is not in normal form at ε then, by condition (2) of Definition 5.2, there exists a ${\phi^i}_0' = <r,l/\!/p>\psi$, $1 \le l \le n$, which is a permutation of ${\phi^i}_0$. Define ${\phi^{i+1}}_j$ to be ${\phi^i}_j$ for all j, $1 \le j \le n$ and $j \ne l$, define ${\phi^{i+1}}_l$ to be ${\phi^i}_l <r,l/\!/p>$, and define ${\phi^{i+1}}_0$ to be ψ. The new ϕ^{i+1} is a permutation of ϕ and satisfies (*).

The series $\phi^0,\phi^1,\ldots$ cannot be infinite because for any ϕ^i, $i = length({\phi^i}_1 \cdots {\phi^i}_n) \le length(\phi)$. Hence there must exist a ϕ^k such that ϕ^k is in normal form at ε. To obtain a permutation of ϕ^k which is in normal form, apply the above construction recursively to ${\phi^k}_1, \ldots, {\phi^k}_n$. □

A rewrite sequence for an input tree in normal form assigns to each position in the tree a **local rewrite sequence**: the rewrite applications done at that position. Formally:

Definition 5.2 *Let τ be a normal form rewrite sequence of the form $\tau_1 \cdots \tau_n \tau_0$ that is applicable to a tree T. The **local rewrite sequence** assigned by τ to a position p in T is defined by $F(T,\tau,p)$, where*

(1) $F(T,\tau,\varepsilon)$ is τ_0, and

(2) if p is of the form $i/\!/q$, for some i, $1 \le i \le n$, and T is of the form $op(T_1,\ldots,T_n)$, then $F(T,\tau,p)$ is $F(T_i,\tau_i,q)$.

*The **local rewrite assignment** of τ and T is the function assigning to each position in T its local rewrite sequence.*

For example the local rewrite sequence assigned by the rewrite sequence of Figure 5.2 at the position 2 of the input tree is $<r_{10},2><r_1,2>$.

Proposition 5.2 *Let **R** be a rewrite system for **Op**, let T be a tree over **Op**, let p be a position in T, and let τ be a normal form rewrite sequence over **R** applicable to T.*

Let τ be a normal form rewrite sequence applicable to a tree T, and let p be a position in T. Let F be the local rewrite assignment of τ and T. Let F_{below} be the function assigning $F(q)$ to each position q in T that is a strict descendant of p, and the null rewrite sequence to the other positions in T. Similarly, let F_{around} be the function assigning the null rewrite sequence to all positions in T that are strict descendants of p, and $F(q)$ to all other positions q.

There is a normal form rewrite sequence τ_{below} whose local rewrite assignment is F_{below}. τ_{below} is applicable to $T_{@p}$. There is a normal form rewrite sequence τ_{around} whose local rewrite assignment is F_{around}. If $T' = \tau_{below}(T_{@p})$ and $\tau_0 = F(p)$, then τ_0 is applicable to $T_{@p \leftarrow T'}$, and τ_{around} is applicable to $\tau_0(T'')$, where T'' is $T_{@p \leftarrow T'}$.

*τ_{below} is said to be the rewrite sequence assigned by τ **below** p. τ_{around} is said to be the rewrite sequence assigned by τ **around** p.*

Proof Straightforward and left to the reader. □

Returning to Figure 5.2, the rewrite sequence assigned by the rewrite sequence of the figure below 2 is $<r_5,2 \cdot 2><r_5,2 \cdot 1>$; and that around 2 is $<r_9,\varepsilon><r_8,\varepsilon>$.

We can now define the BURS(k) property and the BURS class.

Definition 5.3 *Let k be a positive integer and let τ be a rewrite sequence in normal form applicable at some input tree T. τ is in **k-normal form** if it is in normal form and each of the local rewrite sequences assigned by τ to the nodes of T is of length at most k.*

*Let **R** be a rewrite system over **Op**, let L_{in} and L_{out} be sets of trees over **Op**, and let k be a positive integer. The triple $<R,L_{in},L_{out}>$ is said to satisfy the **BURS(k) property** if for any two trees $T \in L_{in}$ and $T' \in L_{out}$ and any sequence τ in **R**, with $\tau(T)=T'$, there is a permutation of τ which is in k-normal form. The class BURS is composed of those triples $<R,L_{in},L_{out}>$ satisfying the BURS(k) property for some positive integer k.*

In the rest of this chapter, L_{in} is frequently L_{Op}, the set of all trees over the operator set **Op**. Also, L_{out} is frequently the singleton $\{G\}$ for fixed-goal REACHABILITY problems, and, L_{Op} for variable-goal REACHABILITY problems. Since this dissertation solves many problems involving only these special types of triples, frequently we will use the phrase "a rewrite system **R** is in BURS" to mean that the triple $<R,L_{Op},\{G\}>$, or $<R,L_{Op},L_{Op}>$, depending on the context, is in BURS.

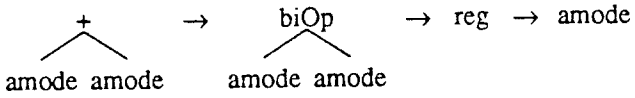
The rest of this chapter assumes only N-patterns because, except in some particular cases, rules with an X-pattern as input pattern inhibit membership in BURS.

Similarly, in general, rewrite systems with non-linear patterns are not in BURS. Non-linearity in the input patterns could be handled by "weakening" the definition of BURS so that instead of allowing only a "permutation" of the original rewrite sequence, duplication of subsequences would be allowed. This would make it possible to "move forward" rewrite applications done "late" in the original rewrite sequence by duplicating them in each of the subtrees matched early on. Unfortunately, this definition would not match the semantics of many applications. Non-linear output patterns are more difficult. The multiple copies of the subtrees produced by the non-linear output pattern can be modified by unrelated rewrite sequences.

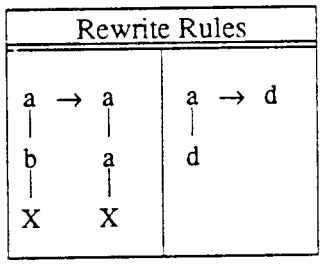
In addition, note that in some cases non-linear rewrite systems do not describe the desired semantics. For example, in a rewrite system describing a target machine (Chapter 6), a symbol like *register* will represent a computation value stored into some member of a register class. A

rewrite rule like $-(X,X) \rightarrow 0$ is normally intended to apply only to the original input tree, but if added to the rewrite system, it would also allow rewrite applications like $-(register,register) \rightarrow 0$, which is probably incorrect as there is no guarantee that both occurrences of *register* refer to the same register. Finally, rewrite rules with an input pattern equivalent to X will, in general, produce unbounded local rewrite sequences and lead to the failure of the BURS(k) property.

For all these reasons, the rewrite rules in this chapter are restricted to contain only linear N-patterns, and do not allow input patterns equivalent to X. For example, if R denotes the rewrite system of Figure 5.1, then $<R,L_{op},\{reg\}>$ satisfies the BURS(3) property. An example of a local rewrite sequence of length 3 is:



An example of a rewrite system that is not in BURS is the one of Figure 5.3, for $L_{out}=\{d\}$. The only normal form rewrite sequence from $a(b(b(...(b(c))...)))$ to d assigns to each non-root node an empty local rewrite sequence, and the rest of the sequence is assigned to the root.



Example of a Rewrite System not in BURS

Figure 5.3

Testing the BURS(k) property is easy when both L_{in} and L_{out} are L_{Op}. First we prove a very useful proposition:

Proposition 5.3 *Let R be a rewrite system over a set of operators Op, and let k be a positive integer. There is an algorithm that will generate every rewrite sequence τ that is a local rewrite sequence at some position p of some tree T in L_{Op} and such that length $(\tau) \leq k$.*

Before proving Proposition 5.3 we prove three auxiliary lemmas. The first lemma gives a strong necessary condition for local rewrite sequences.

Lemma 5.1 *Let R be a rewrite system over a set of operators Op. Let τ_0 be a non-empty local rewrite sequence for some position p of some tree T over Op. Then for every prefix ϕ of τ_0 of the form $\psi <r,q>$ and such that $\psi \neq \varepsilon$, we have*

(1) ψ *has a composition* $<\alpha_\psi \rightarrow \beta_\psi, p>$.

(2) $q = p/\!/s$ *, for some position* s *of* β_ψ *that does not correspond to a variable.*

(3) ϕ *has a composition* $<\alpha_\phi \rightarrow \beta_\phi, p>$.

Proof The proof proceeds by induction on the length of ψ.

We first note that (3) follows from (1) and (2) by Proposition 2.15; (3) is stated just as a convenience in the proof.

Basis. Assume that *length* $(\psi) = 1$. Then, by definition of local rewrite sequence, ψ is of the form $<\alpha \rightarrow \beta, p>$. (1) follows trivially. If (2) were false, then, since the input pattern of r is not equivalent to X, q would be of the form $p/\!/u/\!/t$ for some position u in β corresponding to a variable and some position t, and part (1) of Proposition 2.17 shows that an exchange could be performed between $<\alpha \rightarrow \beta, p>$ and $<r, q>$. If $p \not\equiv \varepsilon$ this provides a contradiction that τ_0 is a local rewrite sequence. If $p \equiv \varepsilon$ then $\alpha \equiv X$, which contradicts the hypothesis. Hence, (2) must be true.

Induction Step. Assume the body of the proposition true for all prefixes ψ' with *length* $(\psi') = k$, we want to prove the body of the proposition true for all prefixes ψ with *length* $(\psi) = k+1$. Let ψ be a prefix of τ_0 of length $k+1$. By definition of prefix, there is a prefix ψ' of τ_0 of length k such that $\psi = \psi' <r', q'>$. By part (3) of the inductive hypothesis, (1) is true for ψ; let $<\alpha_\psi \rightarrow \beta_\psi, p>$ be the composition of ψ. (2) follows by the same argument used in the basis, that is, if (2) were false, then q would be of the form $p/\!/u/\!/t$ for some variable position u in β and some position t, and part (1) of Proposition 2.17 would show that an exchange could be performed between $<\alpha_\psi \rightarrow \beta_\psi, p>$ and $<r, q>$, thus contradicting that τ_0 is a local rewrite sequence or that no rules have input pattern equivalent to X. □

A simple corollary to Lemma 5.1 is:

Corollary 5.1 *Let* τ_0 *be a local rewrite sequence at a position* p *of a tree* T. *Then* τ_0 *has a composition, and it is of the form* $<\alpha_{\tau_0} \rightarrow \beta_{\tau_0}, p>$.

The second lemma regards testing the equivalence of two rewrite sequences. If the two rewrite sequences have a composition, then their equivalence can be tested by a simple structural check.

Lemma 5.2 *Let* R *be a rewrite system over* Op. *Let* τ *and* τ' *be two rewrite sequences with compositions* $\alpha_\tau \rightarrow \beta_\tau$ *and* $\alpha_{\tau'} \rightarrow \beta_{\tau'}$. *Assume that* $Vars(\alpha_\tau) \cap Vars(\alpha_{\tau'}) = \varnothing$. *Then* τ *is equivalent to* τ' *if and only if there is a one-to-one function* f *from* $Vars(\alpha_\tau)$ *into* $Vars(\alpha_{\tau'})$ *such that* $f(\alpha) = \alpha'$ *and* $f(\beta) = \beta'$.

Proof Straightforward and left to the reader. □

The final lemma presents some properties regarding the validity of rewrite sequences in a particular form.

Lemma 5.3 *The following statements are true.*

(1) *Let* R *be a rewrite system over* Op. *Any prefix of a valid rewrite sequence over* R *is a valid rewrite sequence over* R.

(2) *Let* $<\alpha \rightarrow \beta, p>$ *be a rewrite application for some position* p, *and let* $<\alpha' \rightarrow \beta', q>$ *be a rewrite application with* $q = p/\!/s$, *for some position* s *in* β *that does not correspond to a variable in* β. *Without loss of generality assume that* $Vars(\beta) \cap Vars(\alpha') = \varnothing$. *Then,*

(2.1) *If* $<\alpha \rightarrow \beta, p><\alpha' \rightarrow \beta', q>$ *is valid, then* $\beta_{@s}$ *is unifiable with* α'.

(2.2) *If* $\beta_{@s}$ *is unifiable with* α' *then* $<\alpha \rightarrow \beta, p><\alpha' \rightarrow \beta', q>$ *is valid.*

Proof: (1) is straightforward.

(2.1) The unification assignment is constructed from the tree T provided by the validity of $<\alpha\rightarrow\beta,p>$, and the assignment required by $<\alpha'\rightarrow\beta',q>$ being applicable at the result of applying $<\alpha\rightarrow\beta,p>$ to T.

(2.2) If T is a tree at which $<\alpha\rightarrow\beta,p>$ is applicable, and σ is the assignment resulting from the match of α to T, (2.2) follows by using the unification assignment, T, and σ to construct the assignment for the validity of $<\alpha\rightarrow\beta,p><\alpha'\rightarrow\beta',q>$.

The details of all the proofs are left to the reader. □

Lemma 5.3 can be used to construct an algorithm determining whether a rewrite sequence τ_0 satisfying the conditions (1)-(3) indicated in Lemma 5.1 is valid or not. The algorithm tests the validity of all the prefixes of τ_0. If any prefix is not valid then, by part (1) of Lemma 5.3, τ_0 is not valid. The validity of a prefix $\phi=\psi<r,q>$ is tested by first finding the composition of ψ and then using part (2) of Lemma 5.3.

Now we return to our original proposition:

Proof of Proposition 5.3: By Lemma 5.1 we can generate a collection of rewrite sequences that will include all the local rewrite sequences of length no larger than k. For each one of these, by Lemma 5.3 we test whether the rewrite sequence is valid. Let τ be one such valid rewrite sequence. By construction it has a composition, let it be $\alpha_\tau\rightarrow\beta_\tau$. Let $S(\tau)$ be the set of those rewrite sequences that are obtained by reordering the rules in τ and which use positions of height no larger than the maximum of *height*(α_τ) and *height*(β_τ). $S(\tau)$ is a finite and constructible set, and the permutations of τ are those rewrite sequences in $S(\tau)$ which are equivalent to τ. The equivalence between a rewrite sequence σ in $S(\tau)$ and τ is equivalent to the conjunction of the following three conditions: (1) σ is applicable to α_τ (as a tree), (2) the application produces β_τ (as a tree), and (3) the application "uses" all the nodes of α_τ (i.e. if the nodes of α_τ are initially marked as being "unused" and are marked as "used" whenever their labels are used, then in no rewrite application in σ a variable matches a subtree containing "unused" nodes). τ is a local rewrite sequence for $<R,L_{Op},L_{Op}>$ if there is no σ in $S(\tau)$ equivalent to τ violating condition (2) of Definition 5.2. □

Given a rewrite system R, $<R,L_{Op},L_{Op}>$ satisfies the BURS(k) property if and only if there is no valid local rewrite sequence of length $k+1$. From Proposition 5.3, it follows:

Proposition 5.4 *Let R be a rewrite system over a set of operators Op, and let k be a positive integer. There is an algorithm that will determine whether $<R,L_{Op},L_{Op}>$ satisfies the BURS(k) property.*

A simple consequence of the notion of composition of a rewrite sequence and the definition of BURS(k) is:

Proposition 5.5 *Let R_1 be a rewrite system over Op, and let L_i and L_o be sets of trees over Op. Let $<R_1,L_i,L_o>$ satisfy the BURS(k) property for some positive integer k. Then there is a rewrite system R_2 such that $<R_2,L_i,L_o>$ satisfies the BURS(1) property, and such that R_2 implements the same transformation as R_1 and there is a one-to-one mapping between the normal form rewrite sequences in R_1 and in R_2.*

Proof R_2 is constructed from R_1 by adding the composition of all the local rewrite sequences in it. Since there are a finite number of local rewrite sequences, there are a finite number of new rules added. □

5.2. Some BURS Classes

There are significant classes of rewrite systems in BURS. Reduction rewrite systems are a class of rewrite systems that proceed by "reducing" a tree.

Definition 5.4 *A **reduction rewrite system**, **R**, over an operator set with arity, **Op**, is a collection of rewrite rules of one of two types:*

*(1) A **rename rule** is of the form* $op(X_1, \ldots, X_n) \rightarrow op'(X_{\pi(1)}, \ldots, X_{\pi(m)})$, *for some permutation* π *and some* m, *for* $m \leq n$.

*(2) A **reduction rule** is either of the form* $T \rightarrow op$, *for some tree* T *over* **Op**, *or* $op(X_1, \ldots, X_m) \rightarrow X_i$, *for some* $1 \leq i \leq m$.

The main property of reduction systems is:

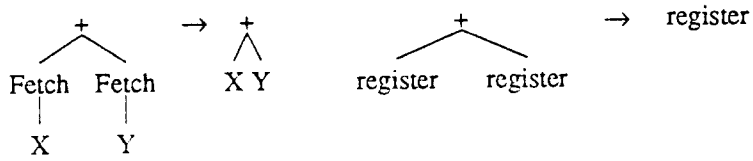
Proposition 5.6 *Let* R *be a reduction system over* **Op**, *then,* $\langle R, L_{Op}, L_{op} \rangle$ *is in BURS.*

Proof Let R be a reduction system. Any local rewrite sequence τ satisfying the characterization of Lemma 5.1 is of one of four forms: (i) $\tau = \alpha$, where α is a, possibly null, rewrite sequence containing only rename rules for nullary operators; (ii) $\tau = \alpha'\beta\alpha$, where α is as above, α' is a, possibly null, rewrite sequence containing only rename rules for operators of arity larger than 0, and β is an application of a reduction rule of the form $T \rightarrow op$; (iii) $\tau = \alpha'\beta'$, where α' is as above, and β' is a single reduction rule of the form $op(X_1, \ldots, X_m) \rightarrow X_i$; or (iv) α' where α' is as above. Let the arity of the root operator of the input pattern of the first rule in α' be n, and the number of different operators in R be m. Since local rewrite sequences have no loops, any candidate rewrite sequence is no longer than $n!+m$. Membership in BURS follows directly from that and Proposition 5.1. □

It is instructive to show how some particularly tempting extensions to Definition 5.4 are not in BURS.

Relaxing the definition of rename rules so that it is possible to rename a single operator in some lower context seems a reasonable extension since the only "context" employed is the lower one, and it seems plausible to collect all the desired information in a bottom-up phase and use it accordingly. The rewrite system containing the rules: $a(b(X)) \rightarrow aa(b(X))$, $b(b(X)) \rightarrow bb(b(X))$, $b(c) \rightarrow c$, $aa(c) \rightarrow d$, and $bb(c) \rightarrow c$ is a counterexample. This rewrite system behaves similarly to the one in Figure 5.3.

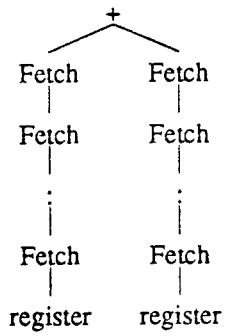
Another possible extension to reduction systems would be to allow lower context only in reductions. Again, this extension includes rewrite systems not in BURS. An example is shown in Figure 5.4, where +, *Fetch*, and *register* are the operators.



Second Example of a Rewrite System not in BURS

Figure 5.4

To show that it does not belong to BURS, take any positive integer k and consider the tree of height $k+3$



No normal form rewrite sequence yielding "register" has a local rewrite sequence at the root of this tree of length less than k.

5.3. State Characterization

We now follow the approach described in the introduction of this chapter. A normal form rewrite sequence defines a local rewrite assignment. Conversely, a local rewrite assignment uniquely determines one rewrite sequence (up to permutations). This leads to a first characterization of the interaction of a rewrite system on a subtree: the ***proto-state***. A proto-state can be used to solve variable-goal REACHABILITY; later notions have more restricted applicability but are leaner and allow faster implementations.

5.3.1. Proto-States

Definition 5.5 *Let R be a rewrite system over Op, and let $<R,L_{Op},L_{Op}>$ be in BURS. The proto-state associated with a tree T is the set of triples $<\tau,T_1,T_2>$ such that there is a normal form rewrite sequence π such that $\pi(T)=T_1$ and τ is a local rewrite sequence with $\tau(T_1)=T_2$. T_1 are the **input trees** and T_2 the **output trees** of the proto-state.*

Proto-states contain enough information to characterize all the rewrite sequences, thus satisfying (STATE-1), and can be computed bottom-up, satisfying (STATE-2). This follows directly from the argument used in the proof of Proposition 5.4, which shows that we can enumerate all the local rewrite sequences of a rewrite system in BURS. Hence:

Proposition 5.7 *Let R be a rewrite system over Op, and let $<R,L_{Op},L_{Op}>$ be in BURS. Let $St(T)$ denote the proto-state associated with tree T. There is a function f such that $St(op(T_1,\ldots,T_n))=f(op,St(T_1),\ldots,St(T_n))$.*

Variable-goal REACHABILITY between T and T' for a system R, can now be solved as follows:

(1) Compute all the proto-states of the input tree T as described above.

(2) Compare the goal tree T' against all the output trees in the local rewrite sequences of the proto-state associated with T. There will be a rewrite sequence in R from T into T' if and only if at least one of the comparisons succeeds. Set N, the current node, to the root, and set G, the current goal, to T'.

(3) Let $T=Op(T_1,\ldots,T_n)$. Select a local rewrite sequence τ_N from the proto-state of T, rewriting some input tree I into G. There must be one. By construction of the local rewrite sequences, I must have the form $Op(G_1,\ldots,G_n)$. Repeat this step recursively for all i, $1\le i\le n$, setting T to T_i, G to G_i, and N to the root of T_i.

(4) The above steps select, if it exists, a local rewrite assignment corresponding to a normal form rewrite sequence from T into T'. A postfix concatenation of the local rewrite sequences produces the desired rewrite sequence.

If R is in BURS, then each proto-state will be finite, and the above procedure is an algorithm for solving variable-goal REACHABILITY. Hence,

Proposition 5.8 *Let R be a rewrite system over Op, and let $<R, L_{Op}, L_{Op}>$ be in BURS. There is an algorithm to solve variable-goal REACHABILITY for R, L_{Op}, and L_{Op}.*

5.3.2. LR graphs

In practice, proto-states may contain either too many trees, or trees that are too large, or both, and the algorithm outlined in the previous section may become impractical. The notions of ***LR graphs*** and ***UI LR graphs*** alleviate these problems for fixed-goal REACHABILITY while still satisfying (STATE-1) and (STATE-2). LR graphs are studied in this section, while *UI LR graphs* are studied in the next section. Each such graph will correspond to a state.

As mentioned at the beginning of this chapter, the notion of LR graphs can be obtained from that of proto-states by changing what sequences are "interesting" and modifying the "information" that is encoded in the state. LR graphs consider a special type of normal form rewrite sequences that may transform the input tree into the (fixed) goal tree.

Definition 5.6 *Let R be a rewrite system over Op, and let τ be a normal form rewrite sequence over R applicable to some tree T over Op. For every position p in T, let $<\alpha_p \rightarrow \beta_p, p>$ be the composition of the local rewrite sequence associated with p by τ. We say that τ is efficient if for every position p in T, for any position q in α_p such that there is a variable X at position q in α_p that does not appear in β_p (thus making $\alpha_p \rightarrow \beta_p$ an erasing rewrite rule), τ assigns an empty local rewrite sequence to any position that is a descendant of $p/\!/q$.*

Intuitively an efficient rewrite sequence is one without "obviously inefficient" rewrite applications. The following proposition shows that we can consider only efficient rewrite sequences for solving fixed-goal REACHABILITY:

Proposition 5.9 *Let R be a rewrite system over Op, let T be a tree over Op, and let G be a nullary symbol in Op. There is a rewrite sequence, τ, in R, such that $\tau(T)=G$ if and only if there is an efficient rewrite sequence τ' in R such that $\tau'(T)=G$.*

Proof Let τ be a rewrite sequence in R such that $\tau(T)=G$. First obtain from τ an equivalent rewrite sequence τ_{nf} in normal form. τ_{nf} assigns to each position in T a local rewrite sequence. Now perform a postorder traversal of the nodes in T removing all the local rewrite sequences that violate the definition of efficient. The resulting set of local rewrite sequences can be composed to obtain the desired rewrite sequence τ' since the only local rewrite sequences removed are those whose effect will be discarded by the application of an erasing rewrite rule. Note that τ' is not necessarily equivalent to τ since τ' applies to a larger set of trees since the rewrite applications in τ that were removed need not be applicable for τ' to be applicable.

The reverse implication is immediate. □

Efficient rewrite sequences also satisfy:

Proposition 5.10 *Let R be a rewrite system over Op, let T be a tree over Op, and let G be a nullary symbol in Op. Let τ be an efficient rewrite sequence in R such that $\tau(T)=G$. Then τ has a composition.*

Proof The proof is left to the reader. It uses Proposition 2.15 and the restriction that the goal G is a tree with a single node. □

Now we continue defining the notion of an LR graph, a state. Intuitively, we will encode only enough information in the state associated with a node N so that we can decide what local rewrite sequences can be applied at any node that is an ancestor of N. To determine this, we start by considering the composition of each local rewrite sequence (Definition 2.24). The input pattern of the composition encodes how much information is needed to determine that the local rewrite sequence can be applied. Since the information (that is, the patterns) available at a node is collected by the descendants of the node, the complete set of patterns that have to be encoded into the states corresponds to some type of closure of the set of input patterns of the local rewrite sequences under the application of the local rewrite sequences themselves. The **extended pattern set** is a set of patterns that satisfies the above requirements.

The notion of extended pattern set depends on the rewrite system $\boldsymbol{R}$ and the goal G. This notion has a technical problem similar to the one present in the definition of match set (Def. 3.4): we want to collect an interesting set of patterns, but we don't want to have two equivalent patterns in the collection. The solution used in Def. 3.4 was to draw patterns only from the "canonical set" Π_F. Unfortunately, here we don't readily have such a canonical set. (Actually, the extended pattern set will become a canonical set for further constructions). This makes the definition more cumbersome than desired.

Definition 5.7 *If τ is a local rewrite sequence with m rewrite applications, and composition $\alpha_\tau \rightarrow \beta_\tau$, let $pre(\tau,i)$ denote, for $0 \leq i \leq m$, the i–th prefix of τ (thus $pre(\tau,0)(\alpha_\tau) = \alpha_\tau$, and $pre(\tau,m)(\alpha_\tau) = \beta_\tau$).*

*Let **R** be a rewrite system over **Op** and let G be a nullary operator in **Op**. We define three sets of patterns, $I'_{R,G}$, $O'_{R,G}$, and $M'_{R,G}$, over **Op** as the minimum set satisfying the rules (1), (2), and (3) defined below. The **extended pattern set** of **R** and G, $EF_{R,G}$ is a $\equiv$-reduction (Def. 3.1) of the union of $I'_{R,G}$, $O'_{R,G}$, and $M'_{R,G}$. The set of **input patterns**, $I_{R,G}$, is the subset of $EF_{R,G}$ that is equivalent to $I'_{R,G}$; the set of **output patterns**, $O_{R,G}$, is the subset of $EF_{R,G}$ that is equivalent to $O'_{R,G}$; and the set of of **intermediate patterns**, $M_{R,G}$, is the subset of $EF_{R,G}$ that is equivalent to $M'_{R,G}$.*

(1) *G belongs to $O'_{R,G}$.*

(2) *Let τ be a local rewrite sequence with composition $\alpha_\tau \rightarrow \beta_\tau$. Let ρ be a pattern in $O'_{R,G}$ non-equivalent to X, and let ρ' be a pattern equivalent to ρ and variable-disjoint from β_τ. If β_τ unifies with ρ', let σ be their most general unifier. Let $\gamma(j)$ denote $pre(\tau,j)(\sigma(\alpha_\tau))$. Then,*

(2.i) *If there is no pattern in $I'_{R,G}$ equivalent to $\gamma(0)$, $\gamma(0)$ is added to $I'_{R,G}$;*

(2.ii) *For every j, $0<j<m$, if there is no pattern in $M'_{R,G}$ equivalent to $\gamma(j)$, $\gamma(j)$ is added to $M'_{R,G}$; and*

(2.iii) *If there is no pattern in $O'_{R,G}$ equivalent to $\gamma(m)$, $\gamma(m)$ is added to $O'_{R,G}$.*

(3) *For every pattern ρ in $I'_{R,G}$, and every child ρ' of ρ, if there is no pattern in $O'_{R,G}$ equivalent to ρ', ρ' is added to $O'_{R,G}$.*

The intention of the extended pattern set is to reflect all the "situations of interest" that may arise in normal form rewrite sequences from trees in L_{in} into G. Currently L_{in} is restricted to be L_{Op}, but hopefully future research will extend the above construction algorithm to more general input and output languages. In the above definition, $I_{R,G}$ are the patterns of interest at the beginning of local rewrite sequences, $M_{R,G}$ are those at the middle, and $O_{R,G}$ those at the end of the local rewrite sequences. The patterns in $O_{R,G}$ are those used to construct members of $I_{R,G}$ "higher up" into the tree. The construction iteration in Definition 5.7 stops at X because we will characterize only efficient rewrite sequences.

Definition 5.7 provides a constructive mechanism to compute the extended pattern set because, by Proposition 5.3 we know how to generate all the local rewrite sequences of normal form rewrite sequences applicable at trees in L_{Op}. Figure 5.5 shows the results of the constructive algorithm described in Definition 5.7 in the case where R is defined to be the rewrite system of Figure 5.1, and G is the pattern *reg*. The figure presents a table with two columns of patterns, corresponding to the input and output nodes.

$O_{R,\{reg\}}$	$I_{R,\{reg\}}$	
reg	*Reg*	*amode*
	+(*Const*, *reg*)	+(*reg*, *Const*)
amode	*Const*	*reg*
Const	*biOp* (*amode*, *amode*)	+(*amode*, *amode*)	−(*amode*, *amode*)
0	+(*reg*, 0)	+(*amode*, 0)	+(*Const*, 0)
	+(0, *reg*)	+(0, *amode*)	+(0, *Const*)
	+(0, 0)	0	

An Extended Pattern Set

Figure 5.5

Note that $+(amode, amode) \in I_{R,reg}$ by an application of step (2) in Definition 5.7, with $\rho = reg$ and $\tau = \langle r_{10}, \varepsilon \rangle \langle r_1, \varepsilon \rangle$, and that $+(amode, 0) \in I_{R,reg}$ also by an application of step (2) with $\rho = reg$ and $\tau = \langle r_8, \varepsilon \rangle \langle r_3, \varepsilon \rangle$.

The desired properties of an extended pattern set can be formalized as (1), (2), and (3) in the proposition below. We currently know how to compute sets with these properties only for the case when the input set L_{in} is L_{Op}.

Proposition 5.11 *Let **R** be a rewrite system over **Op**, let L_{in} be a set of trees over **Op**, and let G be a nullary operator in **Op**. Let **I**, **M**, and **O** be three sets of patterns over **Op**, and let (1), (2), and (3) be the following three statements:*

(1) A pattern ρ is equivalent to some pattern $\bar{\rho} \in I$ if and only if there is a tree $A \in L_{in}$, a position p in A, and an efficient normal form rewrite sequence τ applicable to A such that, if π is the rewrite sequence assigned by τ below p, τ_0 is the local rewrite sequence assigned by τ to p, ϕ is the rewrite sequence assigned by τ around p, and $\zeta = \tau_0 // \phi$, then:

(1.i) $\tau(A) = G$,

(1.ii) ρ *matches at* $\pi(A_{@p})$,

(1.iii) *Let* $\langle \alpha_\zeta \rightarrow \beta_\zeta, \varepsilon \rangle$ *be the composition of* ζ. *Then* $\{p // s \mid s$ *is a position in* $\rho\} = \{q \mid q$ *is a position in* α_ζ *which is a (maybe non-strict) descendant of* $p\}$.

(2) A pattern ρ is equivalent to some pattern $\bar{\rho} \in M$ if and only if the same conditions in (1) apply except for (1.ii) which is replaced by

(2.ii) ρ *matches at* $pre(\tau, k)(\pi(A_{@p}))$, *for some* k *with* $1 \le k \le length(\tau)$,

(3) *A pattern* ρ *is equivalent to some pattern* $\bar{\rho} \in O$ *if and only if the same conditions in (1) apply except for (1.ii) which is replaced by*

(3.ii) ρ *matches at* $\tau(\pi(A_{@p}))$.

Then, if $L_{in} = L_{Op}$, $I_{R,G}$, $M_{R,G}$, *and* $O_{R,G}$ *are as defined in Definition 5.7, (1), (2), and (3) are satisfied by* $I = I_{R,G}$, $M = M_{R,G}$, *and* $O = O_{R,G}$.

Proof We only prove the property for $I_{R,G}$, the proofs for the other sets are similar

Part I: If $\rho \equiv \bar{\rho} \in I_{R,G}$ then ρ satisfies (1). We show that repeated applications of the generative steps in Definition 5.7 preserve condition (1).

Let ρ' be equivalent to some $\bar{\rho}' \in I_{R,G}$, and by induction hypothesis, let A', p', τ', π', τ'_0, ϕ', and ζ' be the values satisfying condition (1). In the application of step (2) of Definition 5.7, let ρ_0 be the s-th child of ρ', let τ_0 be the local rewrite sequence with composition $<\alpha_{\tau_0} \rightarrow \beta_{\tau_0}, p>$, and let σ be the most general unifier of β_{τ_0} and ρ_0. Let $\rho = \sigma(\alpha_{\tau_0})$. Let $X_1, \ldots, x_m$ be the variables in ρ, and let *op* be any nullary operator in R distinct from G. Let $B = \rho_{X_1 \leftarrow op \cdots X_m \leftarrow op}$. We want to show that ρ satisfies (1) with $p = p' /\!/ s$, $A = A'_{@p \leftarrow B}$, π the empty rewrite sequence, τ_0 as given, ϕ the rewrite sequence induced by τ' around ρ, and $\zeta = \tau_0 /\!/ \phi$.

(1.i) $\tau(A) = G$.

$\tau(A) = \tau(A'_{@p \leftarrow B})$ (by definition)
$= \phi(\tau_0(A'_{@p \leftarrow B}))$ (since π is empty)
$= \phi(A'_{@p \leftarrow \tau_0(B)})$ (by definition of local rewrite sequence)
$= \phi(A'_{@p \leftarrow \rho_0})$ (by definition of most general unifier)
$= \phi'(\tau'_0(A'_{@p' \leftarrow \rho'}))$ (by construction)
$= G$ (by inductive hypothesis).

(1.ii) ρ matches at $\pi(A_{@p})$. $\pi(A_{@p}) = A_{@p}$ (since π is empty) $= B$ (by construction). ρ matches at B trivially.

(1.iii) Let $<\alpha_\zeta \rightarrow \beta_\zeta, \varepsilon>$ be the composition of ζ. Then $\{p /\!/ s \mid s$ is a position in $\rho\} = \{q \mid q$ is a position in α_ζ below $p\}$. (Note that ζ has a composition because τ', being an efficient rewrite sequence has one.) Since all patterns are linear and ρ_0 and β_{τ_0} have disjoint variables, a position in ρ corresponds to either a position in α_{τ_0}, or to one in ρ_0.

Part II: If ρ satisfies (1), then $\rho \in I_{R,G}$. Let A, p, τ, π, τ_0, ϕ, and ζ be the values satisfying condition (1), and let ρ be a pattern matching at $\pi(A_{@p})$. Let the position p be $p_1 \bullet \cdots \bullet p_k$, and for any position q let $\zeta(q)$ denote the local rewrite sequence induced by ζ at position q. As elsewhere, if τ is a local rewrite sequence, its composition will be denoted by $\alpha_\tau \rightarrow \beta_\tau$. We construct a sequence of patterns $\rho^0{}_O, \rho^0{}_I, \ldots, \rho^k{}_O, \rho^k{}_I$ with $\rho^j{}_I$ in $I_{R,G}$, and $\rho^j{}_O$ in $O_{R,G}$, $\rho^k{}_I = \rho$, and such that the sequence corresponds to successive generating steps in Definition 5.7. The sequence is obtained from ζ as follows.

$\rho^0{}_O$ equals G. and $\rho^0{}_I$ equals $\alpha_{\zeta(\varepsilon)}$

For all j in $1 \le j \le k$, $\rho^j{}_O$ equals the p_j-th child of $\rho^{j-1}{}_I$, and, $\rho^j{}_I$ equals $\sigma(\alpha_{\zeta(p_1 \bullet \cdots \bullet p_j)})$, where σ is the most general unifier of $\rho^j{}_O$ and $\beta_{\zeta(p_1 \bullet \cdots p_j)}$.

Note that, since τ is efficient, X is not any of the patterns present in the sequence. □

The extended pattern set can now be used to define the LR graphs.

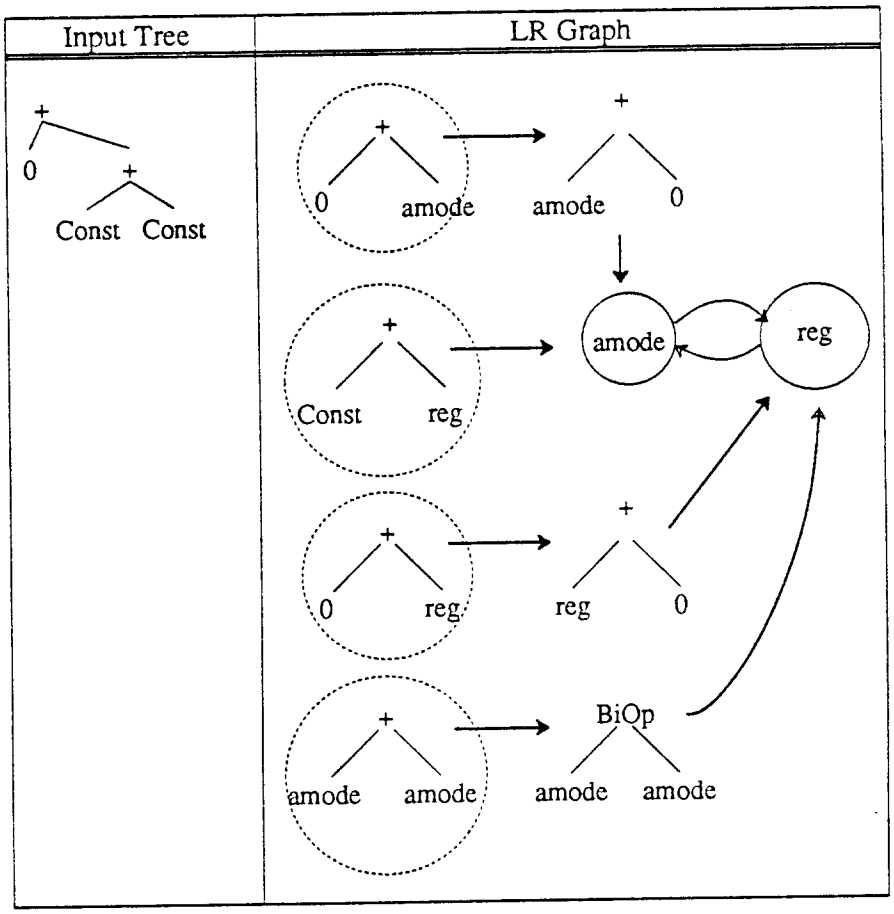
Definition 5.8 *Let* R *be a rewrite system over* Op, *and let* G *be a nullary operator in* Op. *Let* $<R, L_{Op}, G>$ *be in BURS. If* τ *is a local rewrite sequence with* m *rewrite applications, let pre*(τ, i) *denote, for* $0 \le i \le m$, *the prefix subsequence of* τ *of length* i. *The LR graph associated with a tree* T *is a graph* (V, E) *with labeled edges together with a distinguished set of nodes* S, *defined as follows.*

S is the set of patterns in $I_{R,G}$ that match at $\pi(T)$ for some normal form rewrite sequence π with no rewrite application at the root.

V contains all patterns in S, and any patterns in $M_{R,G}$ or in $O_{R,G}$ equivalent to $pre(\tau,i)(\rho)$ for some local rewrite sequence τ of length m, some ρ in S, and some i, $0<i\leq m$.

There is an edge in E from a pattern ρ_1 to another pattern ρ_2 both in V if there exists a rewrite rule r in R such that $r(\rho_1)\equiv\rho_2$

Figure 5.6 is the LR graph for +(0,+(*Const*,*Const*)), using the bottom-up rewrite system of Figure 5.1, and with goal {*reg*}. The extended pattern set on which the LR graph is based is the one shown in Figure 5.5. Input nodes are shown circled using dotted lines, and output nodes with solid lines. Note that the input nodes together with *R* and *G* uniquely determine the LR graph; this is why the notion of an LR graph does not distinguish the output nodes explicitly.



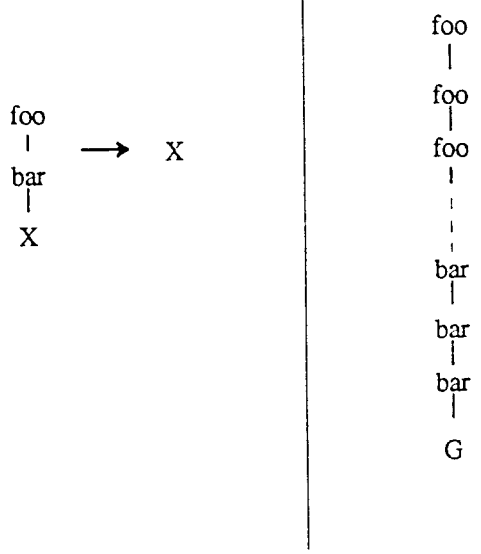
Example of an LR Graph

Figure 5.6

If the rewrite system were extended with an additional rewrite rule $Mul(X,0) \rightarrow 0$, $Mul(X,0)$ would become a member of $I_{R,G}$, and the pattern X would become a member of $O_{R,G}$. Note that we do not use X to generate new members in $EF_{R,G}$, otherwise all the nodes in the LR graph would be marked as output nodes to reflect the fact that any result obtained by rewriting can be discarded later using an application of the rewrite rule $Mul(X,0) \rightarrow 0$. The difference between using X or not corresponds to the difference between tracking all normal forms or only the efficient ones.

The notion of LR graphs is only practical for fixed-goal REACHABILITY in the case when $EF_{R,G}$ is finite.

Proposition 5.12 *There exist* $<R,L_{Op},\{G\}>$ *in BURS with unbounded* $EF_{R,G}$.
Proof By example. The left part of Figure 5.7 shows the only rule of a rewrite system. The right part shows a tree representative of a class where $EF_{R,G}$ is unbounded. □



Unbounded Extended Pattern Set

Figure 5.7

The unbound-ness of EF_R is not a result of a poor definition of extended pattern sets. For this example, no finite set of patterns produces LR graphs that can be used to reconstruct the rewrite sequences using the type of algorithm outlined above for proto-states. This observation motivates the following definition:

Definition 5.9 *Let R be a rewrite system over Op, let G be a nullary operator in R, and let k be a positive integer.* $\langle R, L_{Op}, \{G\}\rangle$ *is said to satisfy the finite BURS(k) property if it satisfies the BURS(k) property and* $EF_{R,G}$ *is a finite set. The class finite BURS is composed of those triples* $\langle R, L_{in}, \{G\}\rangle$ *satisfying the finite BURS(k) property for some positive integer k.*

As in the case of BURS(k) and BURS, frequently the input and output sets of the triple will be defaulted to L_{Op} and $\{G\}$ (or L_{Op}, in variable goal REACHABILITY) and omitted.

The rewrite systems describing the applications studied in this dissertation are finite BURS (Chapter 6, 7). In particular Proposition 5.6 can be strengthened as follows:

Proposition 5.13 *Let R be a reduction system over Op. Then,* $\langle R, L_{Op}, L_{Op}\rangle$ *is in finite BURS.*
Proof As in Proposition 5.6, a general local rewrite sequence has the form $\alpha\beta\alpha'$, where α and α' are sequences of rename rules and β is a single reduction rule. The number of patterns in the extended pattern set can be bound by bounding the height of the input patterns. If K is the height of the tallest pattern in R, then the height of any input pattern is at most K; this is direct if the reduction rule used in β is of the form $\rho \rightarrow op$, and follows by induction if the reduction rule is of the form $op(X_1, \ldots, X_n) \rightarrow X_i$. □

A simple consequence of this definition is:

Proposition 5.14 *Let R be a rewrite system over Op, and let G be a nullary operator in Op. Let* $\langle R, L_{Op}, \{G\}\rangle$ *be finite BURS. Then the set of all the LR graphs that are LR graphs of some tree T over Op is finite.*

Like proto-states, LR graphs satisfy (STATE-2), that is, they can be computed bottom-up:

Proposition 5.15 *Let R be a rewrite system over Op, let G be a nullary operator in Op, and let* $\langle R, L_{Op}, \{G\}\rangle$ *be in finite BURS. Let* $G(T)$ *denote the LR graph associated with tree T. There is a function f such that* $G(op(T_1, \ldots, T_n)) = f(op, G(T_1), \ldots, G(T_n))$.
Proof Since in finite BURS the input nodes uniquely characterize the LR graph, the proof will follow from showing how to compute the set of input nodes I of the parent given the set of output nodes, O_i for $1 \le i \le n$, of the children. There are two cases to consider. The first case is the pattern X. X is equivalent to a pattern in I if X is equivalent to some pattern in $EF_{R,G}$.

The second case is a pattern ρ distinct from X. ρ must be of the form $op(\rho_1, \ldots, \rho_n)$. For each i in $1 \le i \le n$, there is a normal form rewrite sequence π_i such that ρ_i matches at $\pi_i(T_i)$. By part (3) of Definition 5.7, ρ_i is equivalent to some pattern in $O_{R,G}$. Let τ_i be the local rewrite sequence of π_i at T_i. If $\alpha_{\tau_i} \rightarrow \beta_{\tau_i}$ is the composition of τ_i, then ρ_i matches at β_{τ_i}, and β_{τ_i} is equivalent to an output node in $G(T_i)$, as desired. □

Given a rewrite system R over Op and a nullary operator G in Op, it is semi-decidable whether $\langle R, L_{Op}, \{G\}\rangle$ satisfies the finite BURS(k) property. It is an open problem whether there is a decision procedure.

Corollary 5.2 *Let k be a natural number. Let R be a rewrite system over Op, and let G be a nullary operator in Op. It is semi-decidable if* $\langle R, L_{Op}, \{G\}\rangle$ *satisfies the finite BURS(k) property.*
Proof First compute the local rewrites sequences of R, deciding in the process if R satisfies the BURS(k) property. Then compute the extended pattern set for R. If $\langle R, L_{Op}, \{G\}\rangle$ is in finite BURS, then, by Proposition 5.15, the LR graphs can be computed using a bottom-up tree automaton. Try to construct this B-fsa for the LR graphs. The triple $\langle R, L_{Op}, \{G\}\rangle$ satisfies the finite BURS(k) property if and only if the construction algorithm completes. □

LR graphs also satisfy (STATE-1):

Proposition 5.16 *Let R be a rewrite system over Op, and let G be a nullary operator in Op, such that* $\langle R, L_{in}, \{G\}\rangle$ *is in BURS. Let (1) and (2) be the statements described below. If* L_{in} *is* L_{Op} *then (1) and (2) are true.*

(1) For every tree A in L_{in}*, every position p in A, and every efficient normal form rewrite sequence* τ *with* $\tau(A)=G$ *with local rewrite sequence* τ_0 *at p of length m, let* π *be the normal form rewrite sequence assigned by* τ *below p. There is a path in the LR graph of* $A_{@p}$ *of length m and such that the j-th pattern in the path matches* $pre(\tau_0, j)(\pi(A_{@p}))$ *for* $0 \le j \le m$.

(2) For every tree T, and for every non-looping path $\rho_0 \cdots \rho_m$ *from an input node to an output node in the LR graph for T there is a tree* $A \in L_{in}$*, a position p in A with* $A_{@p} = T$*, and an efficient normal form rewrite sequence* τ *with* $\tau(A)=G$*, local rewrite sequence* τ_0 *at p, and with normal form rewrite sequence* π *assigned below p, such that* τ_0 *has length m and for* $0 \le j \le m$, ρ_j *matches at* $pre(\tau, j)(\pi(A_{@p}))$

Proof Both parts follow from Proposition 5.11 and the definition of LR graph. As in Proposition 5.11 this proposition could be simplified substantially since L_{in} is L_{Op}. □

LR graphs contain enough information for solving both TERMINATION and CONFLUENCE (recall the definitions of Section 2.3). A rewrite system is not confluent if there is an LR graph for which there are two nodes (patterns) that cannot reach a common node (pattern). The existence of such an LR graph implies the existence of an input tree rewriting into two trees that cannot be rewritten into a common tree. Thus, testing for CONFLUENCE can be done by testing the above property in all the LR graphs. In the most straightforward approach, this requires a finite number of LR graphs for the situation in which the output set is L_{Op}. TERMINATION is similar, but the property to test is the existence of a loop in the LR graph. Again, it depends on having a finite number of LR graphs.

5.4. Fixed-Goal Reachability and UI LR graphs

The notion of LR graphs leads to a practical algorithm for solving REACHABILITY when L_{out} is a finite set. The algorithm is given in Figure 5.8. The algorithm is non-deterministic because of the step of the line marked (1): "*let* τ be a path in $G(T_{in})$ ending in T_{out} ...". It is clear, from the properties of LR graphs, that any such path will provide an answer. See Section 5.6.1 below for some general comments on a possible implementation of this algorithm.

```
function reachability(T_in , T_out )
    let <R,L_Op ,L_out > be in
    BURS.
    let G(t) be the LR graph of tree t;
    call compute–LR–graphs(T_in );
    comment compute the LR graphs of R,L_out in T_in ;
    if T_o is a node in G(T)
        result = find–sequence(T_in ,T_out );
        emit result;
        return (true);
    return (false);

procedure compute–LR–graphs(T);
    let T be op(T_1, . . . , T_n );
    for each i ∈ [1..n] do
        call compute–LR–graphs(T_i );
    G(T)=f (op ,G(T_1), . . . , G(T_n ));      /* (2) */
    assert f exists by Proposition 5.15

function find–sequence(T_in ,ρ) returns result;
    let τ be a path in G(T_in ) ending in T_out and starting at an input node,
      where ρ matches at T_out ;          /* (1) */
    assert there is at least one such path by Proposition 5.16
    let op(T'_1, . . . , T'_m) be the input node of τ;
    let op(T_1, . . . , T_m)=T_in ;
    result_1 = find–sequence(T_1,T'_1)
    ...
    result_m = find–sequence(T_m,T'_m)
    result = result_1// · · · result_m//τ;
    return (result);
```

Algorithm for Fixed Goal Reachability

Figure 5.8

Definition 5.10 *A* **restriction** *of an LR graph is a subgraph G of that graph such that G contains all the output patterns of the original graph, G contains at least one input pattern, and, for every output pattern in G there is at least one path in G starting at an input pattern in G and reaching the output pattern.*

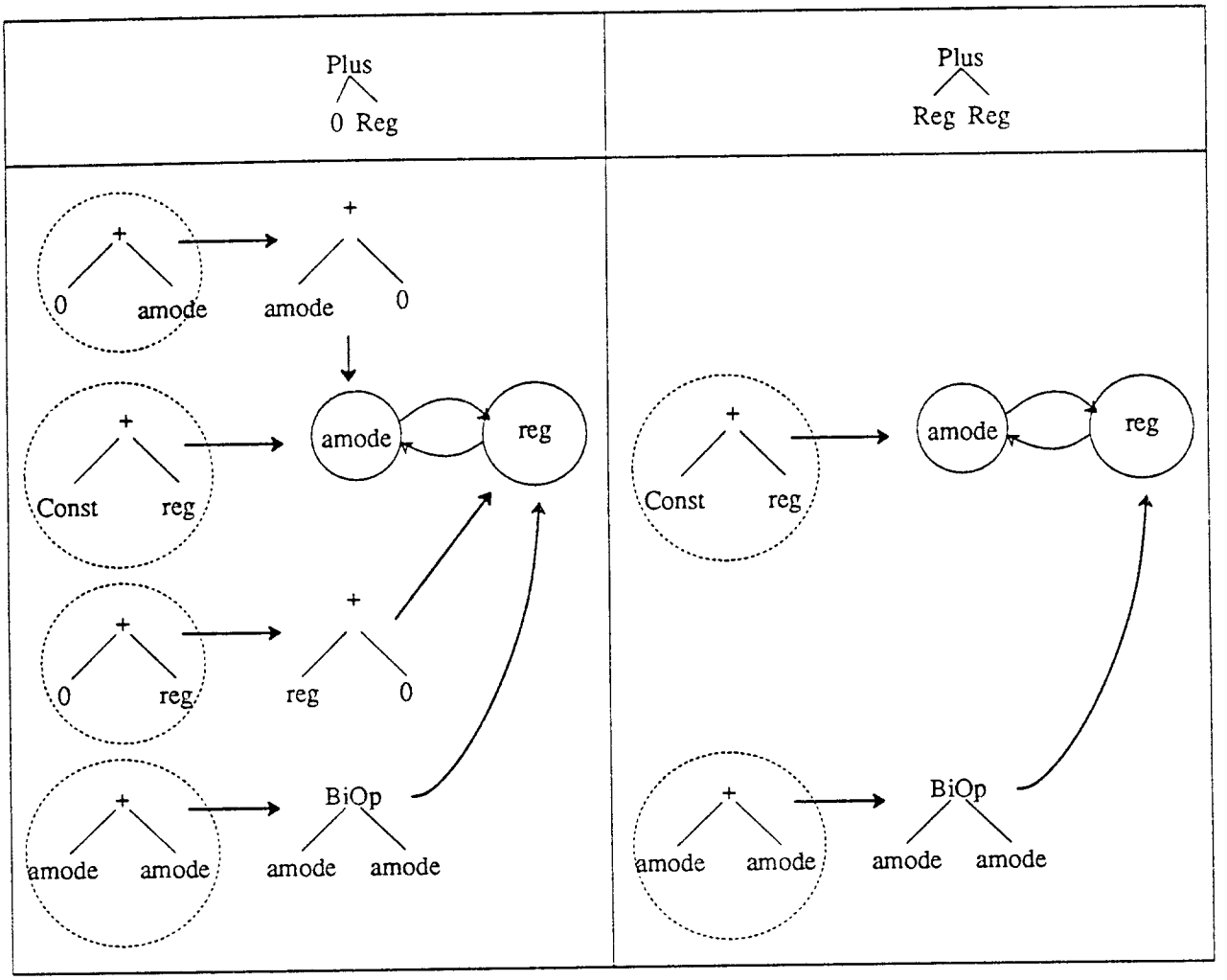
For any output tree the restricted graph will be able to provide a local rewrite sequence τ (line marked (1) in Figure 5.8). Thus, the replacement does not affect solving fixed-goal REACHABILITY. The strongest restriction on a graph is the "uniquely invertible" LR graph.

Definition 5.11 *A* ***uniquely invertible LR graph*** *(UI LR graph) is a restriction of an LR graph in which each node has, at most, only one edge reaching it.*

For an LR graph there may be several UI LR graphs. The replacement of an LR graph by a UI LR graph corresponds to throwing away many possible rewrite sequences, but nothing is lost for solving REACHABILITY.

Two LR graphs may have a common UI LR graph. For example, in Figure 5.9, the LR graph on the left has as an UI LR graph the LR graph on the right. The top row shows the corresponding input trees. Thus, since states correspond to LR graphs, one way to reduce the number of different states that are needed to solve REACHABILITY is to solve the following problem:

Definition 5.12 *Given a rewrite system **R** over **Op**, and a nullary operator G in **Op**, with* $<R, L_{Op}, \{G\}>$ *in finite BURS, the MINIMUM UI LR GRAPH problem consists of assigning to each LR graph a valid UI LR graph such that the number of UI LR graphs used is minimum.*



UI LR graphs

Figure 5.9

Chapter 8 uses this technique in one application. Unfortunately MINIMUM UI LR GRAPH is NP-complete.

Proposition 5.17 *Let R be a rewrite system over Op, and let G be a nullary operator in Op. Let* $<R,L_{Op},\{G\}>$ *be in finite BURS. MINIMUM UI LR GRAPH is NP-complete.*

Proof by reduction. The NP-complete problem to use is MINIMUM COVER [GaJ80]: Given a collection C of subsets of a finite set S, and a positive integer $K \leq |C|$, determine whether C contains a subset C' with $|C'| \leq K$ such that every element of S belongs to at least one member of C'.

MINIMUM COVER is equivalent to the following graph problem: given a bipartite graph with vertices $A \oplus B$, find the minimum number of nodes in B such that every node in A is connected

to at least one node in B. The graph problem, in turn, models exactly the situation present in MINIMUM UI LR GRAPH: the A nodes of the bipartite graph represent the LR graphs, the B nodes the UI LR graphs, and the edges the fact that one UI LR graph is a valid representative of an LR graph.

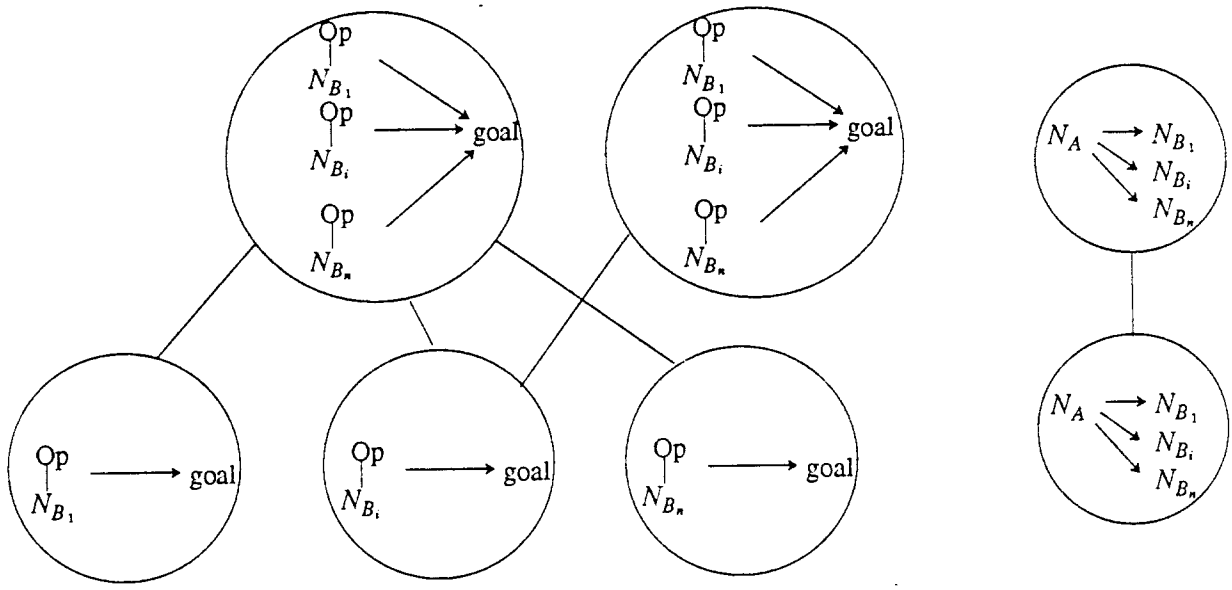
To complete the reduction it is only necessary to show that for any bipartite graph we can construct a rewrite system modeled by that bipartite graph. Let A and B be as above. The rewrite system is defined as follows:

(1) For each node B_i in B, there is an nullary symbol N_{B_i}. For each node A_i in A, there is an nullary symbol N_{A_i}. There is a nullary symbol *goal*, and a unary symbol *Op*, different from all the other ones.

(2) For each node B_i in B, there is a rewrite $Op\,(N_{B_i}) \rightarrow goal$.

(3) For each node A_i in A with n incoming edges from nodes $B_1,\ldots, B_n$, there are n rewrites $N_{A_i} \rightarrow N_{B_1},\ldots, N_{A_i} \rightarrow N_{B_n}$.

With this definitions, the bipartite graph describing the relationship between LR graphs and UI LR graphs can be split into two parts. The first part is the desired bipartite graph as sketched at the left of Figure 5.10, the second part is a very simple bipartite graph that can be solved immediately. □



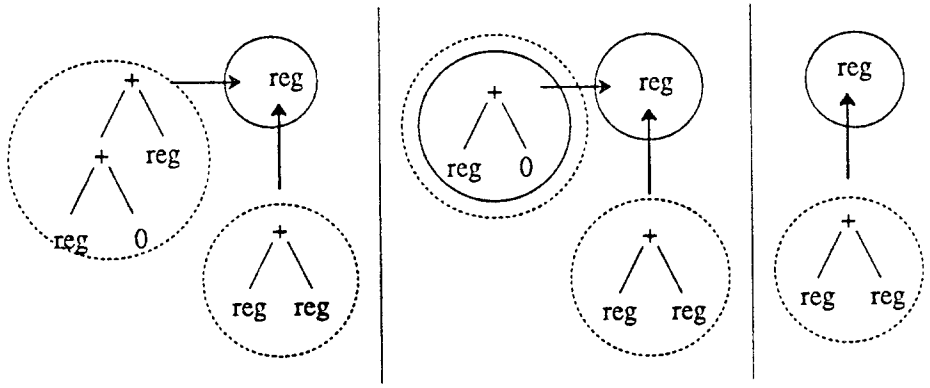
Reduction and NP-Completeness

Figure 5.10

Useless Nodes in LR Graphs

The original LR graphs were constructed to track all the possible rewrite sequences. The initial construction of the graphs guarantees that each node in the graph is useful, that is, there are

some trees for which an execution of the procedure *reachability* (T_{in},T_{out}) will require the presence of the node. The situation changes after the selection of the UI LR graphs. Since it reduces the possible normal form rewrite sequences that are tracked by the new states, it may make some information (nodes) useless. Consider a rewrite system with four rewrite rules: $0 \rightarrow reg$, $+(+(reg,0),reg) \rightarrow reg$, $+(reg,0) \rightarrow reg$, and $+(reg,reg) \rightarrow reg$, and the three LR graphs of Figure 5.11. Initially the three LR graphs are different. If the edges in thicker pen show the UI subgraph selected, the first and the last LR graphs are identical, but the middle one is not because $+(reg,0)$ is an output pattern (and an input pattern too). $+(reg,0)$ was useful because $+(+(reg,0),reg)$ was an input pattern of the first LR graph. Since that pattern is now "useless", the output pattern $+(reg,0)$ in the middle state can be removed, and the three states become the same.



Useless Nodes in LR Graphs

Figure 5.11

Detection of useless nodes can be done using a simple propagation algorithm:

Proposition 5.18 *Let R be a rewrite system over Op, and let G be a nullary operator in Op. Let* $<R,L_{in},\{G\}>$ *be in finite BURS, and let S be a collection of restricted LR graphs (which may or may not be UI). A node N in a graph is* ***useful*** *if there is a tree* $T_{in} \in L_{in}$ *such that N is part of a path (a local rewrite)* τ *selected by a call to reachability* (T_{in},G), *using S as states. If* L_{in} *is* L_{op} *there is an algorithm to determine the set of useful nodes in all the states.*

Proof Note that Proposition 5.16 says that no nodes are useless if S is the set of all LR graphs. The algorithm is that shown in Figure 5.12. Its correctness is straightforward. □

```
procedure useless-nodes()
    for each graph G
        for each node N in G
            set mark[N] = (if N = G then useful else useless );
    while (some mark on a node has changed) do
        for each graph G do
            for each output tree O in G do
                if mark[O] = useful
                    for each node N reaching O
                        set mark[N] = useful;
        for each graph G do
            for each input node ρ in G with mark[ρ] = useful
                let ρ = op(ρ1, . . . , ρn);
                for each i ∈ 1..n do
                    set mark[ρi]=useful in all graphs G
```

Algorithm for Useless Nodes

Figure 5.12

Detecting Blocks

If R is a rewrite system over Op, G a nullary operator in Op, and L a tree language, a *blocking tree* is a tree in L for which there is no rewrite sequence rewriting it to G. The *block detection problem* (Def. 2.28) is determining if there exists such a tree in L.

A given input tree T blocks for some fixed-goal REACHABILITY problem if and only if the state associated with T does not contain G. If the input language is L_{Op}, there will be an input tree that blocks if and only if there is a state that does not contain G. Hence,

Proposition 5.19 *Let R be a rewrite system over Op, and let G be a nullary operator in Op. Let $<R,L_{Op},\{G\}>$ be in finite BURS. There is an algorithm to solve BLOCKING for $<R,L_{Op},\{G\}>$.*

This section has considered only the case where the set of input trees is L_{Op}. The next section explores some consequences of using a general recognizable set as input.

5.5. Influence of the Input Set

The general REACHABILITY problem involves a rewrite system, R, a set of input trees, L_{in}, and a set of output trees, L_{out}. The previous sections have studied only some special cases of L_{in} and L_{out}. The limitation on L_{out} is not significant for the applications explored in this dissertation, but that on L_{in} could be. For example, Chapter 6 presents methods to solve the C-REACHABILITY problem and to use this solution for code generation of expression trees. In this case L_{in} represents the set of expression trees that can reach the code generator. This set should be used in determining if the rewrite system is in finite BURS: it could be that the rewrite system fails in a class of input trees outside of L_{in}. Hence it is desirable to solve the problems of the previous sections relative to a general class of input trees. The obvious candidate for this class are the recognizable sets.

Thus, it would be ideal to solve the problems attacked in the previous sections for the case when L_{in} belongs to RECOG. For completeness, it would also be nice to solve the problem for L_{out} in RECOG. This general problem will be the subject of future research but is not explored in this section. Instead this section assumes that $<R,L_{Op},\{goal\}>$ is in finite BURS and then solves some problems relative to to some recognizable set L_{in}.

The three problems considered are: determining if there exists a tree in L_{in} for which the rewrite system will block; determining which states (LR graphs or UI LR graphs) are useful in solving REACHABILITY for trees in L_{in}; and, determining which nodes in the state (LR graph or UI LR graph) are useful to solve REACHABILITY for trees in L_{in}.

The first problem is very easy:

Proposition 5.20 *Let $<R,L_{Op},\{G\}>$ be in finite BURS. Let L be a recognizable set. There exists an algorithm that will determine if there is a tree $T \in L$ on which R blocks.*
Proof Let A_1 be the B-fsa that computes LR graphs for $<R,L_{Op},\{G\}>$. Let A_2 be the B-fsa obtained from A_1 that accepts a tree if and only if it blocks for $<R,L_{Op},\{G\}>$. Intersect A_2 with the B-fsa that recognizes L obtaining a new automaton A_3. Finally, determine whether A_3 if empty or not. □

The second problem uses the general construction of Proposition 3.15.

Proposition 5.21 *Let R be a rewrite system over Op, and let G be a nullary operator in Op. Let $<R,L_{Op},\{G\}>$ be in finite BURS, and let A_R be the B-fsa that computes the LR (UI LR) graphs for it. Let L be a recognizable set. There exists an algorithm that will determine which states in A_R will be assigned to some subtree of a tree $T \in L$.*
Proof Without loss of generality, we can assume that the B-fsa recognizing L contains only one state st_{reject} that will not lead into an accepting state (in A_L). Compute, using Proposition 3.15, for each node st_L in A_L the set B of all those states st_R in A_R such that there is a tree T with state st_R in A_R that would have state st_L in A_L. After computing it, a state in A_R will be "useful" if it belongs to the set B labeling a non-rejecting state in A_L. □

Note that the construction of the previous proof can also be used to solve the "blocking" problem above: check if any of the rejecting states of A_R is in the set B labeling a non-rejecting state of A_L. Also note that we could not simply construct the intersection of A_R and A_L because A_R has more information than a simple B-fsa (that is the internals of the LR graphs are important).

The third problem is straightforward.

Proposition 5.22 *Let R be a rewrite system over Op, and let G be a nullary operator in Op. Let $<R,L_{Op},\{G\}>$ be in finite BURS. Let L be a recognizable set. Let A_R be the B-fsa computing the (UI) LR graphs for R. There exists an algorithm that will determine for each (UI) LR graph which nodes (patterns) can be used when solving the REACHABILITY problem for $<R,L,\{G\}>$.*
Proof First construct, using the previous propositions, a B-fsa that will compute the (UI) LR graphs and will have no useless states. Then apply the algorithm to compute useless nodes of Figure 5.12 □

5.6. Representation Issues

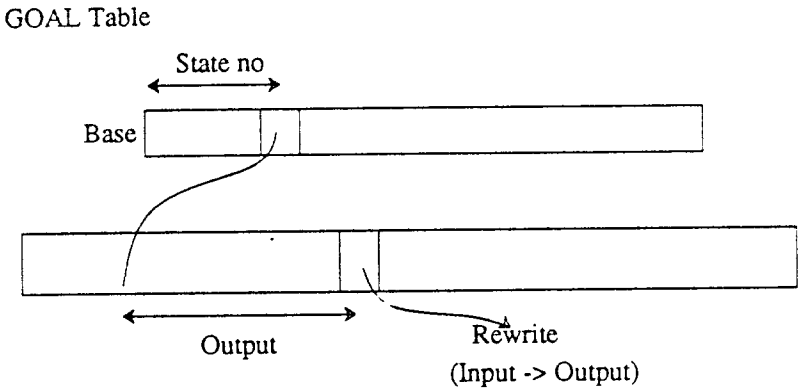
There are several ways of implementing the algorithm of Figure 5.8. This section explores briefly the alternatives that depend on the representation used in the UI LR graphs.

5.6.1. Implicit Representation

The most important case is when the rewrite system is in finite BURS. In this case the UI LR graphs, the local rewrite sequences, and the input and output patterns, can all be represented by an index into an adequate set. The computation of the UI LR graphs (the line marked (2) in Figure 5.8) can be encoded in a B-fsa. Finding, given a UI LR graph and an output goal, a local rewrite and an input goal (the line marked (1) in Figure 5.8) can be precomputed for all possible values, stored in a table, and then accessed through table lookups.

The implementation of the line marked (1) in Figure 5.8 requires some comment. If the goal pattern ρ is X, any path will satisfy the condition indicated in the line marked (1) and, in particular, a valid alternative is to simply stop the recursion at this point in the input tree. For the general case, the search for the T_o at which ρ matches can be done at table construction time and encoded in the tables as described in the next paragraph.

The representation of the B-fsa is similar to the one used for the computation of match sets. (The reader can see Chapter 3 for details). The representation of the paths in the LR graphs can be done in several ways. The fastest possible representation is directly as one 1-dimensional table per UI LR graph, with an entry for each output node giving the index of a rewrite sequence τ. Another alternative is to represent each individual edge in the UI LR graph in a similar form. In either case the 1-dimensional tables have many empty slots. In the execution of *reachability* (T_{in}, T_{out}) in Figure 5.8, these empty slots are only accessed at the top level, when asking if T_{out} belongs to the state. In fixed goal REACHABILITY, this information can be considered separatedly, encoding the single bit per state of whether a state reaches *goal* or not by renumbering states, using a bit-vector, or simply detecting and reporting their existence at solver construction time as a specification error. If one of these encodings is used, the empty slots can be considered ***don't care*** entries, and a cheap and efficient encoding is to overlay the rows using a base and displacement scheme like the one used in YACC [Joh78] (see Figure 5.13). Given a state number corresponding to a UI LR graph and a tree corresponding to an output node in it, the BASE array is accessed with the state number and the result value is then added to the index characterizing the tree to obtain an entry into GOAL. The obtained value is either the desired local rewrite sequence or just one rewrite rule of it depending on what is encoded.



Layout of Rows

Figure 5.13

5.6.2. Explicit Representation

Another alternative is to represent the UI LR graphs explicitly. Such an approach would require much smaller tables. The only information that needs to be encoded is the individual rewrite rules. Their applicability can then be investigated at solving time. The disadvantage is that the effort spent at solving time will be increased substantially. This approach is not explored further in this dissertation.

5.7. Related Work

Chronologically, this chapter grew from an attempt to generalize the results used in the chapter on code generation (Chapter 6) to the context of tree languages (Chapter 7). This explains the emphasis on fixed-goal REACHABILITY.

The closest research related to that presented in this chapter is in the area of code generation. Hatcher [HaC] was particularly instrumental in starting the research. Recently, the author has learned of several researchers that have developed techniques for code generation that are special cases of BURS theory [HeD87, WeW86]. Using this chapter's notation, their techniques involve implicit representation of some type of states for rewrite systems that are strict subclasses of reduction systems. Their work is discussed in Section 6.4.

The theory introduced in this chapter is used successfully in Chapter 6 to attack the problem of code generation, and in Chapter 7 for pattern matching and special types of tree transformations.

CHAPTER 6

Instruction Selection for Expression Trees

Infinite riches
in a little room

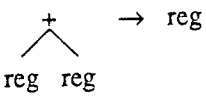
[Christopher Marlowe [1564-1593]]

In this chapter we investigate the optimal selection of instructions and addressing modes for expression trees. This problem can be modeled as a REACHABILITY problem extended with a cost metric on the rewrite sequences; the technique used in this chapter to solve this problem is based on extensions to the results of Chapter 5. The algorithms presented here are used in Chapter 8 to implement a code generator generator.

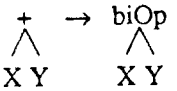
An expression can be represented as a "low level" computational dag: a dag showing common sub-expressions explicitly as subdags with more than one parent, and with nodes representing low level operations, including explicit memory assignment. (In other words, references to locations in arrays are described as indirections through displacements from a base location). Several simplifications must be made to this model before being able to map the problem of generating code into something similar to REACHABILITY.

The presence of common subexpressions substantially increases the complexity of generating optimal code. A traditional solution [Hen84, WJW75] is to generate code for dags by "pulling out" the parts of the input expression with more than one use and replacing them by a single node. The sub-expression then can be evaluated independently and computed into a temporary which is represented by the new node. In general this simplification will produce non-optimal code (for example, it could have been cheaper to recompute the sub-expression instead of computing it into a register) but it works reasonably well in practice. Hence, the first simplification imposed in this chapter is to assume that there are no common subexpressions; that is, that the computation dag is actually a tree.

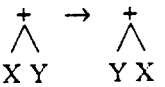
Generating code for the expression tree can be phrased as a problem involving a rewrite system describing the target machine architecture and some properties of the operators present in the expression tree. The effect of an instruction can be modeled as a rewrite rule by which some subtree representing the action of the instruction can be replaced by another representing its result. For example, the rewrite rule



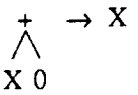
could represent a 3-address, register-to-register, "Add" instruction. Here, the nullary operator *reg* models a whole class of registers, not one particular member of the class. Rewrite rules can also be used to facilitate the description of the target machine by providing some type of "abstraction" facility, as in:



or to model such algebraic properties as commutative operators:



and other algebraic properties such as:



The completion of the computation is represented by a distinguished nullary operator, G, generated by rewrite rules such as *register*$\rightarrow G$. The instructions described through the rewrite system are "virtual" instructions: they need not be implemented directly by any one instruction in the physical hardware and can be implemented as an in-line sequence of physical instructions or by a call to run-time support routines. A rewrite sequence from an expression tree into G corresponds to an instruction sequence implementing the expression tree.

The operators of the expression tree and intermediate trees manipulated by the rewrite system have associated ***attributes***. Following [Hen84], these attributes can be classified into three groups depending on how their values affect the instruction sequences for an expression tree. ***Mandatory*** attributes are those whose value influences whether the shape of an instruction sequence is correct or not. A typical example is the data type of the operators; the correct instruction sequence for +(*register*,*register*) will depend on the relationship between the types of '+' and the two *register* symbols. ***Incidental*** attributes are those whose value is only used in the details, not the shape of the instruction sequence. An example is a nullary symbol *constant*: it models any constant[17]; which one is not important to determine the correct shape of the instruction sequence. ***Optional*** attributes are those whose value is important to determine the best instruction sequence, but discarding them still produces correct sequences.

Attributes with a finite domain set can be encoded syntactically into the names of the symbols. For example, if the data type has three possible values, *byte*, *word*, and *long*, the encoding of the data type attribute of the operator *register* produces the operators *register_b*, *register_w*, and *register_l*. One advantage of encoding mandatory attributes syntactically is that the correctness of the instruction sequences emitted can be enforced through syntactic means. If optional attributes are also encoded syntactically then the optimality of the sequence can be treated identically. [AGH84] contains a discussion of the advantages (and disadvantages) of encoding the mandatory attributes syntactically. Since this chapter and Chapter 8 model the selection of instructions exclusively through the use of a rewrite system with linear N-patterns, all mandatory and some optional attributes are encoded syntactically.

It may be impossible, or impractical, to encode some attributes syntactically. This may be the case if the register set of the target architecture is not "uniform" with "even-odd"

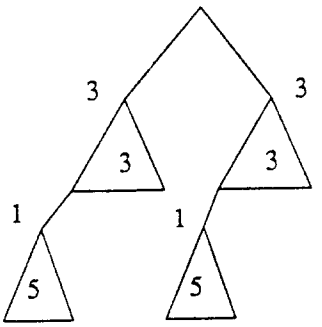
[17] If the target machine distinguished between constants of different size then the ***ranges*** of the constants would be either mandatory or incidental attributes depending on the architecture.

constraints on register pairs, special registers, or similar things. This problem can be alleviated, at some cost in the quality of the generated code, by modifying the "virtual" target architecture. [Hen84] discusses in detail a successful code generator generator based on this purely syntactic approach.

If a target machine architecture is described as suggested above, a rewrite sequence for an expression tree into *goal* corresponds to a valid instruction sequence for the expression tree. Any such instruction sequence can be provided with a ***cost***, and a ***resource usage***. The cost quantifies the "goodness" of the sequence. Examples can be the number of bytes required to represent the sequence and the number of cycles required to execute it. The resource usage describes how many resources the sequence needs for its evaluation. The prototypical example of a resource is a register. The problem of obtaining optimal code for an expression tree T using a machine description can now be phrased as finding a rewrite sequence of the machine description that will transform T into *goal* with minimum cost and with resource usage within the maximum number of resources available. This problem resembles the REACHABILITY problem, except for the two additional requirements: minimal cost, and limited number of resources.

Dealing optimally with a limited number of registers is a difficult problem. There are two major difficulties. One is that it is no longer possible to restrict our attention to rewrite sequences in bottom-up normal form: if the number of resources employed to hold a computation value is not always constant (as is the case with a value of type *double* requiring twice the number of registers than one of type *single*) there are non-bottom-up rewrite sequences that have lower resource usage, and equal cost, than any bottom-up normal form rewrite sequence [AJU77].

As an example of this difficulty consider an architecture having 6 identical registers and consider the expression tree sketched in Figure 6.1. Further, assume that the architecture is such that the number of resources to compute the lower two subtrees is 5, while those for the two upper subtrees is 3, and that the number of registers required to hold the value of the two lower subtrees is 1, while the two upper subtrees require 3. Then, there is a rewrite sequence that computes the tree using only the 6 available registers, but there is no rewrite sequence in bottom-up normal that does so. The desired sequence first computes the lower left subtree and saves the value using a single register; then computes the complete right subtree using the remaining 5 free registers, saves the value in 3 registers and finally completes the evaluation of the top left subtree using the saved value and the 2 free registers.



Example of a Optimal Sequence not in Normal Form

Figure 6.1

The second difficulty is that the resource usage of a rewrite sequence is not the sum of the resource usages of its components, as the example above also showed. As we will see later in this chapter, this characteristic would inhibit using fast techniques to find the optimal rewrite sequence, even if we were willing to restrict our attention only to bottom-up normal form sequences.

If, as in many target machine architectures, there is a finite number of resources, the limitation must be removed somehow. A common simplification is to assume an infinite number of registers. Code generation is then done using "virtual" registers which are later mapped somehow into "physical" registers. Two ways of doing this mapping are "on-the-fly", and "globally". On-the-fly register allocation is easy and computationally cheap. Global register allocation is "global" to some program unit, for example, expressions, basic blocks, procedures, separately compilable units, or the complete load module. A recent technique is based on the notion of "interference graphs" and "graph coloring" [Cha82]. The computational cost of using this technique is related to the size of the program unit, and to the "complexity" of the unit itself.

The C-REACHABILITY problem is, given a rewrite system R and two trees T and T', to determine whether there is a rewrite sequence in R from T to T' and, if so, to find the one with minimum cost. This chapter extends the techniques of Chapter 5 for REACHABILITY to C-REACHABILITY. C-REACHABILITY provides a solution to code generation with no constraints on the number of registers used. This later problem is frequently also called ***instruction selection***. The notion of blocking (Definition 2.28) applies well to this simplified model of code generation: a block corresponds to an input tree for which no code could be generated. Since this is a code generator error, it should be corrected by modifying either the set of input trees or the set of rewrite rules available. Sometimes [Hen84] blocking is called ***intrinsic blocking*** to differentiate it from ***algorithm-induced blocking*** in which code cannot be generated for some input tree due to an inadequacy of a code generation algorithm.

The next section defines formally a cost metric for rewrite sequences, C-REACHABILITY, and the notion of an ***instruction description***. Section 6.2 defines an extension of the LR graphs introduced in Chapter 5, called the δ-LR graphs, and presents a sufficient condition for the existence of a finite number of them. Then, Section 6.3 discusses the modifications that are needed to apply the algorithms for REACHABILITY to solve C-REACHABILITY. The two last sections discuss

related work and present some conclusions. Chapter 8 shows how the theory and techniques developed in this chapter are used to implement a code-generator generator.

6.1. Basic Definitions

"Optimal" code only makes sense in the presence of some cost metric. Since our approach to code generation is based strictly on rewrite systems, this metric is defined on rewrite sequences. The resulting notion is that of an ***extended rewrite system***: a rewrite system where a non-negative *cost* is associated with each rewrite rule.

Definition 6.1 *An **extended rewrite system** R_c is a pair <R,cost>, where R is a rewrite system, and cost is a function associating with each rewrite rule r in R a non-negative integer. R_c is said to be **an extension** of R.*

*If τ is a rewrite sequence in R, the **cost** of τ is the sum of the costs of all the rewrite rules in τ.*

Restricting the cost of a sequence to be linear on the cost of its composing rewrite rules is intrinsic to the approach presented in this chapter. It is possible to relax the linearity constraint to be a linear combination (this would require some redefinitions) but non-linear combinations are outside our approach. The reason will be apparent after defining the notion of a δ-LR graph.

The main problem investigated in this chapter is:

Definition 6.2 *Let R_c be an extended rewrite system over Op, and let L_i and L_o be two sets of trees over Op. The **minimum cost** REACHABILITY problem for R_c, L_i, and L_o, denoted by C-REACHABILITY, consists in determining, given $T \in L_i$ and $T' \in L_o$, whether there is a rewrite sequence τ from R_c, such that $\tau(T)=T'$, and if so to produce one such τ with cost minimum over all rewrite sequences in R_c rewriting T to T'.*

*If L_o is a singleton $\{G\}$, then the C-REACHABILITY problem is said to be **fixed-goal** C-REACHABILITY, and G is called the **goal**.*

Fixed goal C-REACHABILITY provides a solution to instruction selection. Instruction selection uses a special class of extended rewrite systems, the ***instruction set descriptions***. These extended rewrite systems are given a "semantics" by associating with each rewrite rule an uninterpreted string.

Definition 6.3 *Let Op, OpC, and IfC be three mutually disjoint sets of operators, called the **input operators**, the **generic operators**, and the **instruction fragment symbols**, with all the operators in IfC being nullary, and one of them being a distinguished member G. An **instruction set description** over Op, OpC, and IfC is an extended rewrite system where the rewrite system is a reduction system R_c over $Op \oplus OpC \oplus IfC$ together with a function assigning to each rewrite rule its **semantic action**, a (maybe null) string word in some alphabet.*

Three types of rewrite rules are given special names:

*An **instruction fragment** is a reduction rule of the form $p \rightarrow cl$ where $cl \in IfC$ and p is a pattern without variables.*

*A **generic operator** rewrite is a rename rule of the form $op(X_1, \ldots, X_n) \rightarrow op'(X_1, \ldots, X_n)$, where $op \in Op$ and $op' \in OpC$.*

*A **commutative operator** rewrite is a rename rule of the form $op(X_1, \ldots, X_n) \rightarrow op(X_{\pi_1}, \ldots, X_{\pi_n})$, where π is a permutation of $1..n$.*

*An instruction set description is **flat** if IfC is a singleton, otherwise it is **factored**. An instruction set description **has operator classes** if $OpC \neq \emptyset$.*

An instruction set description is a ***simple machine grammar*** *if it has only instruction fragment and generic operator rewrites.*

The original article introducing the Graham-Glanville technique [GlG78] used only instruction fragment rewrites. [Hen84] used factored machine descriptions with operator classes, but handled commutative operators through transformations on the machine description itself. Later we will see that erasing reduction rules are difficult to handle with our approach.

An application of a rewrite rule transforms a tree T into another tree T'; the semantic actions extend the mapping to attributed trees. By extension, the sequence of semantic actions associated with a rewrite sequence from an input tree T into T' defines the value of the attributes of T'. In practice, the semantic actions also have side-effects which are used to emit instructions as they are found. The instruction selection problem can be solved by first solving the fixed-goal C-REACHABILITY problem for the rewrite system, thus obtaining a rewrite sequence for the input tree T into the goal G, and then using this rewrite sequence to rewrite T into G while using the semantic actions associated with each rewrite rule in the rewrite sequence to find the attributes of the intermediate trees.

We will not discuss how to write the semantic actions (see [Hen84] for details), but we explain how some of them can be null. A simple example is a rename rule corresponding to an abstraction like *amode_index*→*amode* where *amode* and *amode_index* have exactly the same attributes. Formally, the semantic action would copy all the attributes of *amode_index* into the corresponding attributes of *amode*, but it may be possible to do better. For instance, assume that the attributes are represented independently of the intermediate tree as objects pointed to by a semantic stack. Then a null semantic action, leaving the stack undisturbed, would have the desired effect. A similar situation would happen if a node in an intermediate tree is represented by an object and the rewrite application only changes its "label" field, leaving the "attribute" fields unchanged. Even more, if no later semantic action refers to the label of the node, the original label could be left unchanged; this works correctly because we do not use the label of the intermediate tree to determine what rewrite rules are applicable: that decision was made when solving the C-REACHABILITY problem.

A more complicated example is a generic operator rewrite rule like $+(X,Y)\rightarrow biOp(X,Y)$. One possible semantic action would change the node + into *biOp* and update an attribute, *class*, of *biOp* to indicate that that *biOp* is an abstraction of a +; later semantic actions would then use the *class* attribute. Another possibility is to leave the + node unmodified, and to write the later semantic actions so that they use the label of the node instead of the *class* attribute. In the second case the semantic action can be null.

As an example of a situation where a semantic action cannot be null consider the commutative rewrite rule $+(X,Y)\rightarrow+(Y,X)$. Although no "new" attributes are computed in this rewrite application, some action will be necessary so that references later in the rewrite sequence to positions in the right subtree of + (the X) will find the values that previously were in the left subtree of +. What specific computation is done depends on how the intermediate trees and their attributes are stored. A similar situation occurs with the rewrite rule $+(X,0)\rightarrow X$.

The formalization of the instruction selection problem is as follows:

Definition 6.4 *Let L be a set of trees over Op, and let ISD be an instruction set description over Op (and OpC and IfC) with G its goal symbol. The* ***unconstrained code generation problem,*** *UCODE*[18]*, consists in determining, for each input tree T in L, whether there is a rewrite sequence*

[18] The U stands for "unconstrained". The constrained code generation problem, CCODE, not defined here, is the corresponding problem where the number of resources used by the sequence of semantic actions is constrained to be smaller than some value.

τ from ISD, such that $\tau(T)=G$, and if so to produce the sequence of semantic actions of such a τ with the additional property of having a cost minimum over all rewrite sequences reducing T to G.

Clearly, solving C-REACHABILITY provides a solution to UCODE, but the converse is not necessarily true due to the presence of null semantic actions (see section 6.3).

Since the rewrite system underlying an instruction set description is a reduction system, we have:

Proposition 6.1 *Let R be the rewrite system of an instruction set description over Op with goal G. Then $<R,L_{Op},\{G\}>$ is in finite-BURS.*

Figure 6.2 gives an example of an instruction set description. The underlying rewrite system is the same as that of Figure 5.1. In the figure, $Op = \{0,Const,Reg,+,-\}$, $OpC = \{biOp\}$, and $IfC = \{reg,amode\}$. The rewrite rules correspond to a 2-address operation, a load, two rename rules, three addressing modes, one commutative operator, two operator class rewrite rules, and a simple reduction rule corresponding to an algebraic law. The first column indicates the type of the rewrite rule (Frag = instruction fragment, GenOp = generic operator, Comm = commutative operator, Red = non-erasing reduction), the second column the rewrite rule itself, and the last two columns its cost and an indication of the whether the semantic action is null or not.

Type	Rewrite	Cost	Action
Frag	biOp (amode amode) → reg	1	–
Frag	Reg → reg	0	–
Frag	amode → reg	1	–
Frag	0 → Const	0	null
Frag	reg → amode	0	–
Frag	Const → amode	0	–
Frag	+ (Const reg) → amode	0	–
Comm	+ (X Y) → + (Y X)	0	–
GenOp	+ (X Y) → biOp (X Y)	0	null
GenOp	- (X Y) → biOp (X Y)	0	null
Red	+ (X 0) → X	0	–

Example of an Instruction Set Description

Figure 6.2

In Figure 6.2 the cost of a rewrite rule is 1 if the rewrite rule corresponds to an instruction and 0 otherwise. Other examples of cost functions are the number of bytes used to represent an instruction sequence, the number of bytes of memory data touched by the execution of an instruction sequence, and the number of cycles needed to execute an instruction sequence. Note that the last two cost functions cannot be modeled accurately in our framework in the presence of several important architecture features such as pipelines, cache memory, and page faults. As in the case of a finite number of registers, the solution is to ignore the limitation of the model, generating as good a code as possible and later using an instruction scheduling algorithm to improve the obtained instruction sequence.

6.2. δ-LR Graphs

Chapter 5 gives two notions of state for solving REACHABILITY: LR graph and UI LR graph. This chapter shows how to enrich these notions with cost information. The new notions are called δ-LR graphs, discussed in this section, and δ-UI LR graphs, which are discussed in the next one. Although not all extensions of a finite-BURS rewrite system will have a finite number of δ-LR graphs, extended rewrite systems that model real instructions sets have a finite number of δ-LR graphs. This observation leads to a code generator generator system described in Chapter 8.

Since the cost of a rewrite sequence does not depend on the particular order of its components, the rewrite sequence of minimum cost in a rewrite system can be found by considering only rewrite sequences in bottom-up normal form. Hence, it is meaningful to try to extend the notion of an LR graph by simply extending the meaning of its nodes. In a first approach, which we call ***full cost LR graph***, the nodes of the graph for a tree T represent the pairs $<\rho, c(\rho)>$, where $c(\rho)$ is the minimum cost required to rewrite T into a pattern of interest ρ, while the paths would represent local rewrite sequences with minimum cost. This definition leads to a notion that can be used to solve C-REACHABILITY in a manner very similar to that used to solve fixed-goal REACHABILITY: a first pass, bottom-up, computing the state information; a second pass, top-down, selecting minimum cost local rewrite sequences; and a third pass, bottom-up, collecting these local rewrite sequences.

Unfortunately, full cost LR graphs do not lead to an efficient implementation. The main problem is that only a few trivial extended rewrite systems will have a finite number of full cost LR graphs. Most interesting systems have unbounded sets of trees for which the minimum cost of rewriting a tree into a given pattern ρ is a function that grows with the height of the tree (an example is the system of Figure 6.2). This unboundness precludes the use of a fully implicit representation of the states and the implementation of the phases as simple table lookups. And, if the costs are carried explicitly at C-REACHABILITY solving time, the computation of the minimum cost requires additions and comparisons in the first, bottom-up, computation phase.

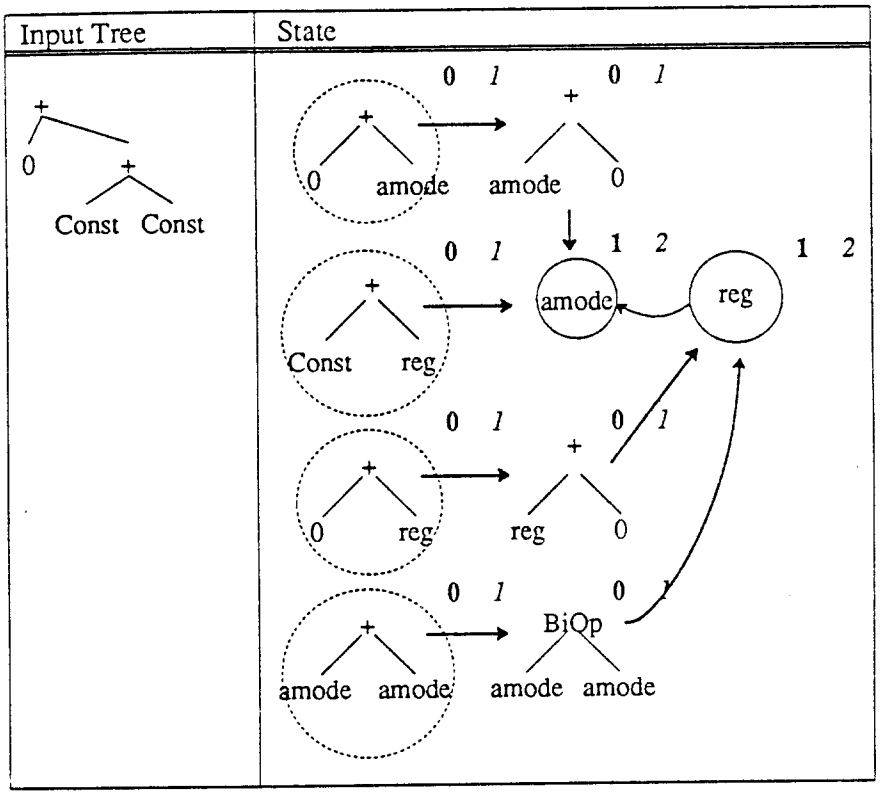
The notion of δ-LR graphs is an elaboration of the one presented above based on the observation that there are only two main requirements for the information kept in the states, similar to those required of states in Chapter 5:

(C-STATE-1) The collection of states associated with the nodes contains enough information to characterize the minimum-cost rewrite sequence applicable to the input cost reaching the goal tree.

(C-STATE-2) The states can be computed in a bottom-up pass over the input tree.

Note that neither (C-STATE-1) nor (C-STATE-2) indicate that the states need to encode the cost of the minimum cost rewrite sequence. The notion of a full cost LR graph satisfies both (C-STATE-1) and (STATE-2), but it is too expensive. A first attempt at a new notion of state would record not the costs themselves, but only their relative ordering. This certainly satisfies (C-STATE-1), but, as the reader can verify, does not satisfy (C-STATE-2). The correct notion of state is somewhere in between these two notions: it stores the difference between the costs associated with the patterns in the state. We call this notion a δ-LR graph, and it can be obtained from

the notion of a full cost LR graph by finding the smallest of all the costs associated with nodes in the full cost LR graph and then subtracting this value from all of the costs. Figure 6.3 is the analog of Figure 5.6 but showing δ-LR graphs instead of LR graphs. The δ costs are shown to immediately to the right of the patterns in bold font, and the full costs to their right in italics.



Example of a δ LR Graph

Figure 6.3

There are two differences between the δ-LR graphs and the LR graphs in Figure 6.3 and Figure 5.6. One is the addition of the cost information, the other is that the edge between *amode* and *reg* has been removed because it does not belong to a rewrite sequence with minimum cost.

δ-LR graphs (and δ-UI LR graphs) are used to solve C-REACHABILITY in a slightly different way than the way in which LR graphs are used to solve REACHABILITY. If the cost information is removed from the δ-LR graphs the result is a restriction of the corresponding LR and UI LR graphs. This restriction is enough to satisfy (C-STATE-1). The cost information in the δ graphs is present only to satisfy (C-STATE-2), to be able to compute the states in a bottom-up pass. This situation was formalized in the notion of LB-fsa of Section 2.4.1, and is explored in Section 6.3.

The formal definition of a δ-LR graph is:

Definition 6.5 *Let R be a rewrite system over Op, let G be a nullary operator in Op, let $\langle R,cost\rangle$ be an extended rewrite system, and let $\langle R,L_{Op},\{G\}\rangle$ be in BURS. The δ-LR graph associated with a tree T over Op is a graph $G=(V,E)$ defined as follows.*

Let (V_0,E_0) be the LR graph of T for R and G. Define, for a pattern $\rho \in E_0$, its cost, $c(\rho)$, to be the cost of the minimum cost rewrite sequence in R_c from T into ρ. Let C_{min} be the minimum value of $c(\rho)$ for all ρ in E_0. The members of V are pairs $\langle \rho, c(\rho)-C_{min}\rangle$ where $\rho \in V_0$. The members of E are those labeled edges $\langle \rho_1,\rho_2\rangle$ in E_0 with label τ_1 for which $c(\rho_1)=c(\rho_2)+cost(\tau_1)$.

It follows that for any tree T, the graph that contains the same edges as the δ-LR graph of T but whose vertices are only the patterns is a restriction (Def. 5.10) of the LR graph of T.

The formalization of (C-STATE-1) and (C-STATE-2) are Propositions 6.2 and 6.3 respectively. Their proofs follow in an straightforward way from the corresponding propositions for LR graphs.

Proposition 6.2 *Let R be a rewrite system over Op, and let G be a nullary operator in Op, such that $\langle R,L_{in},\{G\}\rangle$ is in BURS. Let $\langle R,cost\rangle$ be an extended rewrite system. Let (1) and (2) be the statements described below. If L_{in} is L_{Op} then (1) and (2) are true.*

(1) For every tree A in L_{in}, every position p in A, and every efficient minimum-cost normal form rewrite sequence τ with $\tau(A)=G$ with local rewrite sequence τ_0 at p of length m, let π be the normal form rewrite sequence assigned by τ below p. There is a path in the δ-LR graph of $A_{@p}$ of length m and such that the j-th pattern in the path matches $pre(\tau_0,j)(\pi(A_{@p}))$ for $0\le j\le m$.

(2) For every tree T, and for every non-looping path $\rho_0 \cdots \rho_m$ from an input node to an output node in the δ-LR graph for T there is a tree $A \in L_{in}$, a position p in A with $A_{@p}=T$, and an efficient minimum-cost normal form rewrite sequence τ with $\tau(A)=G$, local rewrite sequence τ_0 at p, and with normal form rewrite sequence π assigned below p, such that τ_0 has length m and for $0\le j\le m$, ρ_j matches at $pre(\tau,j)(\pi(A_{@p}))$

Proposition 6.3 *Let R be a rewrite system over Op, let G be a nullary operator in Op, let $\langle R,cost\rangle$ be an extended rewrite system, and let $\langle R,L_{Op},\{G\}\rangle$ be in finite BURS. Let $Gr(T)$ denote the δ-LR graph associated with the tree T. There is a function f such that $Gr(op(T_1,\ldots,T_n))=f(op,Gr(T_1),\ldots,Gr(T_n))$.*

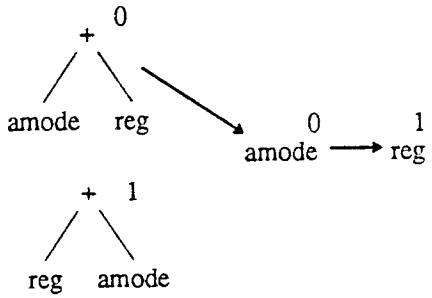
The following example shows that in general, LR graphs do not provide enough information to solve C-REACHABILITY. Consider the rewrite system in Figure 6.4, where the integer on the right is the cost of the rewrite rule to its left.

$+(reg, amode) \rightarrow amode$	0
$+(amode, reg) \rightarrow amode$	0
$amode \rightarrow reg$	1
$Const \rightarrow amode$	0
$reg \rightarrow amode$	0

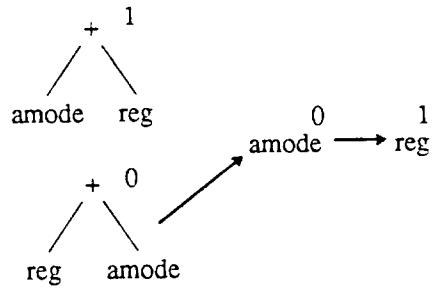
Example of Different δ-LR graphs per LR graph

Figure 6.4

There are input trees with the same LR graph but with different δ-LR graphs. The δ-LR graph for the tree $+(Const, reg)$ would be:



while the δ-LR graph for the tree $+(reg, Const)$ would be:



Both trees have the same LR graph.

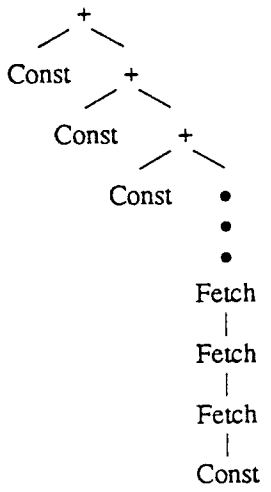
Finite Number of δ-LR Graphs

The definition of δ-LR graph does not guarantee a finite number of states, even if the underlying rewrite system is finite BURS. The top of Figure 6.5 shows such a rewrite system. To show that it is necessary to track an unbounded number of states, consider the class of trees represented at the bottom of the figure. There are two possible ***disjoint*** rewrite sequences rewriting this tree into *goal*: one using *amode*, the other using *imode*. Which one yields the lowest cost depends on whether there are more or fewer nodes labeled with '+' than labeled with *Fetch*.

It is important to note that the lack of finiteness is not a deficiency of the definition of δ-LR graphs: any definition that is based on the bottom-up traversal paradigm that we use in Chapter 5 and in this chapter, requires an unbounded number of states to count the sizes of each tree and

compare them.

Rewrite	
Fetch (*Const*)→*amode*	2
Fetch (*amode*)→*amode*	2
+(*Const*,*amode*)→*amode*	1
amode →*goal*	0
Const →*imode*	1
Fetch (*imode*)→*imode*	1
+(*Const*,*imode*)→*imode*	2
imode →*goal*	0



Unbounded Number of δ-LR Graphs

Figure 6.5

Determining whether there will be a finite number of δ-LR graphs for some extension of a rewrite system is semi-decidable if we already know that the rewrite system is in finite BURS.

Proposition 6.4 *Let R be a rewrite system over Op, and let G be a nullary operator in Op, with* $\langle R, L_{Op}, \{G\}\rangle$ *in finite BURS. Let* $\langle R, cost\rangle$ *be an extension of R. There is a procedure that will generate all the δ-LR graphs for* $\langle R, cost\rangle$ *and G and will stop if there is a finite number of them.*
Proof A closure algorithm that generates keeps generating δ-LR graphs while necessary will stop if and only if there is a finite number of δ-LR graphs. □

Although Figure 6.5 generates an unbounded number of δ-LR graphs, a simple modification guarantees a bounded number of states. If *reg* is added, together with the rewrite ruless *amode* → *reg*, *imode* → *reg*, and *reg* → *imode*, and *reg* → *amode*, then, the two rewrite sequences cease to be disjoint, since it is possible to switch from one to the other. This implies

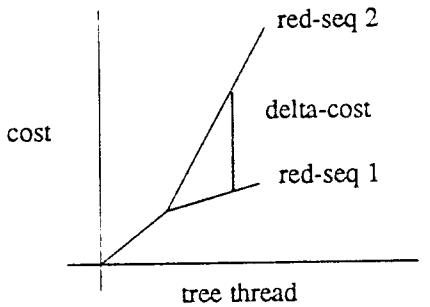
that the difference between the minimum cost in one sequence and the other is never larger than the cost required to switch from one sequence to the other. Fortunately, the new rewrite system is more representative of "real" machine architectures.

The remainder of this section gives a sufficient condition for the existence of a finite number of δ-LR graphs. An intuitive discussion of the situation is given using a graphical representation introduced in [Hen84]: the ***cost diagrams***. Such a diagram is an exact description of the situation only for trees with a unique rewrite "thread", as with trees that contain only unary and nullary operators, but hopefully it will help the intuition of the reader.

The horizontal axis of a cost diagram corresponds to the size of the tree along its only thread. The vertical axis indicates the cost associated with a rewritten tree. A rewrite sequence is described through a ***path***: a connected sequence of straight segments, one for each rewrite rule in the sequence. Several rewrite sequences will show as several paths. Paths may split, join, and cross, according to the behavior of their associated rewrite sequences. The optimal code is that associated with the lowest path that reaches the end of the thread (that is, it does not block).

In this graphical presentation, the δ-LR graph associated with a subtree corresponds to a vertical slice of the cost diagram, with a horizontal coordinate corresponding to the subtree. The paths in the slice correspond to the paths in the δ-LR graph; since the absolute cost is the vertical coordinate of the diagram, the relative cost is related to the distance between the paths. Asserting that the number of δ-LR graph is finite is equivalent to saying that the distances between paths are bounded.

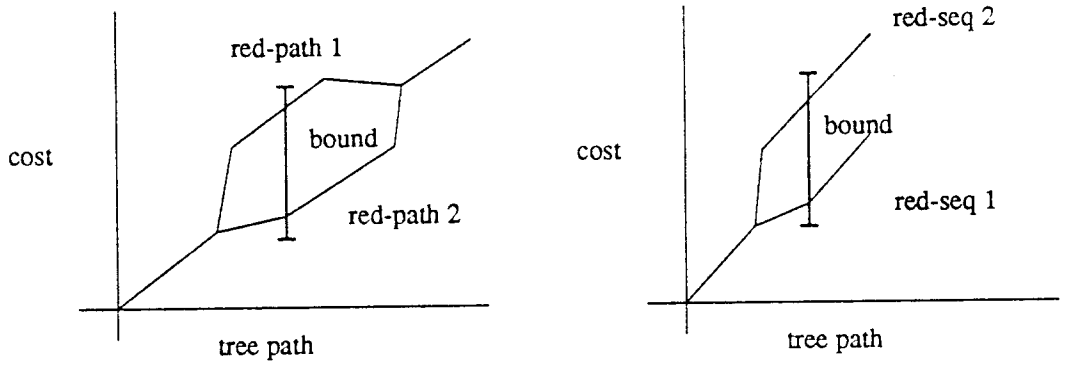
Figure 6.6 shows a tree thread where there are two possible reduction sequences. If both were to reach the end of the thread, the lower one would be the one to choose.



Two Valid Rewrite Sequences

Figure 6.6

As illustrated in Figure 6.7, the case where the cost difference is bounded corresponds to those cases where after separation there is a fast "join", or those where the paths stay separated but approximately "parallel".



Bounded δ Costs

Figure 6.7

If the paths diverge, there may be an infinite number of states. Fortunately this is unrepresentative of real machine architectures. Consider a typical register-oriented instruction set without memory-to-memory instructions. Every tree can be evaluated into a register, since register is always a good target for an instruction. In addition, every "long enough" rewrite sequence for an expression has to load values into registers (in order to continue the rewriting). Intuitively this implies that two paths in a cost diagram cannot diverge very much: given two rewrite sequences for the same input tree, but with different output trees, ***register*** can be used to "bridge" from one rewrite sequence to the other, thus insuring that, if they are both of minimal cost (for the given output trees), the difference in their costs is "small" (i.e. bounded). This observation can be formalized in the next proposition, where ***register*** is generalized to a set of "key" symbols S (in part to deal with the memory-to-memory instructions).

Proposition 6.5 *Let R be a rewrite system over **Op**, and let G be a nullary operator in **Op**, with $\langle R, L_{Op}, \{G\}\rangle$ in finite BURS. Let $\langle R, cost\rangle$ be an extension of R. Let S be a subset of $EF_{R,G}$ such that there are positive integers k_1 and k_2:*

(1) For any pattern ρ in $EF_{R,G}$ and any $s \in S$, ρ can be rewritten into s with a sequence having cost at most k_1.

(2) There is an integer k_2 such that for any tree T and any normal form rewrite sequence γ for T there is a permutation of γ of the form $\gamma_1\gamma_2$ with γ_2 of cost at most k_2, and such that the frontier of $\gamma_1(T)$ contains only members of S.

Then the set of different possible δ-LR graphs is bounded.

Proof Since R is finite BURS, the only way to have an unbounded number of δ-LR graphs is to have an unbounded delta cost in the graph. Let T be an input tree, and α and β be two normal form rewrite sequences leading to ρ_α and ρ_β, in $EF_{R,G}$. Without loss of generality assume that $cost(\alpha) \leq cost(\beta)$. Then, we want to show that there is a constant K, independent of T, α, and β such that there is a rewrite sequence β' applicable to T and leading to ρ_β and such that $cost(\beta') \leq cost(\alpha)+K$. β' is constructed from β and α. The proof proceeds by structural induction.

From (2) applied to β and T, β is of the form $\beta_1\beta_2$, with $\beta_1(T)=T'$ and $\beta_2(\beta_1(T))=\rho_\beta$. The new β' is obtained by replacing β_1 by an equivalent rewrite sequence. By normal form and by structural induction hypothesis on (1), there is a permutation of β_1 of the form $\gamma_1 \cdots \gamma_m$, where γ_i applies to t_i, a subtree of T, and $\gamma_i(t_i)$ is $s_i \in S$. Let α_i be the restriction of α to t_i. Consider now the application of α_i to t_i, and call it Ta_i. Ta_i is a member of $EF_{R,G}$ because $\alpha(T)=\rho_\alpha \in EF_{R,G}$. Hence, by (1), there is a rewrite sequence θ_i with cost at most k_1 such that $\theta_i(Ta_i)=s_i$. The new β'_1 is $\alpha_1\theta_1 \cdots \alpha_m\theta_m$. The cost of this sequence is bound by $m \times k_1 + cost(\alpha)$. Hence, the cost of β' is bound by $m \times k_1 + k_2 + cost(\alpha)$. □

The condition of Proposition 6.5 is not a necessary condition; there are many ways to strengthen it and still have a finite number of δ-LR graphs. For example, there will still be a finite number of δ-LR graphs if the rewrite rules can be divided into two disjoint sets corresponding to floating and integer expressions, with each one satisfying the conditions of the proposition. The proposition is provided here to give a flavor for the type of conditions that produce a finite number of δ-LR graphs.

Erasing reduction rules almost always lead to an unbounded number of δ-LR graphs. Consider a rewrite system with rewrite rules $Mul(X,0) \rightarrow 0$, $0 \rightarrow const$, and $Mul(reg, const) \rightarrow reg$, and an input tree such as $Mul(T_1,0)$ where T_1 can be rewritten into reg in K rewrite applications. Define the cost of the rewrite system to be the number of rewrite applications. The δ-LR graph associated with such an input tree will have an output pattern "0" with cost 1, and an output pattern "*reg*" with cost $K+2$. If there is one such T_1 for each positive integer K, the number of δ-LR graphs will be unbounded.

Alternatives to δ Costs

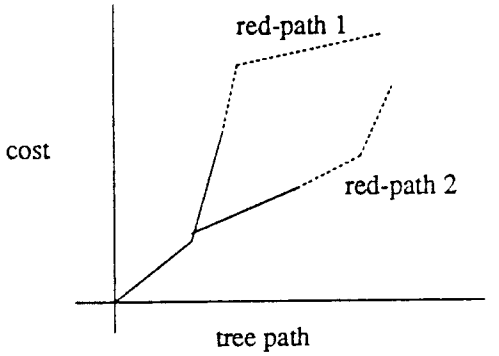
Some extended rewrite systems with an unbounded number of δ-LR graphs can be handled by approximating "large" differences between costs as infinite. Using ordinal numbers one could have a sequence, for example,

$$<e_1,0>,<e_2,\omega>,<e_3,\omega+1>,<e_4,2*\omega>$$

This would be interpreted as meaning that the difference in cost between e_1 and e_2 is "unsurmountable" as is that between e_3 and e_4, but that e_3 is only one unit more expensive than e_2. The difference between the two alternatives reaches an "infinite" value when the alternative with higher cost will be used only if the other one blocks. Cost differences can increase from being finite to being infinite, but not otherwise.

An example of the situation could be an target architecture where it is possible to choose between two different banks of registers but, once a bank is selected it is not possible to change between banks. In addition, the costs of instructions in one bank must be uniformly larger than those in the other. To generate optimal code it is necessary to track the potential costs of each register bank selection, but only up to the point where it is obvious which selection was the best one.

It is not always possible to use an approximation to infinite ordinals to solve the problem. In some cases the two paths first diverge an unbounded distance, and then can slowly get back to a middle ground, as shown in Figure 6.8. Related to the previous example, this corresponds to the case where the advantage of using one register bank over the other is not "uniform".



Sequences of Rewrites that Split and Join

Figure 6.8

In the rest of this chapter it is assumed that it is possible to characterize the cost information using δ-costs, either using finite ordinals or the modified infinite ordinals suggested above.

6.3. Solving C-REACHABILITY and UCODE

The previous section showed how to characterize minimal-cost rewrite sequences through δ-LR graphs. C-REACHABILITY can now be solved by solving a fixed-goal REACHABILITY problem using the algorithm of Figure 5.8 on any uniquely invertible subgraph of the δ-LR graphs. Section 5.4 discussed how to select the UI LR graphs for the case of LR graphs. The same considerations apply here with one difference. It has already been pointed out that the purpose of the B-fsa computing the δ-LR graphs is not to compute them but to eventually obtain the restriction of the LR graphs equivalent to the δ-LR graphs. Because of this, the whole process is best described as the application of an LB-fsa, with the B-fsa computing the δ-LR graphs and with labeling function the one that –informally put– strips the costs away from the δ-LR graphs.

This difference is apparent when minimizing the bottom-up tree automaton. Since in REACHABILITY all the state obtained by the automaton is used (i.e. the labeling function is the identity), removing identical states guarantees a minimum LB-fsa. In C-REACHABILITY the labeling function changes the situation since it ignores cost information. As an example, Figure 6.9 shows the nodes of two δ-LR graphs, St_1 and St_2 from an instruction set description for the Vax-11 [DEC81] with the number of bytes referenced as the cost function. St_1 and St_2 have the same underlying LR graph restriction. Moreover, the nodes can be split into two parts (shown in the figure) so that each subset in both graphs, when considered in itself, contains the same δ cost information, and in all contexts only one of the parts is used. Hence, St_1 and St_2, although being different δ-LR graphs, are equivalent for solving C-REACHABILITY, and would be found so using the algorithm of Proposition 2.27.

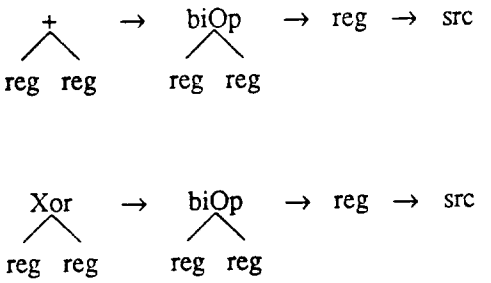
	St_1	St_2
Pattern	cost	cost
compute_trees	4	5
rval_b	4	5
rval_w	7	8
rval_l	7	8
register_l	6	7
register_b	3	4
register_w	6	7
(And_b rval_b constant_l)	1	2
tree	4	5
amodes_uncon	7	8
byte_r_addr	7	8
(And_b Icon_l rval_b)	0	0

Two Equivalent δ-LR graphs

Figure 6.9

A similar situation appears when solving UCODE. UCODE can be solved by solving C-REACHABILITY and using the rewrite sequence to obtain a sequence of instructions, or, more accurately, a sequence of calls to semantic routines that will generate the instructions. Instead, the information can be extracted directly from the δ-UI LR graphs, leading to a situation as before, except that now the labeling is into a graph where the edges are semantic calls. Some states in the B-fsa may be equivalent when there are different rewrite rules with the same semantic calls. Again, we have an LB-fsa that can be minimized using the algorithm of Proposition 2.27.

As an example of this new situation, assume that there are rewrite rules $+(X,Y) \rightarrow biOp(X,Y)$, and $Xor(X,Y) \rightarrow biOp(X,Y)$, where X and Y are variables. Further assume that both + and *Xor* are not used anywhere but in this generic operator rewrite, and that the two rewrite rules have the same semantic action routine (this could be true, for instance, if later semantic actions consult the label of the input tree to distinguish between the two cases). Two δ-UI LR graphs that have identical input sets except that all the patterns in one are labeled with + and in the other with *Xor*, will behave the same, will produce the same semantic actions, and will be equivalent for UCODE, as shown in Figure 6.10. But the two δ-UI LR graphs are different for C-REACHABILITY.



An Example of two LR graphs Equivalent for UCODE

Figure 6.10

6.4. Related Work

Recent years have seen a considerable amount of work in code generation algorithms. This section briefly covers those that relate to the work presented in this chapter. As in many research contributions, sometimes it is difficult to determine the exact chronology and authorship of the different techniques.

Graham-Glanville

The Graham-Glanville (GG) code generation algorithm [GlG78], [AGH84], [Hen84] is a code generator based on SLR(1) technology. It maps trees into strings through their preorder traversal, and has a left-to-right bias (due to the mapping mentioned above) and a "greedy" approach to choose among different alternatives (the so-called "maximum munching" heuristic). This makes the code generator potentially non-optimal. Under the mapping from trees to strings, the transformation rules correspond to grammar rules, instruction fragment class symbols to non-terminals, and IR operators to terminals. This is the origin of the term ***machine grammar*** formalized in Def. 6.3. The sentential forms in the grammar correspond to the elements of the extended pattern set.

GG has been used to construct practical code generator generators, including CODEGEN [AGH84]. CODEGEN follows closely the model presented in this chapter[19] using a purely syntactic approach and an "on-the-fly" register manager. Thus, it essentially tries to solve an approximation to the UCODE problem.

The main advantage of GG is the re-use of a well known technology, SLR(1), for pattern matching and rewrite rule application. The main disadvantage of GG is that it cannot guarantee optimality. It is possible to perform analysis at table construction time [Hen84] that will provide the user of the system with information on those cases where the system will not perform as desired, and, in some cases, to provide automatic corrections. But, in general, the user of the technique requires some understanding of its fundamentals to write descriptions that will work

[19] Or the other way around.

properly. Despite its limitations GG has been quite successful and has been a very influential milestone.

Dynamic Programming

In its simplest form ***dynamic programming*** is a tabular technique used to compute the value of certain recursive functions [AHU75]. That is, if $P(n)$ is a problem of size n, and f is a function on the problem that can be provided with a recursive decomposition into subproblems $f(P_1(n-1)),...., f(P_m(n-1))$, then a table is used to "save" the evaluation of each subproblem as it is solved and thus avoid re-evaluation.

The term dynamic programming is also used to mean a more specific optimization technique in mathematical programming for problems involving a ***multistage decision process*** [RND77]: a process in which a sequence of decisions is made, the choices available being dependent on the current state of the system – that is, on the previous decisions. The optimization problem is to find a sequence of decisions that minimizes some objective function. The key property required of the problem is the ***principle of optimality***:

> An optimal sequence of decisions has the property that whatever the initial state and initial decisions are, the remaining decisions must be an optimal sequence of decisions with regard to the state resulting from the first decision

The principle of optimality leads to a recursive formulation of the minimum cost of the objective function. This value can then be computed using the tabular technique mentioned above. A sequence of decisions for it can then be extracted by following the decomposition "backwards" (maybe with the help of additional information kept in the table) and combining it "forwards".

Aho and Johnson

The notion of dynamic programming is used by Aho and Johnson in [AhJ76] to obtain an algorithm for code generation for expression trees. There is a finite number, N, of identical registers (denoted by r), and an infinite number of memory locations (m). There are only two types of instructions: register operations of the form $T(r,m) \rightarrow r$, where $T(r,m)$ is a tree with operators including r and m, and memory stores of the form $r \rightarrow m$. The cost of a rewrite sequence is defined to be its length, and its resource usage is the maximum number of registers it uses at any given moment. (A rewrite rule of the form $r \rightarrow m$ frees registers). The optimization problem is to determine the sequence of rewrite rules that will rewrite the original input tree into r with a resource usage of less than N registers.

Aho and Johnson first show that it is possible to consider only bottom-up normal rewrite sequences[20]. Then they define a notion of state at a subtree that describes, for each i with $0 \leq i \leq N$, the minimum number of rewrite rules needed to rewrite the subtree to r or m, with a register usage at most i. Applying the dynamic programming technique, this leads to a three pass algorithm. The first pass does a bottom-up traversal of the tree to compute the states. The second pass does a top-down traversal to extract the rewrite sequences which are then reordered in a final bottom-up pass.

The algorithm of Aho and Johnson has many similarities to our algorithm for solving fixed goal C-REACHABILITY. The main disadvantages of the algorithm when compared with our own are:

---

[20] This requires assuming only one class of registers

(D.1) The state in [AhJ76] does not contain any information that can be used by a pattern matcher. Thus the decomposition of the problem is done by trying all the possible rewrite rules and, for each one, reaching "down" the tree at different depths depending on the shape of the rewrite rule being used. In contrast δ-LR graphs allow a direct decomposition.

(D.2) The rewrite rules accepted by [AhJ76] are more restricted than those accepted by BURS theory.

(D.3) Explicit cost information needs to be stored in the states of [AhJ76]. δ-LR graphs do not contain explicit cost information.

The algorithm of Aho and Johnson has some advantages over our algorithm:

(A.1) The theory described in this chapter assumes an infinite number of registers, while [AhJ76] can deal with the additional restriction of a finite number of registers as long as there is a single class of registers.

(A.2) The cost function of Aho and Johnson can be more complex than ours, using the full power of the dynamic programming method.

PCC2

The original algorithm of [AhJ76] was implemented in PCC2 [Hen82], a research version of the portable C compiler code generator. The implementation assumed an infinite number of registers available (thus simplifying the notion of state and removing advantage (A.1)) but still used an ***ad-hoc*** technique for recognizing the applicability of instructions.

Top-Down Pattern Matchers

A significant performance improvement in the above techniques was obtained in [AGT86] by using a top-down pattern matcher. Similar approaches have also been developed by Wilhelm and Weisgerber [WeW86] and by Henry and Damron [HeD87]. In these implementations it is still necessary to represent the states by explicitly mentioning both their (sub)patterns and their costs. This leads to slow code generation algorithms.

Bottom-Up Pattern Matchers

Top-down pattern matchers are slower than bottom-up pattern matchers. The recent work of Henry and Damron [HeD87], of Wilhelm and Weisgerber (a preliminary description is in [WeW86]), and of Moencke [Moe87][21] are modifications to the states mentioned above to use bottom-up pattern matching. The rewrite systems accepted by the system reported in [HeD87] can only contain instruction fragment rewrite rules. Their states are similar to the LR graphs of Chapter 5 for that class of rewrite systems, but UI LR graphs are not contemplated. The states can be precomputed at table generation time and then computed at code generation time using a B-fsa because there is a finite number of states. Cost information is encoded essentially as in the full cost LR graphs mentioned elsewhere in this chapter. This forces the implementation of the state to describe the cost information explicitly and to compute it at problem solving time. Chapter 8 quantifies the time penalty involved.

Henry and Damron observe, in [HeD87], that, for a given cost function, it is possible to determine that some paths of the LR graphs will never be useful. They use this analysis to remove some of the edges and nodes from the LR graphs. One advantage of the use of δ-UI LR graphs over this technique is the larger amount of information available statically which leads to a much better job in removing unneeded alternatives and produces smaller and fewer states.

---

[21] That work has been done independently of the research presented in this dissertation.

Hatcher and Christopher

Hatcher and Christopher claim in [HaC] that it is possible to perform optimal unconstrained code generation with an algorithm that would work in time linear in the size of the input operator tree with a small and instruction set-independent constant of proportionality. The article starts with a description of the (solving-time) code generation algorithm, which is similar to the one used in fixed goal REACHABILITY. But the article's (solver-generation time) code generator generation algorithm is quite confusing. Actually, the research on BURS systems in this dissertation started as an answer to the puzzle created by their code generator generation algorithm.

The main problem with [HaC] is that it tries to get along with too little in its states. The author's current understanding of the notion of state in [HaC] is that of a UI LR graph with "locally optimal" edges; that is, whose paths correspond to the minimum-cost local rewrite sequences. In addition [HaC] has additional constraints that relate to the computation of the encoding of the B-fsa as a row and column folding. The example of Figure 6.4 shows that, even for simple machine grammars, it is necessary to encoding some type of cost information. Hence, the approach of [HaC] is doomed for many machine grammars. [HaC] contains an pessimistic algorithm that is supposed to determine at table construction time when non-optimal code may be generated. [HaC] contains examples where optimal code cannot be generated using this technique, but where new rewrite rules can be added to alleviate the problem, but neither [HaC] nor [Hat85] characterize under what circumstances this can be done.

CHAPTER 7

X-Patterns and Projection Systems

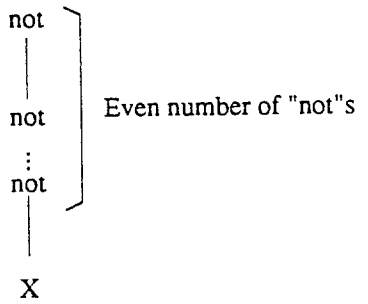
Use it up, wear it out;
Make it do, or do without.

[New England maxim]

This chapter collects two different applications of REACHABILITY. The first application (Section 7.1) is PATTERN MATCHING for typed X-patterns. The second application is in a new type of tree transducer called a ***projection system***; the transducer and its properties are developed in Section 7.2.

PATTERN MATCHING

A pattern defines a set of trees: those at which the pattern matches. Chapters 3 and 4 study PATTERN MATCHING for linear and non-linear N-patterns. The patterns of those chapters are untyped patterns, that is, patterns in which the values to be assigned to the variables are not constrained in any way. This limits the expressiveness of the patterns. For instance, there is no untyped pattern that matches at all trees, and only at those, of the form:



In contrast, typed patterns are expressive enough for the task. The above set of trees can be described by an N-pattern "X" where the type of X is a recognizable set L of trees whose top part contains chains of nots of even length. It can also be described by the X-pattern "$X(Y)$", where the type of X is a recognizable set L' of trees that are chains of nots of even length. The two patterns will be denoted by "$X:L$" and "$X:L'(Y)$" respectively.

Any recognizable set of trees L can be trivially described as the set of trees matching the pattern "$X:L$". This fact suggests dispensing completely with patterns and using, instead, B-fsa specifying recognizable sets. Yet, there are two differences between using a (perhaps typed) pattern and using a B-fsa.

The first difference is that a pattern specifies not only the set of trees at which it matches, but also one or more variable assignments for each of these trees. These assignments "separate" the trees into named components that can be used by the application containing PATTERN MATCH-

ING as a subproblem. For instance, if L' is as above, the removal of useless operator chains[22] could be specified as "$X{:}L'(Y) \rightarrow Y$".

The second difference is that patterns are much more concise and readily grasped by the human reader. In some real sense, patterns can be seen as a higher specification that is implemented as a B-fsa (or, more precisely, a LB-fsa).

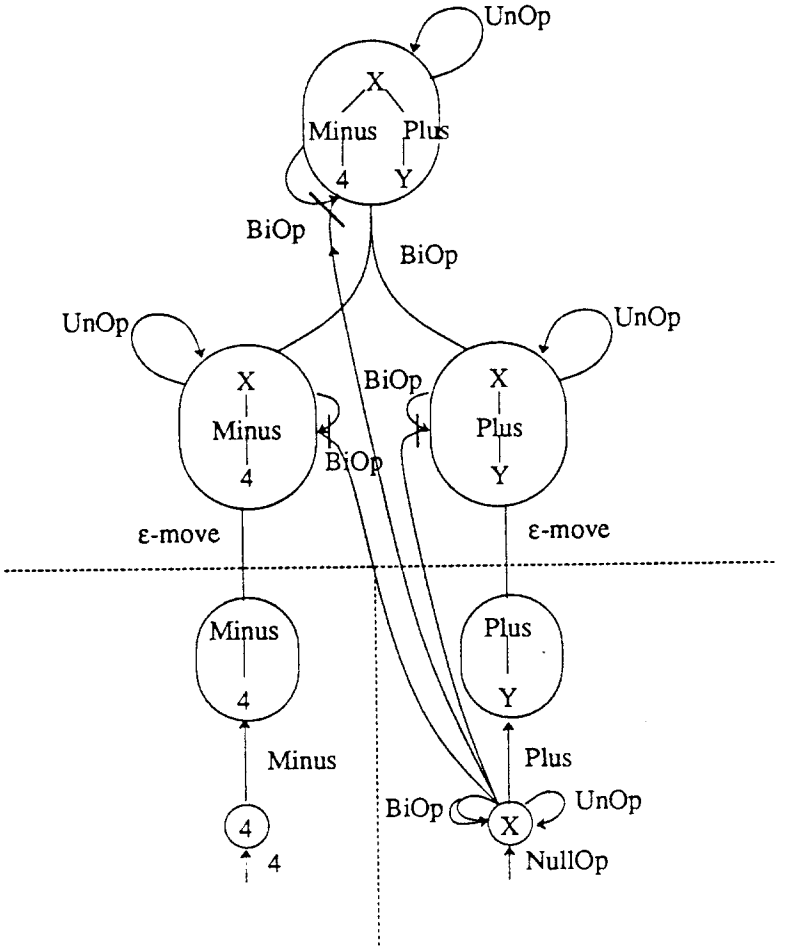
The simplest extension to untyped N-patterns are typed N-patterns. This chapter will only consider linear typed N-patterns; PATTERN MATCHING for non-linear typed N-patterns can then be solved using the approach described in Chapter 4. PATTERN MATCHING for typed N-patterns can be solved by constructing an LB-fsa that will label a node in a subject tree accordingly to whether the pattern matches at the node or not. The LB-fsa can be found by taking the LB-fsa of the untyped version of the pattern and "pasting" into it the B-fsa representing the types of its variables. Section 7.1.1 shows how this is done.

The general idea for solving PATTERN MATCHING for **untyped** X-patterns is still to use some algorithm based on a bottom-up traversal of the subject tree to find match sets, but the situation is more complex because it is non-trivial to find a variable assignment; indeed, there may be more than one assignment valid at a given node.

Forgetting for a moment the variable assignment, PATTERN MATCHING could be solved by pasting together LB-fsa "solving" portions of the pattern. For instance, Figure 7.1 shows a non-deterministic LB-fsa that recognizes the set of trees matched by the pattern "$X(Minus(4),Plus(Y))$". That figure uses a slight extension to the notation used in previous chapters to describe LB-fsa. The symbols "NullOp", "UnOp", and "BiOp" are used to represent any nullary, unary, and binary operators; "ε-moves" indicate transitions that can occur independently of the input; and those transfers that are symmetric are indicated by a short stroke in a thicker pen across the transfer edges. The figure is split into 3 different portions. The lower two correspond to the LB-fsa that recognize $Minus(4)$ and $Plus(Y)$, and the upper one represents the X-variable. The LB-fsa shown is non-deterministic, but the algorithms of Section 2.4.1 show how to obtain from it a deterministic LB-fsa.

---

[22] "Useless", of course, depends on the semantics of the operators. For instance, C hackers will promptly point out that $!!e$ is not equal to e.



A LB-fsa for Untyped X-Pattern Matching

Figure 7.1

One could construct a LB-fsa for any untyped X-pattern following the same approach of "pasting together" portions of patterns, but this would not solve the problem of finding the variable assignment. One possible solution to this problem would be to extend the state associated with each node in the tree with some representation of the (partial) variable assignments. Instead, the solution used in this chapter is to recycle the techniques of previous chapters and use the solution to REACHABILITY or, more precisely, UCODE, to solve this problem.

A rewrite system and a goal tree describe a set of trees: those that the rewrite system can rewrite into the goal. Any recognizable set can be described using this mechanism by constructing a rewrite system in finite-BURS emulating the B-fsa of the recognizable set. Rewrite systems can describe non-recognizable sets, but, if the rewrite system belongs to finite-BURS, the two mechanisms have the same descriptive power. Despite this equivalence, rewrite systems are

more convenient than B-fsa because they are descriptions at a higher level (in the same sense in which patterns are at a higher level than B-fsa) and because they provide convenient "hooks" for the placement of semantic actions.

The idea to solve PATTERN MATCHING for untyped X-patterns, is to define a rewrite system that performs in a way similar to the LB-fsa hinted in Figure 7.1. This rewrite system contains some rewrite rules whose semantic routines are used to store information that is used to record the root and the frontier of the subtrees in the subject tree that are associated with the X-variables in the pattern. The algorithm for PATTERN MATCHING now follows the three-pass algorithm for UCODE: the first, bottom-up, pass assigns an LR graph to each node of the subject tree from which we can extract the collection of patterns matching at the node. The second pass, top-down, can now be done for each pattern in the collection to find the root and frontier of the subtrees associated with the X-variables. Finally, the third pass, bottom-up, can be seen as actually collecting the root and frontier information into the variable assignment. Repeated applications of passes two and three could produce the variable assignments of all the patterns, but typical applications will require the assignment for only one pattern.

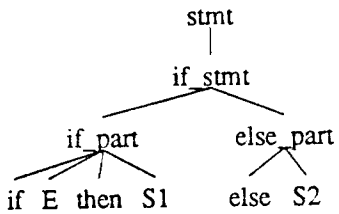
The approach sketched above and formalized further in Section 7.1.2 has the obvious advantage of recycling the technology developed for UCODE. It also has the advantage that, since most applications of pattern matching do not require the variable assignments of all the patterns, it will run substantially faster than an approach that tracks the variable assignment as it traverses the tree bottom-up. Finally, it can be extended to deal with typed X-patterns, as is shown in Section 7.1.3.

Projection Systems

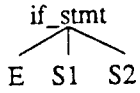
The intuitive notion of "projecting" a input tree is to "split" the tree into parts and to consistently replace these parts by new tree fragments which, when "pasted" together, will produce an output tree. The simplest instance of this notion of projection is the relabeling of Def. 2.17. Relabelings can be used to describe simple tree transformations, and can also be used to describe RECOG sets as relabelings of local sets (Proposition 2.10). As an example of this last proposition, the tree language composed of the trees of the form: is not the set of derivation trees for any context free grammar, but can be obtained as the relabeling of one such set. In particular, it is the result of the relabeling of e_1 by e and e_2 by e in the derivation trees of the context-free grammar having the rewrite rules:

$$e_1 \rightarrow e_2\ pepe$$
$$e_2 \rightarrow e_2\ foo$$
$$e_2 \rightarrow bar$$

There are many tree transformations that correspond to our intuitive notion of a "projection" but cannot be described using a relabeling. Linear homomorphisms (Def. 2.18) are one generalization of relabelings in which an operator in the input tree can be replaced by a subtree. Still, neither relabelings nor linear homomorphisms can describe a transformation that requires "context" information. For instance, one would want to be able to rewrite a tree like:



into



where E, $S1$, and $S2$ are variables that will match trees representing expressions and statement lists.

The notion required to describe such types of "projection" is that of a ***projection system***. A projection system is a rewrite system that is applied in a different way. The key concept is the idea of a "cover", a non-overlapping collection of matchings of the input patterns in the system completely covering the input tree. Given a cover for a tree, its projection is obtained by replacing each input pattern by its associated output pattern.

Finding the cover can be done by solving a REACHABILITY problem, thus providing an algorithm to compute the projection. Similarly, the results of Chapter 5 can be used to determine if any tree in a recognizable set can be covered. Moreover, the function defined by the projection system can be "inverted", that is, given an output tree it is also possible to use the solution to another instance of REACHABILITY to obtain an input tree belonging to a recognizable set that projects into the output tree.

One application of projection systems is to describe the transformation between **concrete** and **abstract** syntax trees. The concrete syntax of a programming language is the syntax used by the programmer, normally described by a context-free grammar. The concrete syntax tree for a program is the derivation tree for the program. Since the concrete syntax grammar is used to direct the compilation process, it may contain several constructs (new non-terminals, additional productions) that are irrelevant to characterize the "deep structure", or meaning, of the program. The ***abstract syntax*** of the language only reflects this deep structure, making it a more convenient representation for semantics, both "static" and "dynamic".

Some tree mappings between concrete and abstract syntax trees can be described using a projection system. The application of the projection defines the mapping from concrete to abstract trees, while the application of the inversion algorithm provides an mapping between abstract and concrete trees. An application of these mappings could be in a language-based editor like PAN [BVG87], where the abstract syntax tree is used internally to save space but the concrete syntax tree is required for incremental parsing or for pretty-printing; see [BBG87] for another approach to this problem.

Another application of the "inversion" algorithm is to allow more flexible ways to define recognizable sets: as images of local sets under projection systems instead of only under relabelings.

7.1. Typed Patterns

The goal of this section is to show how to solve PATTERN MATCHING for typed X-patterns. Before attacking the general case (Section 7.1.3), the section first solves two useful

simplifications: typed N-patterns (Section 7.1.1) and untyped X-patterns (Section 7.1.2). The simpler cases serve as an introduction to the more complex one as well as being useful in their own right.

7.1.1. Typed N-patterns

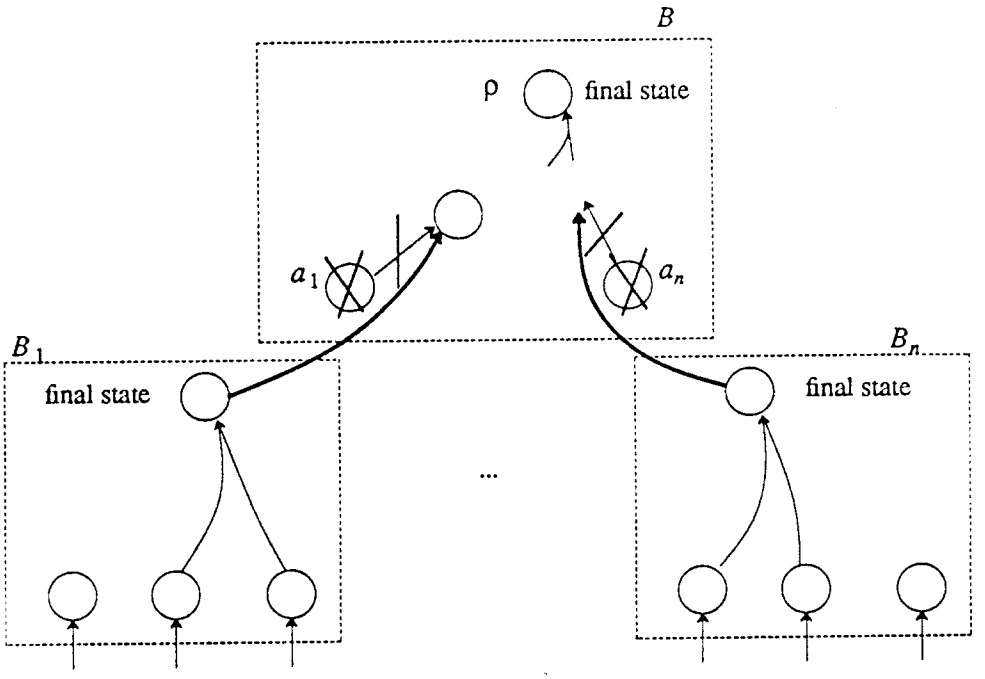
As indicated in the introduction, the idea used here is quite simple: graft the B-fsa describing the types of the variables into the B-fsa describing the untyped N-pattern. The key proposition is:

Proposition 7.1 *Let* ρ *be a typed linear N-pattern over* ***Op****, with the types in* ρ *being described by B-fsa over* ***Op****. There is an algorithm that will generate a (deterministic and minimal) LB-fsa* $\langle A,L \rangle$ *such that for every subject tree T over* ***Op*** *a node in T is labeled with* $\{\rho\}$ *if* ρ *matches at the subtree rooted by the node, and is labeled with* $\emptyset$ *otherwise.*

Proof We first produce a non-deterministic LB-fsa with the desired properties. The LB-fsa can then be converted into a deterministic one and minimized using the appropriate algorithms of Section 2.4.1.

Denote the typed variables in ρ by $X_1, \ldots, X_n$, and let ρ' be $\rho_{X_1 \leftarrow a_1 \cdots X_n \leftarrow a_n}$ where $a_1, \ldots, a_n$ are new nullary operators not in ***Op***. Clearly, there is a correspondence between trees over ***Op*** matched by ρ and trees over $Op \cup \{a_1, \ldots, a_n\}$ matched by ρ'. Now construct a match set LB-fsa for ρ', B. Let $B_1, \ldots, B_n$ be the n B-fsa that recognize the types associated with $X_1, \ldots, X_n$. Assume that $B_1, \ldots, B_n$ have a single final state and that the states in B, $B_1, \ldots, B_n$ are all mutually disjoint. We construct a new (non-deterministic) LB-fsa B' from B, $B_1, \ldots, B_n$. The states in B' are all the states in the B-fsa except for the states in B corresponding to the nullary operators $a_1, \ldots, a_n$. The transfer function in B' is the union of those in the B-fsa, except that those transfers that used to come from a state a_i, now come from the final state of B_i. The labeling function for B' assigns ρ to the states in B' that correspond to states in B that are labeled ρ, and $\emptyset$ to the remaining states. □

The situation discussed in the proof of Proposition 7.1 is sketched in Figure 7.2.



An LB-fsa for Typed N-patterns

Figure 7.2

Given a collection of typed N-patterns, we can construct an LB-fsa for each one of them and then collect them into a non-deterministic LB-fsa that will recognize the collection. If desired the LB-fsa can then be converted into deterministic form and minimized. Thus:

Proposition 7.2 *Let F be a collection of typed linear N-patterns over* **Op** *with the types given as B-fsa over* **Op**. *There is an algorithm that will generate a (deterministic and minimal) LB-fsa* $\langle A,L \rangle$ *such that the label of a node in a tree contains* $\rho \in F$ *if and only if* ρ *matches at the subtree rooted by the node.*

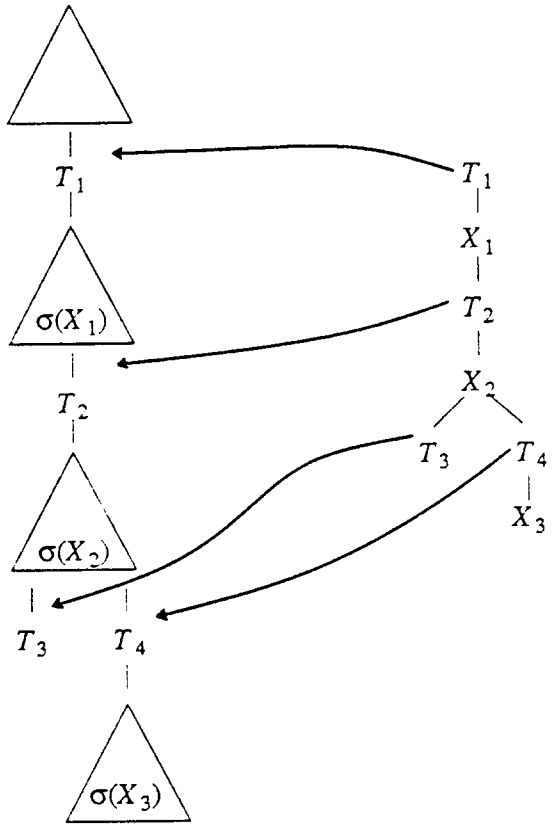
7.1.2. Untyped X-patterns

This subsection deals with linear untyped X-patterns. The introduction of types is studied in the next subsection. Again, we restrict ourselves to linear patterns since solving PATTERN MATCHING for X-patterns where an X-variable appears more than once is quite difficult. In particular, solving this problem would require comparing two internal portions of the subject tree which, unlike the situation for N-patterns, cannot be solved by simple pointer equality in a computation dag.

This section solves PATTERN MATCHING by solving a particular class of REACHABILITY problems. The idea is to construct for each X-pattern a rewrite system that contains symbols to indicate that certain parts of the pattern have been recognized, including one for the complete pattern. The rewrite rules encode the conditions under which the given parts are recognized, and a

tree can be rewritten into a symbol representing a pattern if and only if the pattern matches at the tree.

If ρ is a match of an X-pattern at a subject tree, ρ assigns to each n-ary variable in the X-pattern a linear N-pattern with n variables. This N-pattern can be characterized by $n+1$ pointers to nodes in the subject tree: one corresponding to the root of the N-pattern, the others to the location of its n variables. The root pointer does not need to be represented explicitly: saying that the X-pattern matches at some subtree is equivalent to saying tha the root pointer points to the root of that subtree. This is why we have not mentioned any explicit pointers when discussing pattern matching for N-patterns. But for X-patterns, the remaining n-pointers are required to uniquely determine the assignment to the variable. Figure 7.3 sketches the situation for an assignment ρ that is a matching for the X-pattern (shown at the right of the figure) at the subject tree (shown at the left of the figure). In the figure, T_i stand for fixed trees, X_i for variables, and $\sigma(X_1)$, $\sigma(X_2)$, and $\sigma(X_3)$ are the trees (N-patterns) assigned to X_1, X_2, and X_3 by the matching.



Representing a Variable Assignment

Figure 7.3

The assignment of the matching can be found by associating a semantic action with the rewrite rules "outlining" the lower boundary of the X-variables. These semantic actions save the

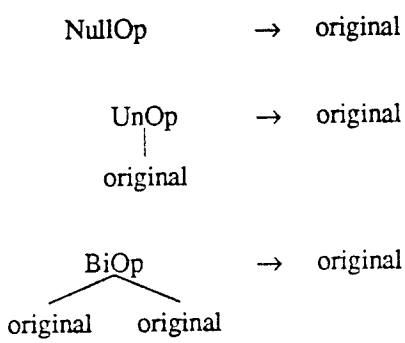
pointer to the node in the subject tree currently being visited at matching time.

Proposition 7.3 *Let* ρ *be an untyped (linear) X-pattern over Op. There is a rewrite system* R_ρ *and a goal symbol* G_ρ *such that a tree T can be rewritten by* R_ρ *into* G_ρ *if and only if* ρ *matches at T.*

In addition, R_ρ *has n distinguished rewrite rules for each n-ary variable in* ρ. *If* ρ *matches at T, then any normal form rewrite sequence in* R_ρ *from T to* G_ρ *contains exactly one of each of these distinguished rewrite rules. The input nodes at which the rewrite rules are applied define a variable assignment for* ρ, *and any variable assignment corresponds to some normal form rewrite sequence in* R_ρ.

Proof The construction of the rewrite system is given. It is left to the reader to show, using structural induction on the patterns, that it satisfies the proposition. The description only contains the cases of 0,1,2 operators. Larger arities can be obtained either by describing them using the lower arities (e.g. $X(\rho_1,\rho_2,\rho_3)$ becomes $X(\rho_1,X(\rho_2,\rho_3))$, or by direct extension of the technique). *NullOp*, *UnOp*, and *BiOp* should be replaced by all nullary, unary, and binary operators in *Op*[23].

There is a symbol "*original*" that will represent a portion of the subject tree on which no interesting rewrite rules can be applied. The rewrite rules associated with it are:



None of the remaining rewrite rules will introduce either *original* or an operator in *Op*. Thus, a tree can be rewritten into *original* only if no other rewrite rule is used.

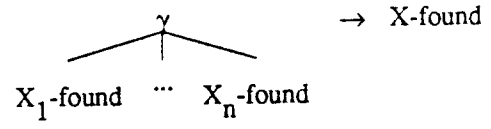
The next rewrite rules recognize portions of the subpattern. They use several symbols for each variable X in ρ: *X-found* indicates that the subpattern of ρ rooted by X matches at the subtree, and, if X is a binary variable of the form $X(\rho_1,\rho_2)$, *X-left* and *X-right* respectively mean that the patterns $X(\rho_1)$ and $X(\rho_2)$ match.

If X is a subpattern (that is, X has arity 0), then there is a rewrite rule of the form

original → X-found

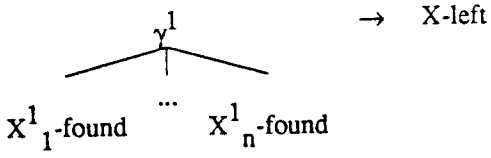
If $X(\rho_1)$ is a subpattern, and ρ_1 is of the form $\gamma(\delta_1, \ldots, \delta_n)$ where γ has no variables in it and, for $1 \le i \le n$, δ_i has a variable X_i at its root, then

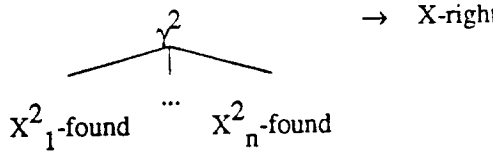
[23] An alternative is to make them symbols and use a generic operator rewrite.



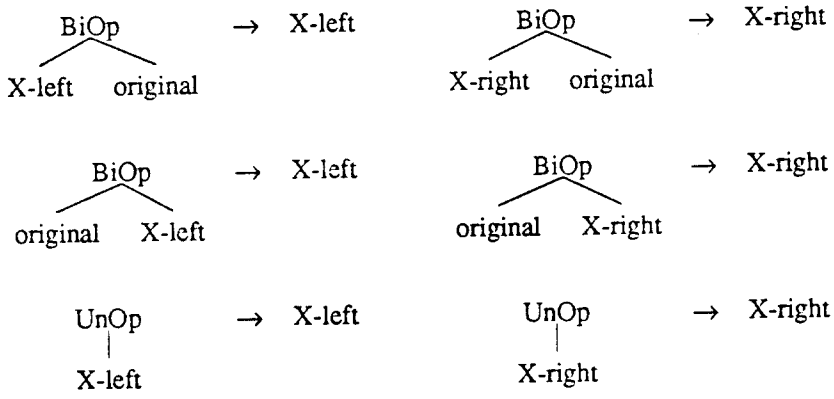
This rewrite rule is distinguished and will have a semantic action associated with it.

If $X(\rho_1,\rho_2)$ is a subpattern, ρ_1 is of the form $\gamma^1(\delta^1_1,\ldots,\delta^1_n)$, ρ_2 is of the form $\gamma^2(\delta^2_1,\ldots,\delta^2_n)$ where γ^1 and γ^2 have no variables, and δ^j_i $(1\leq i\leq n)$ each has a variable X^j_i at its root, then

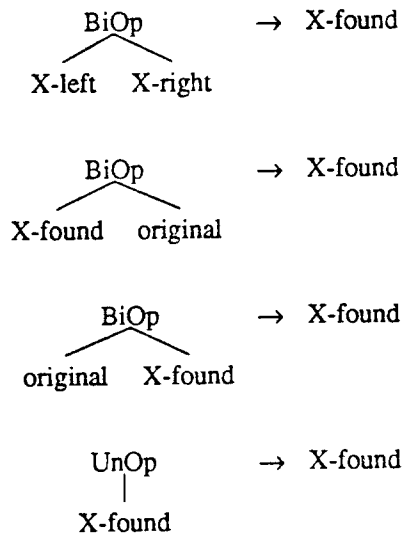




These two rewrite rules are distinguished and have a semantic action associated with them to record the tree position at which they are invoked. In addition there are also rewrite rules of the form

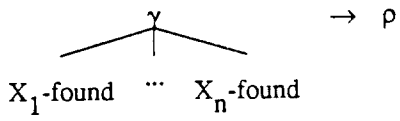


And, finally, for any X-variable, we have



The definitions guarantee that a tree can be rewritten into *X-found* if and only if the subtree of ρ rooted by X matches at the tree.

The rewrite system can be completed with a final rewrite rule corresponding to the top of the original pattern ρ. If ρ can be decomposed as $\gamma(\delta_1, \ldots, \delta_n)$ where δ_i is rooted by a variable X_i, then there is a rewrite rule of the form:



where ρ is a symbol representing the matching of the pattern ρ.

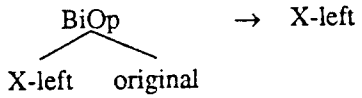
Note that the distinguished rewrite rules detect the variable assignments. □

The rewrite system can now be used to solve PATTERN MATCHING by associating with all the rewrite rules the *null* semantic function except for the distinguished rewrite rules finding the variable assignment and solving UCODE. The algorithm for UCODE can be specialized further: the second, top-down, phase can be modified so that whenever *original* is a goal, the traversal is stopped. Solving UCODE instead of REACHABILITY has the advantage of yielding substantially smaller tables.

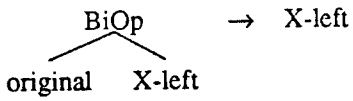
There is no problem in dealing with a collection of patterns: just add new rewrite rules for the new subpatterns. Subpatterns that appear in more than one pattern produce only one set of rewrite rules. Consequently:

Proposition 7.4 *Let F be a collection of untyped X-patterns. There exists an algorithm that will solve PATTERN MATCHING (except for finding the variable assignment) for F in time linear in the size of the subject tree. If the variable assignment is wanted, the extra time required is linear in the size of the subject tree for each desired pattern.*

The technique shown above also allows a simple, but probably useful, modification. When constructing the UI LR graphs one can prefer, whenever possible, some rewrite rules instead of others. Thus, if



is selected always before

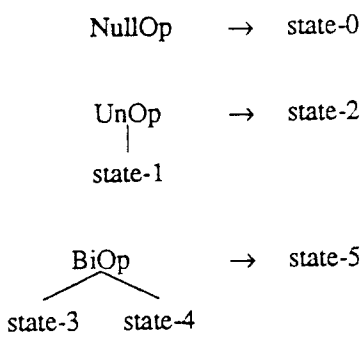


the left subtree of a binary X-pattern is chosen "as far to the left" as possible. This might be useful to reduce some of the ambiguity inherent in X-patterns. The next stage after this is to require that the subtree associated with X must have some particular shape, that is, typed X-patterns.

7.1.3. Typed X-patterns

This section, finally, deals with the more complex case. The idea is simply to take the rewrite system used in the proof of Proposition 7.3 and encode the transitions of the B-fsa describing the types into it.

The types of the nullary variables can be dealt with pretty easily. Assume that the B-fsa for these types have disjoint states and are deterministic. The rewrite rules for *original* are replaced by collections of rewrite rules of the form

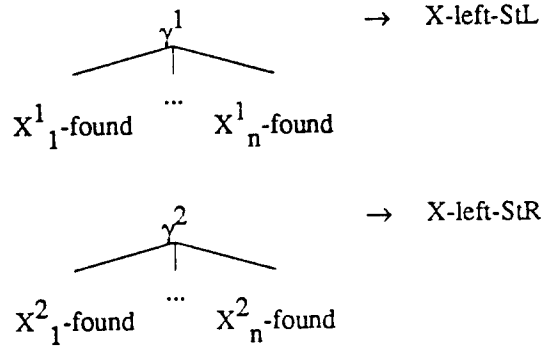


where *NullOp*, *UnOp*, and *BiOp* are instantiated to be specific nullary, unary, and binary operators, and *state-0* ,..., *state-5* are new symbols corresponding to the states in the B-fsa of the types related by *state-0* = *f(nullop)*, *state-2* = *f(unop,state-1)*, and *state-5* = *f(biop,state-3,state-4)*. Then, if *state* is a state in a B-fsa B, a tree can be rewritten into *state* if and only if B associates *state* with the tree.

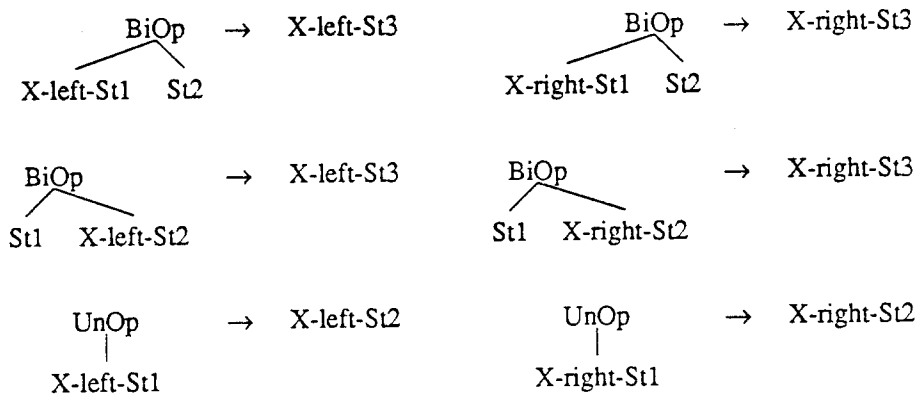
Dealing with the types of the X-variables is similar. Recall (Def. 2.3) that the type of an n-ary X-variable is a recognizable set of trees using the operator set extended with n nullary operators, $child_1, \ldots, child_n$ representing the "slots" for the children of the X-variable. Thus, for each typed X-variable we can assume the existence of a deterministic B-fsa over $Op \cup \{child_1, \ldots, child_n\}$. The types of these B-fsa are then encoded into the rewrite rules that are associated with each X-variable.

Only the case for binary X-variables is shown; unary X-variables are similar. For each typed binary X-variable X and each state st in its B-fsa, there will exist symbols *X-left-st*, *X-right-st*, *X-found-st*, and *X-found*. The first three encode the fact that the tree can be rewritten into

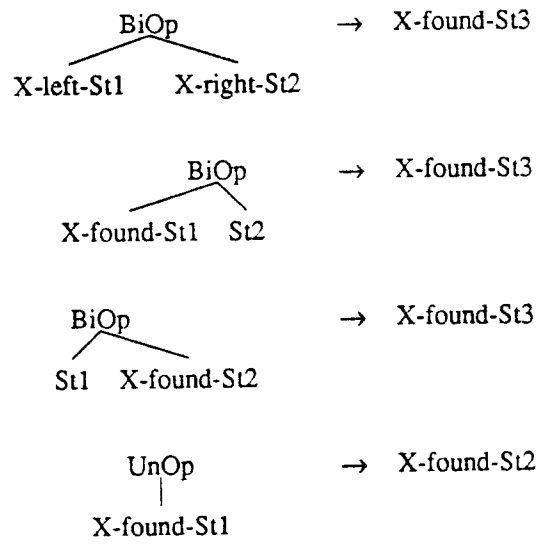
the *X-left*, *X-right*, and *X-found*, of untyped X-patterns where the subtree assigned to the X-variable has state *st* under the B-fsa. The symbol *X–found* encodes the fact that the tree can be rewritten into the untyped *X-found* with a final state. These states are computed by simulating the transitions of the B-fsa with the rewrite rules. Thus, if the type is described by a B-fsa B with transfer function f, one would have:



where $StL=f\ (child_1)$ and $StR=f\ (child_2)$. The values are then propagated by rewrite rules like the ones below, where the states satisfy $St\,3=f\ (biop\ , St\,1, St\,2)$ and $St\,2=f\ (unop\ , St\,1)$.



And:



And, if St is the final state in B, finally used as follows:

$$\text{X-found-St}F \rightarrow \text{X-found}$$

The approach to solving PATTERN MATCHING is identical to the one used in untyped X-patterns: solving a UCODE problem for this rewrite system, although many rewrite rules will be frequently found useless if the LB-fsa computing the UI LR graphs is minimized. Thus, we have:

Proposition 7.5 *Let F be a collection of typed X-patterns, with the types of the variables provided by a collection of B-fsa. There exists an algorithm that will solve PATTERN MATCHING (except for finding the variable assignment) for F in time linear in the size of the subject tree. If the variable assignment is wanted, the extra time required is linear in the size of the subject tree for each desired pattern.*

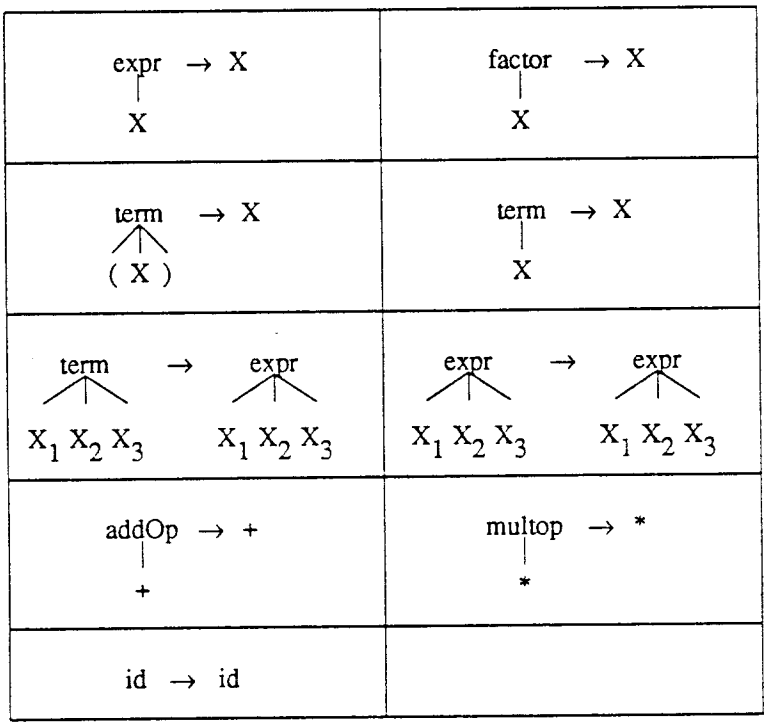
Applications to Rewrite Systems

The pattern matching algorithms presented in this section can be used in an algorithm dealing with rewrite systems. In particular, they could be used in a rewrite system for which REACHABILITY is being solved. Postponing a deeper analysis of the implications for future work, it seems that typed N-patterns and both untyped and typed X-patterns can be used as input patterns in rewrite rules without interfering with the basic principles behind the theory used in Chapter 5 to solve REACHABILITY. In any case, output patterns are still restricted to be untyped N-patterns: adding types to the variables is meaningless, and dealing with X-patterns seems to easily lead to rewrite systems outside BURS.

7.2. Projection Systems

Projection systems are tree transducers that are a generalization of linear homomorphisms and relabelings. The generalization allows the system to use context information in determining how to "project" a portion of the input tree. Formally, a projection system is identical to a rewrite system, but it is applied in a different way.

An example of a use of a projection system is the one shown at the beginning of this chapter involving "if-then-else" statements. Another application could be the specification of the mapping between concrete and abstract representations of expression trees. For example the rewrite system of Figure 7.4 describes how portions of a parse tree for the context-free grammar of Figure 7.5 would be modified. The rewrite rule $expr(X_1,X_2,X_3) \rightarrow expr(X_1,X_2,X_3)$ reflects the fact that this case for *expr* is left unmodified. This rewrite rule should be used only if no other rewrite rule applies and, in particular, if $expr(\text{"("}, X, \text{")"}) \rightarrow X$ does not apply. This constraint on the application of the rewrite rules is formalized below as the "more coarser than" notion. The rewrite rule is required to satisfy that the input patterns of the projection system *cover* (Definition 7.1) the input tree.



Example of a Projection System

Figure 7.4

```
expr   →term addop expr
expr   →term
term   →factor mulop term
term   →factor
factor →id
factor →"(" expr ")"
addop→"+"
mulop→"*"
```

Example of a Context-Free Grammar

Figure 7.5

The application of the projection system to an input tree first finds a "partition" of the input tree using the input patterns of the rewrite rules in the projection system, and then replaces them by the output patterns. The formal notion involved is called a **cover**.

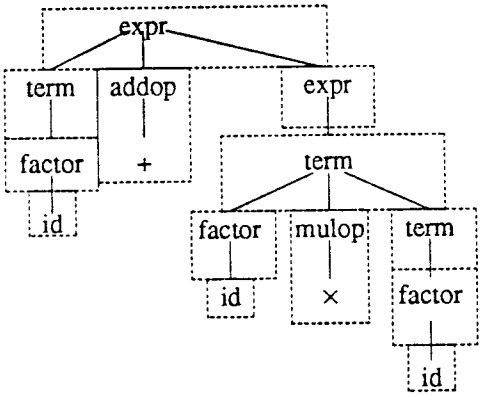
Definition 7.1 *Let **F** be a set of N-patterns over **Op**, and let T be a tree over **Op**. A **cover** for T using **F** is a function assigning to each position p in T either the distinguished value* **nil** *or a pair* $\langle \rho, \sigma \rangle$ *where* σ *is a match for the pattern* ρ *in **F** at the subtree* $T_{@p}$*, and such that:*

*(1) The cover assigns a non-*nil *value to the root of T ;*

(2) If the cover assigns $\langle \rho, \sigma \rangle$ *to position p , and the positions of the variables in* ρ *are* $p\,1, \ldots, p_n$*, then the cover assigns non-*nil *values to the positions* $p/\!/p\,1, \ldots, p/\!/p_n$*; and*

*(3) The only positions to which the cover assigns non-*nil *values are those indicated by (1) and (2) above.*

For example, Figure 7.6 shows a cover of a tree in the local set induced by the grammar in Figure 7.5 using the input patterns of Figure 7.4.



Example of a Cover

Figure 7.6

Definition 7.2 *Let **F** be a set of patterns over **Op**, let T be a tree over **Op**, and let C_1 and C_2 be two covers for T on **F**. C_1 is* **immediately coarser** *than C_2 if there is a position p in T such that, for every position q different from p, if C_1 assigns a non-***nil** *value to q, then the pair assigned is identical to the one assigned to q by C_2, The relation* **coarser**, *denoted $>_c$, is the transitive closure of "immediately coarser than".*

A cover for T is a **maximal** *cover if it is maximal over $>_c$.*

Note that, since both C_1 and C_2 in Definition 7.2 are covers, the pair assigned by C_1 to the position p is, in some sense, "covering" the same portion of the input tree covered by the remaining non-**nil** values of C_2.

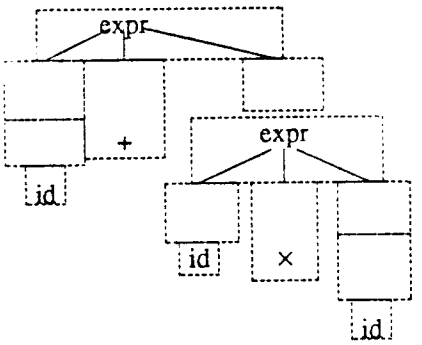
Maximal covers are useful because larger patterns in a projection system indicate a more precise specification.

Definition 7.3 *A* **projection system** ***P*** *over **Op** is a rewrite system with rewrite rules $\alpha \rightarrow \beta$ where (1) α and β are both linear patterns with exactly the same variables, and (2) for every tree T, there is a cover for T on $\{\alpha \mid \alpha \rightarrow \beta \in P\}$.*

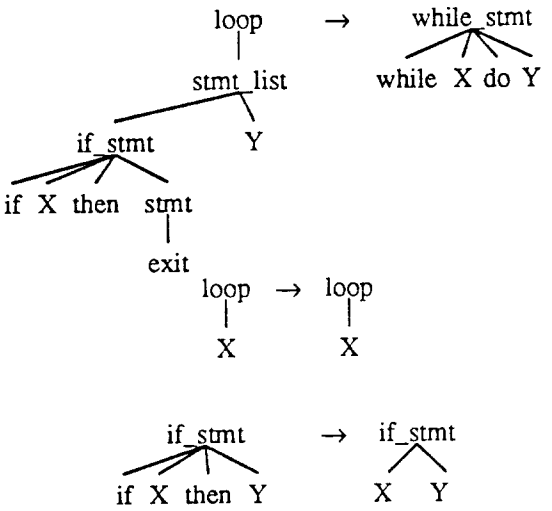
*The result of applying **P** to T is a new tree $h(P,T)$ defined recursively as follows: if P assigns to $\alpha(T_1, \ldots, T_n)$ a pattern $\alpha(X_1, \ldots, X_n)$, $h(P, \alpha(T_1, \ldots, T_n))$ is defined to be $\beta_{X_1 \leftarrow h(P,T_1) \cdots X_n \leftarrow h(P,T_n)}$*

A **good** *projection system for some recognizable set **L** is one such that each $T \in L$ has a single maximal cover.*

The cover in Figure 7.6 is a maximal cover for the set of input patterns in Figure 7.4. As such, it defines a projection of the input tree, namely, the one shown below:



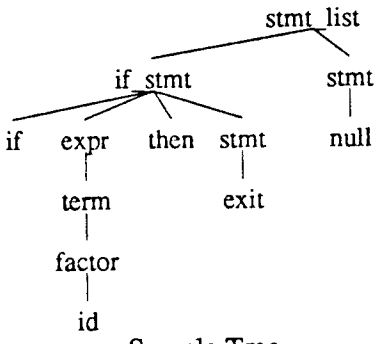
Finding a maximal cover is more complex for some projection systems than for others. Using the same example of the mapping between concrete and abstract syntax trees, one might want to add the rewrite rules in Figure 7.7 to those of Figure 7.4. As always, *X* and *Y* are variables.



Second Example of Projection System

Figure 7.7

In this case, there is more than one possible cover for the tree of Figure 7.8. Following the definition of projection system, a maximal cover should be used.



Sample Tree

Figure 7.8

The projection system containing the rewrite rules of Figure 7.4 together wih the rewrite rules of Figure 7.7 still satisfies the property that there is a single maximal cover for every input tree. Determining whether this and other properties involving projection systems hold can be answered using results and techniques from REACHABILITY theory. The first question is how to compute covers.

Proposition 7.6 *Given a set F of patterns over Op, and a tree T over Op, there is an algorithm that finds all the covers on T by Op.*

Proof For any pattern α with n variables $X_1, \ldots, X_n$, we will use $\alpha(\perp)$ to denote the replacement of all the variables with $\perp$, that is, $\alpha_{X_1 \leftarrow \perp \cdots X_n \leftarrow \perp}$. For any set of (linear) patterns F, $F(\perp)$ will be used to denote the rewrite system over $Op \oplus \{\perp\}$, where the rewrite rules are of the form $\alpha(\perp) \rightarrow \perp$. $F(\perp)$ is a simple reduction system, and, thus, in BURS. Moreover, a reduction corresponds to finding a cover. An invocation of *fixed_goal*$(T, \perp)$ will find all the covers of T over F. □

The above algorithm can be modified to find good covers.

Proposition 7.7 *Given a set F of patterns over Op, and a tree T over Op, there is an algorithm that will compute a maximal cover for T over F.*

Proof The algorithm is a variation of the one used for the previous proposition. As before we solve fixed-goal REACHABILITY for $F(\perp)$, but now we replace LR graphs by an appropriate restriction. If G is an LR graph of $F(\perp)$, associate with it a restricted LR graph G' where there is an edge between $\alpha(\perp)$ and $\perp$ only if α is maximal under ***subsumes*** over all those patterns β such that $\beta(\perp) \rightarrow \perp$ is in G. Using these restricted LR graphs, a solution to fixed-goal REACHABILITY with goal $\perp$ over tree T produces a cover for T. In addition, it is a maximal cover: otherwise there would be a highest node in T at which a pattern β was chosen when some other pattern β', with $\beta' \geq \beta$, should have been chosen, contradicting the construction of the restricted LR graphs. □

The above algorithm also provides information on whether the maximal cover is unique or not.

Proposition 7.8 *Given a set F of patterns over Op, and a tree T over Op, there is an algorithm that will determine whether T has a unique maximal cover over F.*
Proof There is a unique cover if and only if all nodes encountered during the solving of the fixed-goal REACHABILITY problem using the restricted LR graphs defined in Proposition 7.7 have a single incoming edge. □

The questions addressed in the propositions above can also be asked relative to a recognizable set described through a B-fsa.

Proposition 7.9 *Let F be a pattern set and let L be a recognizable set over Op. Further assume that L is given as a B-fsa. (1) There is an algorithm that will determine whether every tree in L has a cover over F. (2) There is an algorithm that will determine whether every tree in L has a unique maximal cover over F.*
Proof (1) can be solved by solving the blocking problem for $F(\perp)$ and L (Proposition 5.20). (2) can be solved by removing useless nodes in the restriction of the LR graphs defined in the proof of Proposition 7.7, removing useless nodes in them for L using Proposition 5.22, and then checking whether there is any restricted LR graph where $\perp$ has more than one incoming edge. □

Computing projections is a direct corollary of the definition of projection and Proposition 7.7.

Corollary 7.1 *Given a projection system P over Op and a tree T over Op, there is an algorithm that will compute all the applications of P to T.*

If there is a unique maximal cover, Proposition 7.7 and the previous corollary show:

Corollary 7.2 *RECOG is closed under inverse projections.*

Finally, one can determine the shape of the output trees:

Proposition 7.10 *Given a projection system P over Op and two B-fsa B_1 and B_2, there is an algorithm that will determine whether there is any tree t over Op accepted by B_1 such that its projection under P is not accepted by B_2.*
Proof We construct a new rewrite system R from the projection system and from B_2. Assume, without loss of generality, that B_2 has a single final state St_{final}. R will be defined over Op extended with nullary symbols representing the states in B_2. For every rewrite rule in the projection system of the form $\alpha(X_1, \ldots, X_n) \rightarrow \beta(X_1, \ldots, X_n)$ where the variables in α and β are $X_1, \ldots, X_n$, there are rewrite rules in R of the form $\alpha_{X_1 \leftarrow St_1 \cdots X_n \leftarrow St_n} \rightarrow St_0$ where $St_0, \ldots, St_n$ are symbols corresponding to states in B_2, and St_0 is the state that would be assigned by B_2 to a tree of the form $\alpha(X_1, \ldots, X_n)$ if, for $1 \leq i \leq n$, the subtree rooted by X_i had state St_i.

Clearly there are finitely many rewrite rules in R, and they can be computed. The reductions in R track both covers and the state information. By structural induction one can easily prove that there is a rewrite sequence in R from T to St if and only if a projection T' of T under P is assigned state St by B_2. Moreover, R is a reduction system and a member of finite-BURS.

The question in the body of the proposition is equivalent to determining if there is a blocking tree in B_1 for R and goal St_{final}. □

Inverting a Projection

A useful question involving projection systems is the following: given a projection system P and a tree T', determine if there is a tree T that yields T' under P. The answer is very simple if there are no constraints on T:

Proposition 7.11 *Let **P** be a projection system over **Op**. There is an algorithm that will determine for a tree T' over **Op** whether there is another tree T over **Op** such that the projection of T over **P** is T' and, if so, will provide one such tree T.*
Proof There will be such a tree if and only if there exists a cover of T' using the output patterns of **P**. An application of the rewrite rules of **P**, "in reverse" will produce T. □

A more useful version of inversion is when the input tree is required to belong to a recognizable set.

Proposition 7.12 *Let **L** be a recognizable set described using a B-fsa **B**. Let **P** be a projection system over **Op**. There exists an algorithm that will determine, given a tree T' over **Op** whether there is a tree T accepted by **B** such that the projection of T over **P** is T' and, if so, will provide one such tree T.*
Proof The proof is similar to the proof of Proposition 7.10. We construct a new rewrite system R from the projection system and from the B-fsa. Assume, without loss of generality, that B has a single final state St_{final}. R will be over Op extended with nullary symbols representing the states in B. For every rewrite rule in the projection system of the form $\alpha(X_1, \ldots, X_n) \rightarrow \beta(X_1, \ldots, X_n)$ where the variables in α and β are $X_1, \ldots, X_n$, there are rewrite rules in R of the form $\beta_{X_1 \leftarrow St_1 \cdots X_n \leftarrow St_n} \rightarrow St_0$ where $St_0, \ldots, St_n$ are symbols corresponding to states in B, and St_0 is the state that would be assigned by B to a tree of the form $\beta(X_1, \ldots, X_n)$ if, for $1 \leq i \leq n$, the subtree rooted by X_i had state St_i.

Clearly there are finitely many rewrite rules in R, and they can be computed. The reductions in R track both covers and the state information. By structural induction one can easily prove that there is a rewrite sequence in R from T' to St if and only if there is a tree T such that its projection under P is T and its state under B is St.

Since R is clearly a member of finite-BURS, we can construct the UI LR graphs for R and, for any given output tree T', solve the fixed-goal REACHABILITY problem for R with goal St_{final}. A desired T will exist if and only if T' rewrites into St_{final}, and if so, it can be extracted from the rewrite sequence. □

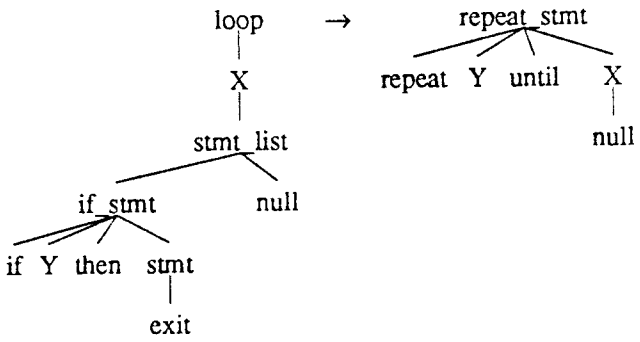
The above proposition shows that the projection of any recognizable set is a recognizable set. In particular, it provides an answer to the question raised in Chapter 2 regarding the "inverse" of Proposition 2.10.

Corollary 7.3 *RECOG is closed under forward applications of projection systems. In particular, if G is any context free grammar, and **P** be any projection system, then the set of the projections of $L(G)$ is a recognizable set.*

Extensions

Up to this point this section has assumed that the input patterns in the rewrite rules of the projection systems were untyped linear N-patterns. A first extension to the notion of projection system is to allow typed linear N-patterns to restrict the applicability of some projection transformation. This can be done using the techniques presented in Section 7.1. We leave the details for the reader.

Adding non-linearity is complex because of the same reasons that forbade its use in BURS theory, but it is feasible to consider allowing X-patterns, both typed and untyped as input patterns. This extension requires redefining the notion of a cover. The key point is what to do with the portions of the input tree that are assigned to an X-variable. Two obvious alternatives are to copy them directly, and to recursively transform them using the projection system. In most cases the second alternative is the more useful one. For example, that would be the case if we wanted to extend the projection system of Figure 7.7 to include detecting **repeat** statements. The transformation could be described as:



A Third Example of a Projection System

Figure 7.9

Either approach to X-patterns can be incorporated in projection systems using the techniques described in Section 7.1. The details of this approach will be discussed in a future document, which will, hopefully also include reports on the use of the technique.

7.3. Previous and Related Work

The techniques used for typed N-patterns are described by Engelfriet in [Eng75] in the context of finite state tree transducers. The author is not aware of any work done in the context of X-patterns.

The major previous work related to that presented in Section 7.2 are the tree transformation grammars (TT-grammars) of Keller et al. [KMP84]. TT-grammars are used to describe the mapping between two local sets. The description contains the input and output context-free grammars, associations between subgrammars of them, and associations between individual symbols in productions of the subgrammars. The input subgrammars are used to tile the input tree in a way similar to projection systems. The associations are then used to construct output tree fragments which are then put together to construct the output tree. TT-grammars can be seen as a cross between our projection systems and an attribute grammar [Knu68]: they are attribute grammars in which the domains of the attributes are always fragments of derivation trees of the output grammar, and the semantic functions are either simple "pasting" of the tree components, or just copies of tree-values, but they are extended so that the unit of locality (at which the semantic actions are chosen) is an input subgrammar instead of a production.

The main emphasis in TT-grammars is in being able to determine the shape of the sets of input and output trees while increasing the context information used to choose the transformation of a portion of the input tree. [KMP84] contains three subclasses of TT-grammars of varying degree of complexity. The simplest class, called *dual grammar translation scheme* (DGTS), corresponds to a projection system where the input patterns represent single productions in the input grammar (which can be described using typed variables where the type indicates that the derivation tree is rooted by a given non-terminal) while the output patterns may represent larger pieces of derivation trees.

The second class of TT-grammars, called *single input production – explicitly qualified* (SIPEQ), is an extension of the first class that shows more the AG relation of TT-grammars. In this class the input subgrammars still correspond to single productions, but the output subgrammars can be several, possibly disconnected, output parse subtrees. In addition it is possible to select between several output subgrammars based on some semantic attribute. Transformations described with this type of TT-grammar can "move" output tree values up and down the input tree and may not be describable using projection systems, as expressing such a transformation using a rewrite-based system seems to require iterated rewrite rule invocation.

The final class of TT-grammars discussed in [KMP84] is the *multiple input production* TT-grammars, that is, unrestricted TT-grammars. Unlike the first two classes, this one has not been implemented and its definition in [KMP84] has some obscure points: especially with the association of symbols when both the input and the output subgrammars have recursive non-terminals. As in SIPEQ TT-grammars, this class is implemented using an attribute grammar. Moreover, understanding the meaning of an specification using the full power of these two classes of TT-grammars requires the same kind of reasoning used with attribute grammars.

TT-grammars are more powerful than projection systems in the sense mentioned above. In a different direction, projection systems allow descriptions of transformations between any two sets of trees, and, given the properties proved in this section, the sets of trees can be restricted to be any recognizable sets, not just derivation trees. This would be useful in transformations like those dealing with abstract syntax trees. A disadvantage of the more general TT-grammars is that they have to rely on attribute grammars for their implementation and to reason about their semantics, while projection systems have a simpler and more natural semantic model. An additional advantage of projection systems is that they can be inverted; [KMP84] explicitly mentions this question as an open one for TT-grammars.

[KMP84] mentions that DGTS and SIPEQ TT-grammars have been used to specify and implement transformations from Ada to DIANA abstract syntax trees, from DIANA abstract syntax trees to C trees, and from FORTRAN 77 format statement to an intermediate representation, and to implement the front end of a manipulation system. Many of these applications seem describable using projection systems extended with typed patterns and X-patterns in the ways indicated at the end of the previous section. Further research will investigate the correctness of this hypothesis.

CHAPTER 8

A Code Generator Generator Using BURS

Now! Now! cried the Queen,
Faster! Faster!

[Lewis Carroll [1832 - 1898]]

The proof of the pudding
is in the eating.

[Miguel de Cervantes Saavedra [1547 - 1616]]

Chapter 6 presents an algorithm for UCODE but leaves open a major question: will a finite number of δ-UI LR graphs suffice to generate code for real target machines, and, if so, will the required tables be competitive with other code generation techniques? This chapter describes an implementation of a code generator generator based on BURS and the results of experiments with several machine descriptions that allow us to answer the question affirmatively.

The implementation of the code generator generator follows the theory of Chapter 6 with two main limitations: only factored machine grammars [Hen84] are accepted, and the input language is assumed to be the set of all trees over the given operator set. The first limitation does not impair the applicability of the implementation to code generation, but the second limitation should be removed in a production-quality code generator as it affects the detection of blocks and produces tables that are larger than necessary.

The machine descriptions used for the experiments describe three different target machines: VAX-11 [DEC81], a popular CISC (complex instruction set computer) architecture with a quite orthogonal instruction set; Mc68000 [Mot82], a popular micro-processor with a moderate number of addressing modes and a (relatively) small number of irregularities; and, RISC-II [KSP83], an experimental RISC (reduced instruction set computer) machine. Some target machines have more than one description.

All the machine descriptions come from the CODEGEN research effort, led at the University of California, Berkeley, by Susan L. Graham, and its successor at the University of Washington, UW-CODEGEN, under the direction of Robert R. Henry. While CODEGEN contains only a code generator based on Graham-Glanville technology, UW-CODEGEN is designed to compare different code generation techniques and also contains two techniques that generate locally optimal code based on dynamic programming: one using top-down pattern matching and the other using bottom-up pattern matching. The experiment reported in this chapter consisted of adding to UW-CODEGEN a new table-driven code generator and table generator based on BURS theory and measuring their behavior for several machine descriptions

The experiment used two groups of descriptions, summarized in Figure 8.1. The first group contains descriptions developed at UCB for a Graham-Glanville-based code generator (see [AGH84] for a report of that experiment). The second group are simplifications of these descriptions done at UW.

Keyword	Description	Types used in the description
vax.bwl	VAX-11	Byte, Word, Long
vax.bwlfd	VAX-11	Byte, Word, Long, Float, Double
vax.bwlfdgh	VAX-11	Byte, Word, Long, Float, Double, Giant, Huge
mot.bwl	Mc68000	Byte, Word, Long, some Float, some Double
risc.bwl	RISC-II	Byte, Word, Long
vax.ng	VAX-11, no generics	Byte, Word, Long
vax.ng.ne	VAX-11, no generics, no exceptions	Byte, Word, Long
mot.ng	Mc68000, no generics	Byte, Word, Long

Machine Descriptions

Figure 8.1

The first group comprises three VAX-11 descriptions using different data types, and one description each of Mc68000 and RISC-II. The machine descriptions in this group include generic operator rewrite rules since they are supported by the Graham-Glanville technology. The second group comprises three descriptions, one for the Mc68000, two for the VAX-11. These descriptions do not have generic operator rewrite rules because some of the techniques used in UW-CODEGEN do not allow them. In addition some rewrite rules that are useless for the input tree language actually reaching the code generator are removed. Removing these rules manually allows some more meaningful comparison between techniques when, as in our case, only some of them can benefit from specifying the input set. *Vax.ng.ne* is a modification of *vax.ng* where some rewrite rules have been dropped. The dropped rules are the ones that produce poor code in the Graham-Glanville technique.

All the machine descriptions produce a finite number of δ-LR graphs. Henry explains the structure of this type of machine descriptions [Hen84]. The main symbols are *register*, *rval*, and *lval*, describing values computed into a register, values that can be used in the right-hand side of an assignment, and values referring to locations that can be used in the left-hand side of an assignment, respectively. In addition there are a number of other symbols to describe individual addressing modes and classes of them, different types of constants, and other features of the target machine.

The VAX-11 machine descriptions are reasonably complex, due mainly to the large number of addressing modes present in that machine. The descriptions satisfy the conditions of Proposition 6.5 (on the finite number of δ-LR graphs) with $S = \{rval, register\}$; *rval* is needed in addition to *register* because there are unbounded rewrite sequences of the form $lval \rightarrow rval \rightarrow lval \cdots$ if the memory-to-memory operations of the VAX-11 architecture are used. Since VAX-11 is a CISC architecture its machine description allows many valid rewrite sequences rewriting an input tree into the goal. The RISC-II machine descriptions are very simple. Since RISC-II is a load/store architecture, Proposition 6.5 is satisfied directly with $S=\{register\}$, and there are few alternative ways to implement a given input tree. The Mc68000 machine descriptions are not as large as the VAX-11 descriptions because they do not have as many addressing modes or instructions, but they are more complex due to the non-uniformity of the register set of the Mc68000. This non-uniformity is handled using the techniques of "syntax for semantics" [Hen84], with different instruction fragment classes to track two sets of attributes:

dedicated and temporary registers, and address and data registers. Despite the complexity, Proposition 6.5 applies directly with $S=\{rval\}$.

All the machine descriptions associate with each rewrite rule a 4-tuple of positive integers. With these four values there are 6 cost functions of principal interest: each of the 4 projections, the lexicographic order on the tuple, and a constant cost function (effectively ignoring all the costs). The cost tuples for the different machine descriptions are shown in Figure 8.2. The combination of a machine description m and a cost function c is frequently denoted in this chapter as $m.c$

Machine	Cost tuple
vax.bwl	(M I S O)
vax.bwlfd	(M I S O)
vax.bwlfdgh	(M I S O)
mot.bwl	(C S I M)
risc.bwl	(C B I M)
vax.ng	(M I S O)
vax.ng.ne	(M I S O)
mot.ng	(M I S O)

Key	Meaning
K	Constant cost for all fragments
I	Number of instructions of the fragment
M	Number of memory bytes referenced by the fragment
C	Number of CPU cycles of the fragment
B	Number of bus cycles of the fragment
O	Number of operands in the fragment
S	Number of operands with hardware side effects in the fragment
L	Lexicographic ordering on the cost-tuple

Cost Functions

Figure 8.2

Although the finiteness of the number of δ-LR graphs can be inferred from the rewrite system underlying the machine descriptions, the actual number of states needed depends strongly on the cost function used. The experimental results of Section 8.1.2 show that the differences in the number of states required for each of the different cost functions can be substantial. For instance, the number of states for *vax.bwl.M* is 1249, while that for *vax.bwl.I* is 422, and *vax.bwl.K* is only 91. This difference stems from the different behavior of the cost functions.

Some cost functions lead to δ-UI LR graphs with small δ-costs and, thus, small total number of graphs. These cost functions are described as "shallow". The opposite situation is a "deep" cost function. In the VAX-11 description, the number of instructions is a shallow function because the VAX-11 has very powerful instructions and addressing modes and most of the time an input tree can be covered with a very small number of instructions, while the number of memory references is a deep cost function since different addressing modes may vary widely in

the number of memory references used. The "shallowest" cost function is the constant function; the "deepest" is the lexicographic order on the 4-tuple. The implementation reported here cannot deal gracefully with lexicographic order; see Section 8.4 for further comments.

The experiment involved implementing a a table generator and a code generator. The table generator, BURS-TG, is a stand-alone program that generates tables in the form of initialized C structures; its implementation details and experimental results are discussed in Section 8.1. The implementation details of the code generator, BURS-CG, are discussed in Section 8.2, with most of the experimental results being described in Section 8.3 where BURS-CG is compared with three other code generators integrated in UW-CODEGEN. The figures in that section show that the tables generated by BURS-CG are competitive with those generated by the other techniques and that the time spent solving UCODE in BURS-CG is much smaller than that spent in the other locally optimal code generators in UW-CODEGEN, and, indeed, even significantly faster than the one based on Graham-Glanville technology.

Section 8.4 presents the conclusions and lists the areas for further work.

8.1. Implementation of BURS-TG

The general organization of BURS-TG is essentially the one described in Chapter 6: first the δ-LR graphs are generated, then δ-UI LR graphs are chosen to minimize the number of required states, finally the resulting transfer tables are packed with some reasonable efficiency and are printed out as C initialization structures, to be used in BURS-CG.

8.1.1. Generating δ-LR Graphs

The δ-LR graphs are generated using a modified version of David Chase's algorithm for generating match sets in linear pattern sets (Section 3.5).

The central part of Chase's original algorithm is a generation of match sets from the combination of previous match sets. This is done in the procedure *add*() of Figure 3.9. This procedure has to be modified to generate new δ-LR graphs from previous δ-LR graphs, or, more exactly, the set of input nodes of the new δ-LR graph is computed from the set of output nodes of the previous δ-LR graphs. To support this computation the representation of a δ-LR graph includes a description of its set of output nodes and *add*() is actually invoked with a set of input nodes, which will then generate a δ-LR graph (and then will be considered for new combinations and so on, as in the generation of match sets).

Sets of nodes are sets of pairs $<\rho,c>$ where ρ is a subpattern, and c is its cost. For δ-LR graphs, the only important sets are those that are ***normalized***, i.e. those where the minimum cost in the set is 0. The modified algorithm uses sets of nodes whenever the original algorithm used sets of subpatterns. The key data structures involved are:

- **δ-LR graph.** In addition to a straightforward representation of the graph, the implementation keeps two copies of the set of output nodes. The first copy is just an optimization for speed, and always corresponds to the information in the graph. The second value is initially a copy of the first, but it will never be changed, even when the δ-LR graph is later modified to reflect restrictions. This second value is used to retrieve the folding information.
- $P_{op,i}$. This data structure is identical to the one used by Chase (see Section 3.5); namely, it is a set of trees, stored as a bit vector.
- $R_{G,op,i}$. For pattern matching, $R_{\sigma,op,i}$ is an indexed set of match sets, but for solving C-REACHABILITY $R_{G,op,i}$ is an indexed set of normalized cost-pattern pairs: the normalized set of the nodes of the δ-LR graph G appearing in $P_{op,i}$.

The other data structures are mostly unchanged. The modified version of *add*() is shown in Figure 8.3.

```
procedure add(IS)
    let IS be a set of input nodes;
    if IS is not new then
        return index of IS in the set of all input sets;
⇒
    compute the δ-LR graph G associated with IS;
    compute the output set from G, and normalize it
    store IS into the set of all input sets,
     and associate G with it;
⇒
    comment now we update R_{G,op,i};
    for each op ∈ OP do
        let n be the arity of op
        for each i, 1≤i≤n do
            compute R_{G,op,i};
            if R_{G,op,i} is new then
                store it;
                mark it with last_iteration (a global variable);
    return the index of IS into the set of all input sets;
```

Adding a δ-LR Graph

Figure 8.3

The implementation of this routine is quite straightforward, the only issue is in the specific data structures used. In this implementation, sets of <*pattern*,*cost*> pairs are represented as bit vectors plus explicit index+cost arrays, which allows the algorithms to select the fastest representation for a given operation, but it is quite expensive in memory. A production-quality implementation should be very careful in the selection of the data structures.

The above version of *add*() maintains a one-to-one mapping between input sets and δ-LR graphs. The next subsection explains that this is not always the case; the changes are done by inserting code at the places indicated by the ⇒.

The basic algorithm described above is modified slightly to deal with generic operator rewrite rules. For simplicity, most generic operator rewrite rules are transformed into normal reduction rules "before" generating the δ-LR graphs. This is done by considering the contexts where the rules could be applied and generating new rewrite rules that correspond to "instantiations" of the generic rewrite rules to those contexts.

An exception to the "instantiating" process described above are those generic operator rewrite rules, $Op(X_1,\ldots,X_n)\rightarrow Op'(X_1,\ldots,X_n)$, such that Op appears only in this rewrite rule, and where the semantic action does not refer to any of the attributes associated with $X_1,\ldots,X_n$. In this case, the application of the rewrite rule could have been done in a previous phase. Recognizing these patterns may reduce the number of patterns to consider at table generation time by ignoring the operators Op and using only Op'. This technique is particularly useful for the machine descriptions that were designed for Graham-Glanville technology, where generic operator rewrite rules are frequently used just as an abbreviation mechanism. A typical example are the rewrite rules abstracting the comparison operators (for example, $Ne\_l\rightarrow Cmp$). The

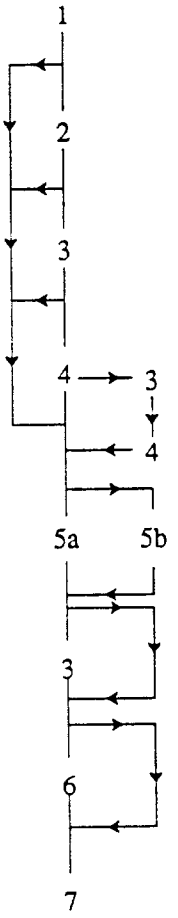
recognition of these rewrite rules by BURS-TG does not imply that BURS-CG needs to split its behavior into these two phases. A single phase is used in BURS-CG since it yields smaller tables and runs faster. The two-phase approach is slower because of the presence of the additional phase, and has larger tables because in the one-phase implementation the tables of all the operators that are abstracted into the same generic operator can be fully shared, while a two-phase implementation would (normally) require many entries to describe identity transformations in the first phase.

8.1.2. Selecting the δ-UI LR graphs

The next stage after generating the δ-LR graphs is to select uniquely invertible subgraphs that contain no useless nodes, contain enough information for solving UCODE, and are as small as possible. Proposition 5.17 showed that there is little hope for solving this problem optimally. Fortunately, the heuristic algorithm described in this section does an adequate job.

The general idea is to intermix the removal of useless nodes with the selection of increasingly more restricted δ-LR graphs until obtaining the δ-UI LR graphs. At each stage, the restriction of the δ-LR graphs is selected in a "first-fit" manner: two partially restricted LR graphs are compared and, if possible, are replaced by a common restriction. After a few iterations, 2 or 3 for our machine descriptions, this process converges to a set of, probably not uniquely invertible, restricted δ-LR graphs. At this stage δ-UI LR graphs are chosen. We provide two different mechanisms to do this. One algorithm tries to reduce the total number of δ-UI LR graphs, the other tries to reduce the "diameter" of the δ-UI LR graphs, that is, the length of the longest local rewrite sequence of the graphs. Then the LB-fsa that computes the δ-UI LR graphs is minimized using the algorithm described in Section 2.4.1. The overall organization[24] is shown in Figure 8.4; the rest of this subsection explains the most interesting details and then provides experimental results. The actual implementation follows quite closely this description except for some speed-ups, of which the most important one is the use of the folded representation of the LB-fsa in steps 3 and 6.

[24] The actual implementation allows some other heuristics. The figure only reports the most successful ones.



1 Original δ-LR graphs
2 On-the-fly Restriction of δ-LR Graphs
3 Find Useless Nodes and identical δ-LR graphs
4 Restriction of δ-LR Graphs
5a Select δ-UI LR Graphs (small)
5b Select δ-UI LR Graphs (fast)
6 LB-fsa minimization for UCODE
7 Final δ-UI LR graphs.

Computing Equivalent States

Figure 8.4

On-the-fly Restriction (Step 2)

The first (optional) restriction is on-the-fly, done inside the *add*() procedure of Figure 8.3. This is done by inserting code at the points indicated by ⇒ in Figure 8.3. As described in that figure, *add*(), maintains a one-to-one mapping between input sets and δ-LR graphs. When the on-the-fly restriction is used, the mapping becomes many-to-one, reflecting the fact that different input sets may have been found to generate different δ-LR graphs with a common restriction. If the mapping is called *inputs_to_graphs*, the code to be inserted at the two ⇒ would look like:

if IS is in the domain of *inputs_to_graphs* ***then***
| ***return*** the index of *inputs_to_graphs* (*IS*);

and

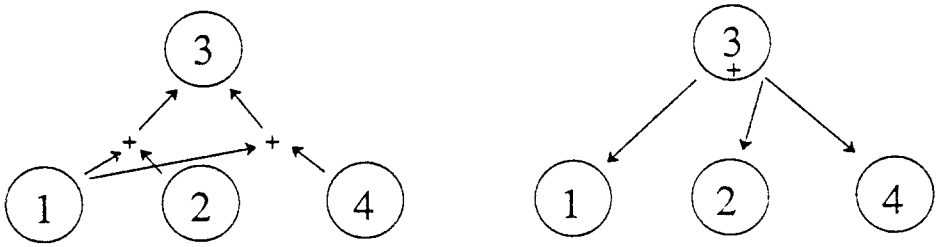
search for a previous state that can be merged with the current state
if there is one such G' *then*
    add (IS, G') to *inputs_to_graphs*
    return the index of G';

This restriction is very effective in reducing the number of δ-LR graphs that have to be considered, as it is done while the graphs are being generated.

Removal of Useless Nodes in the Graphs (Step 3)

The δ-LR graphs generated by the modification to Chase's algorithm assume that any output node is useful. Detecting and removing the useless nodes allows for further graphs to have common restrictions (and also leads to smaller representations for the graphs themselves). In addition, whenever an δ-LR graph is inverted, more nodes may be found useless. Thus, there is a need for the detection of useless nodes.

In the current implementation useless nodes are found by constructing a summary version of the LB-fsa computing the δ-LR graphs, and propagating useful nodes through it. The data representation used to represent the summary is quite simple and uses the fact that, in the stages before LB-fsa minimization, there is a single operator in the input language labeling the root of all the patterns in the set of input nodes of each LR graph. Since the constructed transition graph is going to be used only to propagate information backwards, all that is needed is to store, for each graph and for every position of a tree as a child of the operator associated with the graph, a list of all the graphs that, when characterizing a tree in that position, may transfer to the current graph. The transformation is shown in Figure 8.5. The list of states is implemented using a bit vector representation. This algorithm is simple but it is also moderately expensive in space and time.



Summary Transfer Graph

Figure 8.5

After removing useless nodes in the δ-LR graphs, identical states can be found, for example, by comparing the set of nodes of the graphs.

Restricting the δ-LR Graphs (Step 4)

Detection and removal of useless nodes shows new opportunities for the presence of common restrictions in the δ-LR graphs. Thus, each application of the previous algorithm is followed by an attempt to further restrict the graphs. This is done in a way identical to the application of

the "on-the-fly" restriction, using the updated information.

Repeated stages will further restrict the graphs until they are very "thin", almost uniquely invertible, and no further restrictions are possible.

Selecting the δ-UI LR Graphs (Steps 5a and 5b)

At some point the partially restricted δ-LR graphs are restricted completely to obtain the δ-UI LR graphs. This is normally done using a "global" heuristic that attempts to make the resulting δ-UI LR graphs as similar as possible. Since the previous stages were stopped at a point where no two graphs had a common restriction, this stage will not succeed in making any two graphs identical, but, if the graphs are similar, they may become identical after removing the nodes that became useless after this stage. If two or more edges reach a node in a graph, the heuristic tries to select an edge that corresponds to a rewrite rule used previously in another uniquely inverted graph. This makes graphs similar after the inversion, and reduces the number of different rewrite rules used (and therefore reduces table size). This heuristic is related to the heuristic used above for restricting the δ-LR graphs and, in some cases, its effects can subsume those of the previous stages. These "first-fit" heuristics yield tables with reasonable sizes. The real performance of the heuristic is difficult to estimate since the problems are large enough to preclude computing the optimal solution to the NP-problems exhaustively.

The previous paragraphs have assumed the goal is to minimize the number of non-equivalent states, that is, to generate small tables. BURS-TG can also try to generate a "faster" code generator at the expense of larger tables. This could be done optimally by defining a new cost function that would be a lexicographic ordering on a pair where the first component would be the (normal) cost function to minimize, and the second would represent the (code generation time) cost associated with generating the instruction fragment[25]. This approach has not been explored in BURS-TG for implementation reasons but seems likely to generate large tables. Instead BURS-TG computes an approximation: it avoids performing any of the restriction operations on the δ-LR graphs, and then uses a modified global heuristic for selecting the δ-UI LR graphs, trying to select always the shortest possible path in the δ-LR graphs. A later section reports on the results of these changes.

LB-fsa Minimization (Step 6)

This step is an straightforward implementation of the algorithm to minimize a LB-fsa using the considerations of Section 6.3.

Experimental Results

Figure 8.4 contains many possible sequences of actions to compute a final set of δ-UI LR graphs. A first collection of runs of the table generator provides some information on the value, and cost, of the most interesting sequences. Figure 8.6 contains two tables. The first table contains the number of states of *vax.bwl.I* at positions in 5 sequences; the second table does the same for *vax.bwl.M* except that the last sequence is not shown. The first table also shows the time required to create the tables. Each entry contains two numbers: real (elapsed time) and user time, as provided by the UNIX 4.3 system call *getrusage*. These times are in seconds in a Vax-11/8600.

---

[25] Note that it is an additive function.

vax.bwl.I									
Sequence	1	2	3	4	3	4	5a-3	6-7	Time(R/U) (seconds)
States	1298	798	525	473	441	441	441	422	470/424
States	1298	798	525	473	-	-	441	422	369/361
States	1298	-	962	655	512	463	441	422	629/597
States	1298	798	525	-	-	-	489	470	371/366
States	1298	798	-	-	-	-	492	470	296/292
States	1298	-	-	-	-	-	603	561	606/424

vax.bwl.M								
Sequence	1	2	3	4	3	4	5a-3	6-7
States	8619	4006	1726	1626	1596	1596	1596	1249
States	8619	4006	1726	1626	-	-	1596	1249
States	8619	4006	1726	-	-	-	1685	1309
States	8619	4006	-	-	-	-	1738	1318

Computing Equivalent States for *Vax.bwl.{M,I}*

Figure 8.6

Recall that step 2 is an on-the-fly application of step 4. The tables in Figure 8.6 suggest that it is normally useless to repeat step 4 more than twice. In fact, only one of the description-cost cases shows any change with three applications of step 4, and in that case the difference disappears after step $5a-3$. Since an additional application of these steps is expensive in time, we limit ourselves to two applications of step 4. These are chosen as steps 2 and 4, since the on-the-fly application is substantially faster, and cuts the memory requirements significantly.

The experiments also show that the deeper the cost function the larger the reduction obtained by the LB-fsa minimization. But, the benefits of the LB-fsa minimization are independent, to a certain extent, of the benefits of a careful selection of the δ-UI LR graphs. Trying to save time by doing a too hasty selection of the δ-UI LR graphs backfires since step 6 is a very significant part of the total time and it depends heavily on the number of states given. That is why the last row in the first table has such large times. Even worse, the same combination is missing from the second table because that sequence of steps would exceed the memory limits of the machine where it was run, in this case about ~28Mb.

It is necessary to explain where to apply step 3, detection and removal of useless nodes in the δ-LR graphs. Step 3 has to be performed between restrictions (steps 2 or 4) to allow the later restrictions to proceed. It is not applied between the last application of step 4 and steps $5a-3$ because the heuristic done in step $5a$ acts globally and pays no attention to the presence or absence of the nodes[26]. Finally, it is done between steps $5a$ and 6 because step $5a$ may introduce

[26] One could make a good argument for changing the global heuristic in *5a*, but this has not yet been done.

new useless nodes and step 6 will not detect them.

Because of the above considerations, the path 1–2–3–4–5*a*–3–6–7 is the one used by default in BURS-TG; we will call it the "preferred path". Figure 8.7 now shows the effect of this path on all the machine descriptions and cost functions.

Machine	1	2	3	4	5a-3	6-7
vax.bwl.K	645	304	171	109	95	91
vax.bwl.M	8619	4006	1726	1626	1596	1249
vax.bwl.I	1298	798	525	473	441	422
vax.bwl.S	1286	753	442	390	371	352
vax.bwl.O	651	304	168	109	95	91
vax.bwlfd.K	809	380	230	146	127	121
vax.bwlfd.M	12330	5777	2753	2635	2483	1835
vax.bwlfd.I	1677	1060	736	674	626	605
vax.bwlfd.S	1674	1013	643	585	552	531
vax.bwlfd.O	818	380	226	146	127	121
vax.bwlfdgh.K	1010	472	301	187	161	153
vax.bwlfdgh.M	-	-	-	-	-	-
vax.bwlfdgh.I	2123	1373	988	916	848	825
vax.bwlfdgh.S	2132	1333	894	830	779	756
vax.bwlfdgh.O	1022	472	296	187	161	153
risc.bwl.K	136	80	75	66	66	56
risc.bwl.C	227	151	143	142	142	132
risc.bwl.B	227	151	143	142	142	132
risc.bwl.I	222	151	145	144	144	134
risc.bwl.M	222	151	145	144	144	134
mot.bwl.K	362	200	192	187	158	147
mot.bwl.C	4509	1154	768	666	662	583
mot.bwl.B	362	200	192	187	158	147
mot.bwl.I	392	228	221	215	188	178
mot.bwl.M	1751	634	517	441	436	392
vax.ng.K	416	149	149	112	100	95
vax.ng.M	7482	2972	1228	1120	1120	1045
vax.ng.I	933	430	366	310	302	296
vax.ng.S	859	435	382	303	291	286
vax.ng.O	419	149	143	111	100	95
vax.ng.ne.K	379	182	182	115	100	95
vax.ng.ne.M	3049	1733	816	731	731	652
vax.ng.ne.I	754	417	328	276	276	270
vax.ng.ne.S	660	417	348	283	273	268
vax.ng.ne.O	381	182	143	111	100	95
mot.ng.K	293	190	185	182	182	167
mot.ng.M	3914	1089	787	643	637	576
mot.ng.I	293	190	185	182	182	167
mot.ng.S	309	213	211	209	209	194
mot.ng.O	1160	537	489	408	408	374

Preferred Path to Compute Equivalent States

Figure 8.7

The entry for *vax.bwlfdgh.M* could not be computed because, due to the space-hungry implementation of BURS-TG, the program runs out of swap space (28M). We expect the final table size to be in line with the results for *vax.bwl.M* and *vax.bwlfd.M*, but the final numbers will

have to wait for more swap space or better table-generation algorithms (more on this later). The space problem is even worse for the lexicographic cost functions. The only one that can be computed with the current algorithm and the available hardware is the one for RISC-II, where it is 134. This value is not very interesting because the RISC-II description is so simple and different from the other machine descriptions.

There is a large reduction in the number of states for *vax.ng.M* to *vax.ng.ne.M*, especially considering that the difference between the two machine descriptions is only the removal of 9 rewrite rules. These are the rules that make Graham-Glanville produce poor code for some input trees. Tracking the effect of these rules almost doubles the number of states required (but, as we will see, the table size increase is not as pronounced).

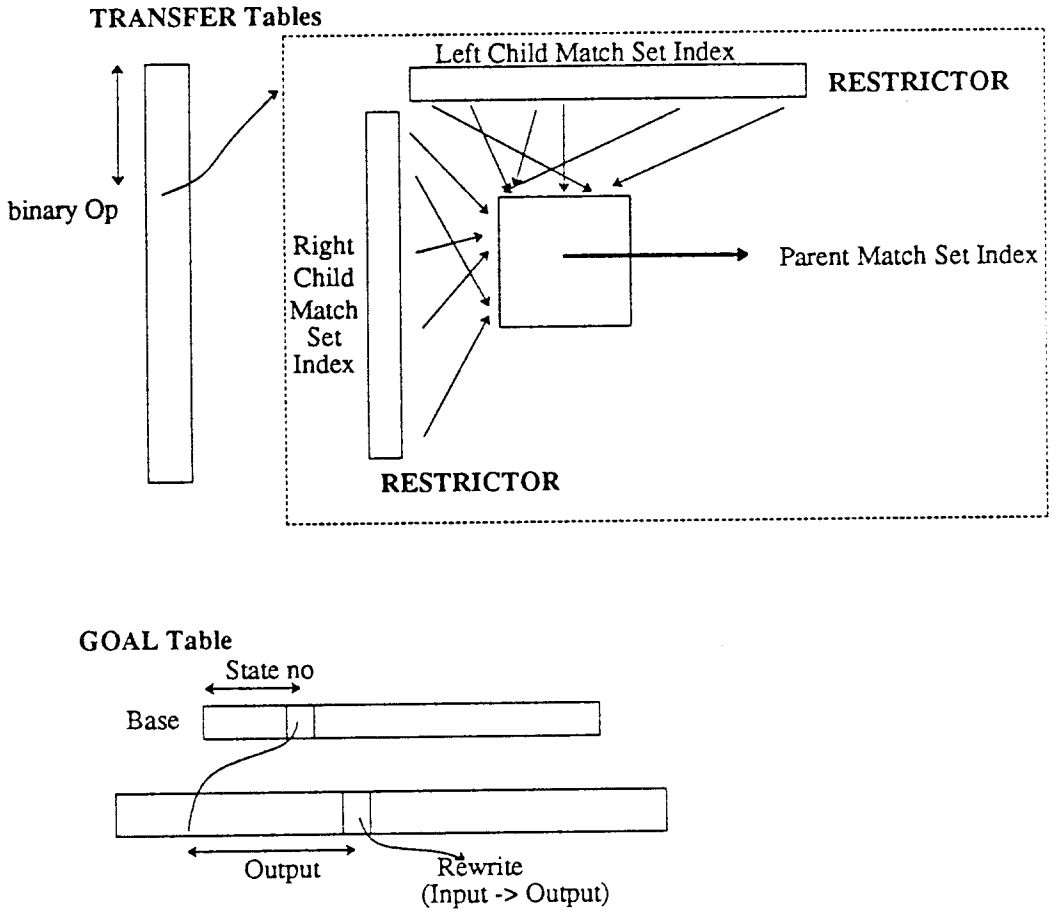
The entries in the table above show much the number of δ-UI LR graphs depends on the depth of the cost function, and also on the details of the underlying rewrite system, including the number of attributes that are encoded syntactically. It also reminds the reader of the potential effects of incorporating non-orthogonal input sets. The number of states in *vax.bwl* and *mot.bwl* are larger than those in *vax.ng* and *mot.ng*; although some of the difference is due to slightly different cost functions, most of the difference probably follows from the manual removal of rules that are known to be useless for the actual input language. A more controlled experiment will have to wait until BURS-TG is extended to allow for the description of the input set as a recognizable set.

The preferred path analyzed in the tables above tries to minimize the number of final states. Two other useful goals are to generate tables corresponding to code generators that obtain shorter final rewrite sequences, and to generate the tables fast. The first goal can be seen as an approximation to lexicographic order; it is called the "good code generation" (good-cg), and it is obtained by avoiding any of the restrictions and applying a heuristic that tries to obtain δ-UI LR graphs that have short diameter: the path 1-5b-3-6-7. The second approach is called the "fast table generator" (fast-tg) and it is obtained by generating the δ-LR graphs using step 2 to speed up the generation, and then applying the global heuristic: the path 1-2-5a-7. Section 8.1.4 summarizes results using these two paths and the preferred one.

8.1.3. Packing Tables

The output of the generation of the δ-UI LR graphs are tables organized as in Figure 8.8[27]. The top part of the figure, the TRANSFER tables, represent the LB-fsa computing the δ-UI LR graphs. The lower part represents the δ-UI LR graphs themselves.

[27] The two parts of this figure have appeared before in Chapters 3 and 5.



Data Structures of the Code Generator

Figure 8.8

8.1.3.1. Representing the δ-UI LR Graph LB-fsa

Most of the size reduction in the tables representing the LB-fsa is the result of the row and column folding obtained using the improvements to Chase's generative algorithms, including sharing identical restrictor arrays shown in Section 3.5, and the state equivalence techniques mentioned in Section 8.1.2. A final table compression mechanism is to represent the restrictor arrays using a variable number of bits (1, 2, 4, 8, or 16). This does not substantially slow the code generator but reduces considerably the table requirements. Figure 8.9 shows the influence of the encoding used for the restrictor arrays for one machine description; the other machine descriptions should be similar. Section 8.1.4 contains additional information on table measurements.

Mode	Restrictor Size	Total LB-fsa Size
No sharing/Full Size	123084	146404
Sharing/Full Size	55186	78506
Sharing/Bit-encoded	26010	48230[28]
No sharing/Bit-encoded	37196	59416

Influence of the Representation of the Restrictors for vax.bwl.M

Figure 8.9

The part of the LB-fsa that corresponds to *Plus_1* is always a large percentage of the total, because *Plus_1* appears in the patterns corresponding to the addressing modes. This is especially true for the VAX-11 descriptions. For instance, in *vax.bwl.M*, Plus_1 has 14.6KB out of 65KB of total table size, and in *vax.bwlfd.M*, 18KB out of 106KB. Several unsuccessful approaches were attempted to try to find a more compact, but still fast, representation.

The most interesting attempt notices that several rows (respectively, columns) differ in just a few columns (resp. rows). Hence it is possible to represent a column (row) by a "default" representative and a collection of "differences" relative to that representative. The columns (rows) and the number of entries in which they disagree can be modeled by vertices in a graph and weighted edges between them. If one allows multiple indirections (that is, the "default" of some column can be, in itself, represented by a default and a list, and so on) the smallest representation corresponds to finding a minimum-cost spanning forest of the graph. Each connected component is represented by a single column listed in "full" and as many difference lists as edges in the component. Access to the values of a given column requires consulting the lists associated with the edges until it is found, or the fully described column is reached.

This approach was implemented but the results were disappointing. The new tables had a size similar to the initial table, and much slower access time. The problem is that the columns are not similar enough, the difference lists are too big, and too many representatives are needed.

8.1.3.2. Packing the δ-UI LR Graphs

The representation used for the δ-UI LR graphs is the one discussed in Section 5.6.1. A δ-UI LR graph can be seen as a sparse row, with one entry per possible pattern appearing in the extended pattern set of the rewrite system. An entry will either be a ***don't-care*** entry if the represented pattern does not appear at all in the graph, or contain an indication of the rewrite rule representing the edge leading to the node in the graph. A collection of states can be stored together by superimposing the rows into a single 1-dimensional array, GOAL, with rows starting at different positions in the array indicated by another array BASE.

Finding the packing with the smallest GOAL is complex. The problem can be split into two different stages motivated by the observation that given an entry value, there is a single position in a row where it can appear: the one corresponding to the output pattern of the transformation

[28] The table that encodes whether a state includes the goal symbol or not can also be bit-encoded

whose number is the entry value. In the first stage, we determine what rows will share the same "base", that is, what rows are overlapped one directly on top of the other. This problem can be described as a clique detection problem, which is NP-complete [GaJ80]. (Construct a graph where each node corresponds to a row, and there is an edge between two nodes if they can be overlaid at the same BASE position. An overlapping arrangement is done by selecting a collection of disjoint cliques covering the graph. The best arrangement is one that minimizes the number of cliques). The second stage tries to find how to place the resulting (denser) rows into GOAL so that the number of "empty" slots is minimized.

The implementation follows these two stages, applying a first-fit approach to each problem. In the first stage, rows are overlapped as much as possible. In the second stage, each possible open position in GOAL is tried. The resulting layouts seem quite reasonable.

The representation of the δ-UI LR graphs described above does not specify how to detect that an input node has been reached. One possibility is to associate with those nodes an edge looping from the node to itself. A better approach is based on the observation that a node is an "input node" depending only on its associated pattern. Thus one can mark patterns as being an "input" or not. Furthermore, this information can be encoded into the pattern by a simple renumbering of the pattern's index. This has the additional advantage that it produces slightly better packing for the GOAL tables.

GOAL and BASE contribute to about half of the space requirements of the representation of the graphs, the rest being used to represent the rewrite rules themselves. For a large table like *vax.bwlfd.M*, GOAL is 7.6KB, and BASE is 3.6KB. The typical density (that is, the number of non-zero entries) of GOAL is about 60%, which places an upper limit on how much smaller it can become. Of course this limit can be unobtainable. One problem in obtaining higher densities are rows (corresponding to a δ-UI LR graph) that are very dense in some region, and impede any other row from being laid out crossing that region. If two of the dense regions are separated by a sparse region, that area may have trouble being filled in. A heuristic that tries to correct this situation removes the dense areas by "expanding" the rows by a factor k. This corresponds to changing the external numbering of the output trees. Unfortunately, this heuristic has not been effective for our machine descriptions, and it is clearly not effective for large values of k because the increase in the spread of the rows dominates whatever gains could be done with a better layout. Overall it seems to do little and it is not used unless explicitly requested by the user of the table generator.

The above paragraphs represent the graph one edge at a time. Another possibility is to provide, for each pattern in the output set of the state, the full local rewrite sequence (or, actually, the sequence of associated semantic actions). This approach increases the table space requirements (a "back-of-the-envelope" analysis of the *vax.bwl.I* machine description indicated a 4 to 5-fold increase in the space requirements for representing the graphs), but might produce slightly faster code generators (or it may not due to factors such as cache thrashing). Overall, this approach is not promising, and it is not used in our system.

8.1.4. Summary of Table Sizes

This section collects the results of several experiments showing the behavior of BURS-TG for different machine descriptions, cost functions, and the three heuristics mentioned previously (preferred, good code generation, fast table generation).

The table of Figure 8.7 indicates, for each machine description and each cost function, the number of δ-UI LR graphs generated initially, the number generated with the restriction on-the-fly, and the final number of non-UCODE-equivalent δ-UI LR graphs obtained using the "preferred path" of Figure 8.4. Figure 8.10 is just a summary of that of Figure 8.7.

Machine	K			M			I			S			O		
vax.bwl	645	304	**91**	8619	4006	**1249**	1298	798	**422**	1286	753	**352**	651	304	**91**
vax.bwlfd	809	380	**121**	12330	5777	**1835**	1677	1060	**605**	1674	1013	**531**	818	380	**121**
vax.bwlfdgh	1010	472	**153**	-	-	-	2123	1373	**825**	2132	1333	**756**	1022	472	**153**
vax.ng	416	149	**95**	7482	2972	**1045**	933	430	**296**	859	435	**286**	419	149	**95**
vax.ng.ne	379	182	**95**	3049	1733	**652**	754	417	**270**	660	417	**268**	381	182	**95**
mot.ng	293	190	**167**	3914	1089	**576**	293	190	**167**	309	213	**194**	1160	537	**374**
Machine	K			C			B			I			M		
mot	362	200	**147**	4509	1154	**583**	362	200	**147**	392	228	**178**	1751	634	**392**
risc	136	80	**56**	227	151	**132**	227	151	**132**	222	151	**134**	222	151	**134**

States for Preferred Path (Original Generated **Final**)

Figure 8.10

Figure 8.11 presents information on the table sizes, distinguishing between the contribution from the LB-fsa computing the δ-UI LR graphs and the contribution from the encoding of the graphs themselves. For each machine description and each cost function, it provides the size of the LB-fsa, the size for representing the graphs, and the total size.

Machine	K			M			I			S			O		
vax.bwl	1336	7560	**8896**	48230	16782	**65012**	11010	14546	**25556**	8572	13180	**21752**	1336	7560	**8896**
vax.bwlfd	1982	10470	**12452**	80146	25918	**106064**	18158	21934	**40092**	15292	20880	**36172**	1982	10470	**12452**
vax.bwlfdgh	2634	13682	**16316**	-	-	-	27198	31022	**58220**	24332	29932	**54264**	2634	13682	**16316**
vax.ng	1722	7582	**9304**	36690	17338	**54028**	6362	12778	**19140**	6812	12896	**19708**	1722	7638	**9360**
vax.ng.ne	1722	7698	**9420**	26388	15980	**42360**	5760	12320	**18080**	6390	12294	**18684**	1722	7682	**9404**
mot.ng	3652	12588	**16240**	22500	20788	**43288**	3652	12588	**16240**	4542	13978	**18520**	10320	18692	**29012**
Machine	K			C			B			I			M		
mot	2630	12290	**14920**	24944	20316	**45260**	2630	12290	**14920**	3634	13858	**17492**	12298	17942	**30240**
risc	1196	4376	**5572**	2396	7228	**9624**	2396	7228	**9624**	2412	7412	**9824**	2412	7412	**9824**

Table Sizes in bytes for Preferred Path (LB-fsa Graphs **Total**)

Figure 8.11

This figure can be combined with the previous one to obtain the table size (in bytes) per state. The table shows a very strong regularity; on closer examination, a similar regularity appears in the previous two tables. Using this data, one can make an "educated guess" to the values of *vax.bwlfdgh.M* : a bit over 60 bytes per state, about 2500 states, and about 150K. Testing this hypothesis will wait for larger computing resources or for better algorithms.

Machine	K	M	I	S	O
bwl	97.75	52.05	60.55	61.79	97.75
bwlfd	102.90	57.80	66.26	68.12	102.90
bwlfdgh	106.64	-	70.56	71.77	106.64
vax.ng	97.93	51.70	64.66	68.90	98.52
vax.ng.ne	99.15	64.98	66.96	69.71	98.98
mot.ng	97.24	75.15	97.24	95.46	77.57
Machine	K	C	B	I	M
mot	101.49	77.63	101.49	98.26	77.14
risc	99.50	72.90	72.90	73.31	73.31

Total Table Size (in bytes) per State

Figure 8.12

Machine	K			M			I			S			O		
vax.bwl	**184.48**	182.42	1.56	**8514.76**	7841.16	235.32	**691.38**	687.34	2.52	**613.28**	610.38	2.14	**185.12**	182.76	1.74
vax.bwlfd	**362.79**	360.14	1.82	**36791.3**	18953.2	4843.42	**1456.94**	1449.98	4.64	**1336.63**	1331.16	3.74	**364.36**	361.64	1.96
vax.bwlfdgh	**677.65**	673.56	2.74	-	-	-	**2900.34**	2864.36	15.96	**2737.53**	2696.84	16.7	**679.81**	675.78	3.2
vax.ng	**126.57**	124.64	1.34	**5699.23**	5552.36	67.78	**399.2**	396.62	1.8	**379.72**	377.4	1.58	**130.36**	128.36	1.32
vax.ng.ne	**132.6**	129.2	1.7	**2361.3**	2264.6	14.7	**398.8**	337.7	2.3	**368.4**	339.5	3.7	**148.1**	134.4	1.4
mot.ng	**282.8**	279.2	1.74	**2582.22**	2573.64	5.	**281.54**	279.34	1.4	**324.74**	321.58	1.9	**841.16**	836.32	2.92
Machine		K			C			B			I			M	
mot	237.64	**234.78**	2.8	2324.84	**2317.6**	4.78	237.68	**235.4**	1.64	278.98	**275.76**	2.4	929.93	**924.6**	3.32
risc	39.18	**33.12**	0.88	72.1	**67.12**	1.3	67.9	**65.52**	1.4	66.9	**65.48**	1.	67.12	**65.36**	1.

Table Generation Times for the Preferred Path (**Real** User System) (in seconds)

Figure 8.13

Finally, Figure 8.13 lists the time spent generating the tables. The values are on a Sun 3/175 with 12M of memory and no local disk. It must be re-emphasized that the implementation of BURS-TG can be called "exploratory programming", and that these numbers should be considered only rough upper bounds.

The numbers in Figure 8.13 correspond to the preferred execution path in BURS-TG. The next tables compare this path to the "good code generator" and the "fast table generator" paths discussed at the end of Section 8.1.2. Only the values for *vax.ng*, *vax.ng.ne*, and *mot.ng* are shown here.

States (Original Generated Final)					
vax.ng	**K**	**M**	**I**	**S**	**O**
Preferred	416 149 **95**	7482 2972 **1045**	933 430 **296**	859 435 **286**	419 149 **95**
Fast-tg	416 149 **149**	7482 2972 **2972**	933 430 **430**	859 435 **435**	419 149 **149**
Good-cg	416 416 **139**	- - -	933 933 **369**	859 859 **390**	419 419 **130**
vax.ng.ne	**K**	**M**	**I**	**S**	**O**
Preferred	379 182 **95**	3049 1733 **652**	754 417 **270**	660 417 **268**	381 182 **95**
Fast-tg	379 82 **182**	3049 1733 **1733**	754 417 **417**	660 417 **417**	381 182 **182**
Good-cg	379 379 **139**	3049 3049 **739**	754 754 **326**	660 660 **368**	381 381 **130**
mot.ng	**K**	**M**	**I**	**S**	**O**
Preferred	293 190 **167**	3914 1089 **576**	293 190 **167**	309 213 **194**	1160 537 **374**
Fast-tg	293 190 **190**	3914 1089 **1089**	293 190 **190**	309 213 **213**	1160 537 **537**
Good-cg	293 293 **185**	3914 3914 **705**	293 293 **185**	309 309 **207**	1160 1160 **481**

Comparing States for Alternate Paths

Figure 8.14

Figures 8.14, 8.15, and 8.16 provide some preliminary information on the tradeoffs available at table generation time. The "preferred" path yields the smallest number of states and the smallest tables. The table generation time can be (in some cases) reduced by more than a factor of 2 from that of the preferred path by using the "fast-tg" path. This reduction may be coupled to a table size increase that may be up to a factor of 2 relative to the "preferred" path. in some cases. The "good-cg" path is a first attempt to get some information on the table requirements for minimizing lexicographic cost ordering. The figures indicate that, for this approximation to lexicographic cost, the table size is increased over the "preferred" values, but without reaching the values for the "fast-tg" path. The table generation time for "good-cg" is increased significantly from that for "preferred", and so is the memory usage (not shown), which leads to a rapid deterioration of the behavior of the table generator in the case of *mot.ng.M* and reaches an extreme with *vax.ng.M*, for which the table generator cannot complete.

Table Sizes (in bytes) (fsa states total)															
vax.ng	**K**			**M**			**I**			**S**			**O**		
Preferred	1722	7582	**9304**	36690	17338	**54028**	6362	12778	**19140**	6812	12896	**19708**	1722	7638	**9360**
Fast-tg	2582	8570	**11152**	79588	22872	**102460**	9146	13926	**23072**	9250	14438	**23688**	2582	8570	**11152**
Good-cg	2514	10810	**13324**	-	-	-	7956	13892	**21848**	8862	14590	**23452**	2256	10548	**12804**
vax.ng.ne	**K**			**M**			**I**			**S**			**O**		
Preferred	1722	7698	**9420**	26388	15980	**42368**	5760	12320	**18080**	6390	12294	**18684**	1722	7682	**9404**
Fast-tg	2924	8729	**11652**	46924	18576	**65500**	8452	13324	**21776**	8772	13564	**22336**	2924	8728	**11652**
Good-cg	2514	10786	**13300**	29114	15858	**44972**	7090	19686	**20776**	8450	14282	**22732**	2256	10452	**12708**
mot.ng	**K**			**M**			**I**			**S**			**O**		
Preferred	3652	12588	**16240**	22500	20788	**43288**	3652	12588	**16240**	4542	13978	**18520**	10320	18692	**29012**
Fast-tg	4102	13446	**17548**	35354	23538	**58892**	4102	13446	**17548**	4818	14402	**19220**	13454	20674	**34128**
Good-cg	4210	14598	**18808**	26964	22360	**49324**	4210	14598	**18808**	4906	15194	**20100**	13734	20842	**34567**

Comparing Table Sizes for Alternate Paths

Figure 8.15

Table Generation Times (in seconds) (Real User System)															
Vax.ng	K			M			I			S			O		
Preferred	**126.5**	124.6	1.3	**5699.2**	5552.3	67.7	**399.2**	396.6	1.8	**379.7**	377.4	1.5	**130.3**	128.3	1.3
Fast-tg	**87.1**	85.1	1.4	**2313.5**	2263.4	23.8	**223.7**	221.2	1.6	**224.8**	222.3	1.4	**87.5**	85.9	0.9
Good-cg	**193.6**	190.7	1.5	-	-	-	**568.3**	563.6	2.6	**505.9**	499.7	2.8	**177.3**	175.1	1.5
Vax.ng.ne	K			M			I			S			O		
Preferred	**132.6**	129.2	1.7	**2361.3**	2264.6	14.7	**398.8**	337.7	2.3	**368.4**	339.5	3.7	**148.1**	134.4	1.4
Fast-tg	**94.1**	83.6	1.4	**921.3**	867.4	6.8	**207.7**	181.1	2.2	**196.5**	180.7	2.5	**111.7**	84.6	1.6
Good-cg	**204.7**	174.2	2.4	**4016.3**	3671.4	89.9	**451.2**	423.8	3.5	**391.5**	371.1	2.3	**178.2**	158.8	2.1
mot.ng	K			M			I			S			O		
Preferred	**282.8**	279.2	1.74	**2582.2**	2573.6	5.	**281.5**	279.3	1.4	**324.7**	321.5	1.9	**841.1**	836.3	2.9
Fast-tg	**172.4**	168.2	2.2	**1757.9**	1748.3	5.6	**170.7**	168.1	1.6	**194.6**	192.2	1.6	**518.9**	515.2	2.2
Good-cg	**216.7**	214.2	2.4	**19984.3**	7671.3	2564.2	**217.6**	213.9	2.3	**236.4**	231.8	2.1	**1158.3**	1132.4	6.38

Comparing Table Generation Times for Alternate Paths

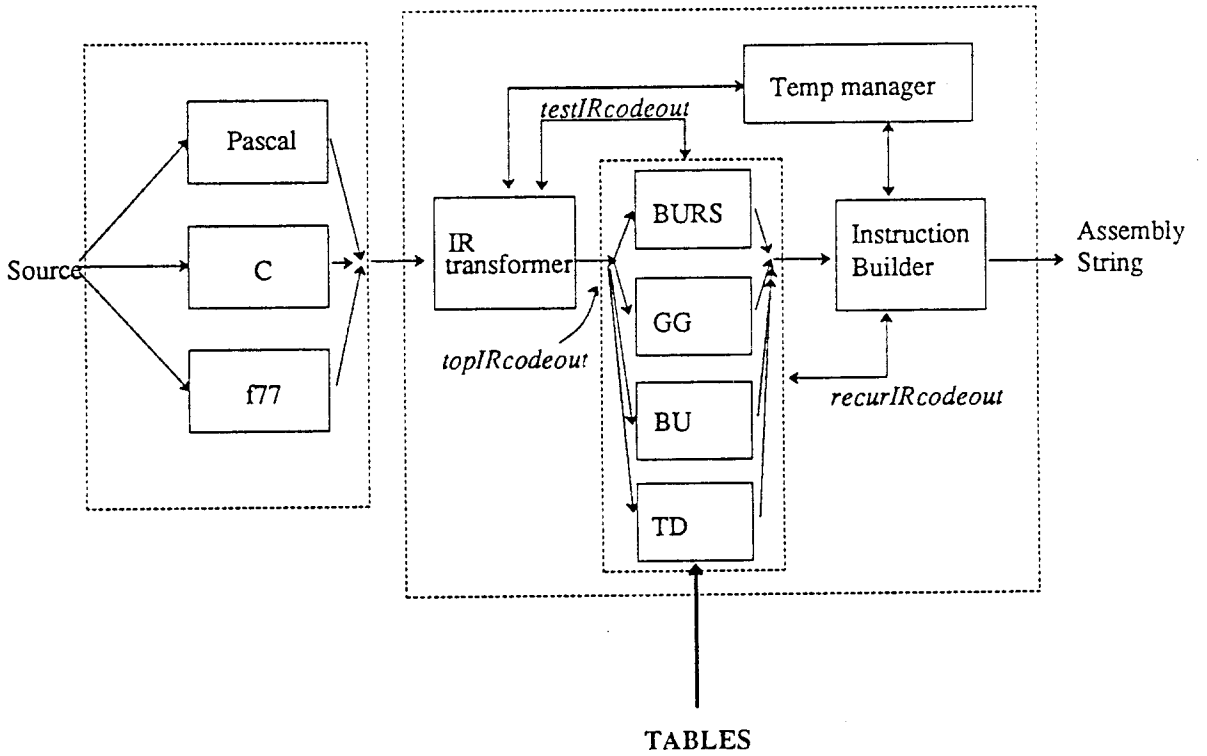
Figure 8.16

Vax.ng.M does not show any values for "good-cg" because BURS-TG runs out of swap space on it.

8.2. Implementation of BURS-CG

The tables of Figure 8.8 are used at solving time by BURS-CG following the ideas presented in Section 5.6.1, and in the context of UW-CODEGEN.

UW-CODEGEN implements a plug-in compatible replacement for the code generation phase of the portable C compiler PCC [Johes]. As such it accepts as input PCC-IR, a collection of expression trees, and produces as output assembly code. Register allocation is done on-the-fly, one expression tree at-a-time. The internal organization of UW-CODEGEN is shown in Figure 8.17; see [Hen84] for more details.



Organization of UW-CODEGEN

Figure 8.17

The kernel of the code generator is what Robert Henry calls a ***tree pattern match select replacer*** (abbreviated TPMSR) [HeD87], which is, in fact, a module that will solve UCODE, or an approximation to it. The TPMSR are all generated automatically from the machine description, are mostly table-driven, and all have three main interfaces with the rest of the code generator: ***topIRcodeout***, ***recurIRcodeout***, and ***testIRcodeout***. In each case the TPMSR is given an input tree, and the output is the cost, maybe infinite, needed to rewrite it into the goal, plus, in some cases, a side-effect. UW-CODEGEN contains implementations for several TPMSRs. Section 8.3 below compares their performance with the BURS-based TPMSR described in this chapter.

The "principal" interface to all the TPMSR is ***topIRcodeout***; as a side-effect it will invoke the sequence of semantic actions associated with a rewrite sequence for the expression tree that is its input. This expression tree is the output of an ***IR transformer***, currently a simple type of tree transformer. The IR transformer generates a sequence of trees for each of the original PCC-IR trees. The transformation process sometimes requires measuring the complexity of the subtrees. This is done by querying the machine description via the ***testIRcodeout*** interface to the TPMSR. In this interface only the cost is computed and no semantic functions are evaluated.

The instruction-building actions and the IR transformer also interact through the temporary and register manager, which does the on-the-fly register allocation and assigns temporaries. The result of the instruction-building actions is a sequence of assembly instructions. In some cases

the cleanest way to generate code for some construct is to build an instruction tree and then generate code for it. This is done with a recursive invocation of the TPMSR, now using the ***recurIR-codeout*** interface.

The internal implementation of each interface of BURS-CG is very simple. There are two main recursive routines. The first routine is invoked once per node to compute the δ-UI LR graphs; it implements the first, bottom-up pass of UCODE. The second routine is conceptually invoked once per rewrite rule application; when going down the tree it finds the local rewrite sequences, when unraveling it can be used to invoke the semantic functions. The actual implementation preallocates enough space per node for all the possible rewrite rules and performs only one recursive call per node. **topIRcodeout** is identical to **recurIRcodeout**. **TestIRcodeout** varies in that the semantic functions are not invoked.

8.3. Comparison with Related Work

Chapter 6 mentions several other approaches to UCODE, and compares them against the BURS-based techniques from a theoretical perspective. This section measures the performance of three of these techniques, denoted by GG, TD, and BU, and compares them against our own, denoted by BURS. all in the framework of UW-CODEGEN.

GG is the "classical" Graham-Glanville technique, in its "pure syntax" mode. It only solves an approximation to C-REACHABILITY. [AGH84] is still the best reference for this method. TD and BU both solve C-REACHABILITY exactly, using representations of the full-cost LR graphs. TD uses a top-down pattern matcher and represents the full-cost LR graphs explicitly. TD is related to the proposal of Aho, Ganapathi, and Tjiang [AGT86], except that in that proposal, the tree transformation phase is mixed with the solving of reachability, while in UW-CODEGEN the two phases are done sequentially. [AGT86] does not provide many details on the interaction between the tree transformer and the reachability phase, but it seems to be quite similar to the one in UW-CODEGEN. It is cast in a better formalism but it has the disadvantage of describing many actions purely semantically, thus not providing much information at table construction time. BU uses a bottom-up pattern matcher and uses a mixed representation of the full-cost LR graphs. The LR graph is represented implicitly with an index value, while the cost information is represented explicitly. Reinhard Wilhelm and Beatrix Weisgerber in [MWW84] propose two methods that are similar to TD and BU. [HeD87] contains a detailed description of the implementation of UW-CODEGEN and measurements of GG, TD, and BU.

Since all the TPMSRs in UW-CODEGEN are plug-in compatible, it is possible to make meaningful comparisons in table size and code generation speed. This is the intention of this section. The UCB machine descriptions cannot be integrated fully into UW-CODEGEN and the only descriptions used for comparing the different approaches are the UW descriptions, *vax.ng.ne* and *mot.ng*.

The comparison uses the BURS tables obtained with the 3 alternate paths of the previous section; in this section they are named BURS-p for the preferred path, BURS-tg for the fast table generation path, and BURS-cg for the "good" code generation path. The three tables lead to code generators that run at very similar speeds; based on the length of the rewrite sequences that they find, BURS-cg should be faster than BURS-p which, in turn, should be faster than BURS-tg but the difference is small enough it cannot be detected reliably using the profile tools available (and given the variation due to the presence of architectural features like caches).

For measuring the dynamic performance of the algorithms to solve C-REACHABILITY, the same benchmarks are used as in [AGH84], [Hen84] and [HeD87]. There are 6 benchmark programs, all written in C, shown in the table of Figure 8.18.

Name	Size in Lines	Description
grep	466	find instances of regular expressions in a file
reader	1013	tree manipulation routines from the front end of PCC
knuth	118	with many complex expressions and array indices
puzzle	125	Forest Baskett's bin packing program
dh	1266	terminal driver
mark	509	one module of a VLSI circuit simulator (Crystal)

Benchmark Programs

Figure 8.18

Table Size

Figure 8.19 gives information on table size. The size information is split into two categories: the representation for the automaton used (which particular automaton is used varies depending on the method) and the representation of the states themselves.

Target		GG	TD	BU	BURS-p	BURS-cg	BURS-tg
vax.ng.ne	fsa	33.7	20.1	56.4	26.3	29.1	46.9
	states	8.8	18.2	18.6	15.9	15.8	18.5
	total	42.5	38.3	75.0	42.3	44.9	65.5
mot.ng	fsa	33.0	18.6	50.2	22.5	26.9	35.3
	states	8.7	18.2	18.2	20.7	22.3	23.5
	total	41.7	36.8	68.4	43.2	49.3	58.8

Table Size for several TPMSR (in KBytes)

Figure 8.19

The values for GG, TD, and BU are from [HeD87]; all the BURS values correspond to the 1st cost component (M) and were measured directly. *Vax.ng.ne* is used because it is the value used in GG, TD, and BU. A comparison of the values of TD and BU (both implemented by Henry and Damron) show a substantial difference in the table sizes of the top-down and bottom-up automata. Chase reports in [Cha87] that the *vax.ng* generated tables ranging from 78.7KB for no fancy encoding at all, to 22.9KB for a bit-encoding similar to the one used in our own implementation, to 12.4KB for a run-length encoding. The middle alternative, which is the one used in BURS-CG, causes little loss of solving time speed and would produce tables very comparable to the top-down pattern matching ones. In addition, recall that Chase's original implementation did not share identical "restrictor" arrays. Most likely, with that additional optimization, the

bottom-up tables would be smaller than the top-down tables.

The encoding of the 652 states in BURS-p takes 15.9KB while the 380 states in BU use 18.6KB. The reason is that the BURS states, being δ-UI LR graphs, encode only one alternative per useful output node, while the BU states need to encode all the possibly useful alternatives.

The comparison between BURS-p, BURS-cg, and BURS-tg is as expected: the smallest table corresponds to BURS-p, and the largest to BURS-tg, where no effort was made to generate small tables. Most of the increase in table size comes from the LB-fsa, since the representation of the states depends substantially on the rewrite rules being used, which stay constant.

Speed Solving UCODE

The time spent solving UCODE was measured by using a profiler, ***gprof***, [GKM83] to measure the time spent in the routines solving the problem, and by compensating for the time spent measuring the routines themselves. Currently UW-CODEGEN runs on Vax-11. Two implementations of the architecture were available for the experiments: a Vax-11/8600 and a Vax-11/750. The 8600 was chosen because it has a more rational cache architecture. Despite this precaution and running the programs only on unloaded machines, individual measurements fluctuate quite a bit, so the actual values used were the average of 6 compilations of each benchmark program.

Figures 8.20 and 8.21 give information on the time required to solve the UCODE by the different TPMSRs. The time includes the three code interfaces to the TPMSRs. There was no significant difference between the different versions of BURS, and a single column (for BURS-p) is shown. The interface between UW-CODEGEN and BURS-CG is slightly less efficient than it should be, so the numbers for BURS could actually be slightly better.

vax.ng.ne	GG	TD	BU	BURS
Time in UCODE Normalized to GG				
grep	1.00	3.79	3.39	0.83
reader	1.00	3.29	2.88	0.61
knuth	1.00	4.60	3.72	0.56
puzzle	1.00	3.01	2.55	0.51
dh	1.00	3.37	3.04	0.63
mark	1.00	3.55	3.06	0.65
Percentage of Code Generation Time in UCODE				
grep	12.02	34.96	32.24	9.52
reader	15.10	35.63	33.40	8.84
knuth	13.67	22.10	21.51	8.73
puzzle	16.02	33.98	32.27	7.93
dh	13.35	34.06	33.16	8.26
mark	14.09	37.10	34.25	9.16
Total Code Generation Time (User System) (in seconds)				
grep	**7.67** 0.79	**9.69** 0.95	**9.35** 0.96	**7.10** 0.89
reader	**22.24** 2.46	**29.36** 2.79	**28.15** 2.51	**20.88** 2.52
knuth	**3.20** 0.18	**6.69** 0.67	**5.67** 0.64	**2.88** 0.18
puzzle	**4.97** 0.57	**6.39** 0.67	**6.04** 0.69	**4.63** 0.61
dh	**11.08** 1.17	**14.29** 1.83	**13.53** 1.23	**10.46** 1.05
mark	**7.81** 0.92	**10.33** 0.90	**9.90** 0.91	**7.25** 0.85

Code Generation Time for vax.ng.ne

Figure 8.20

mot.ng	GG	TD	BU	BURS
Time in UCODE Normalized to GG				
grep	1.00	3.50	2.92	0.53
reader	1.00	3.65	3.13	0.52
knuth	1.00	4.53	3.78	0.60
puzzle	1.00	3.67	3.23	0.53
dh	1.00	3.72	3.10	0.59
mark	1.00	3.85	3.18	0.53
Percentage of Code Generation Time in UCODE				
grep	15.97	41.55	37.05	8.31
reader	17.68	46.17	41.94	8.73
knuth	16.27	24.33	25.23	9.60
puzzle	15.93	41.55	38.12	8.01
dh	16.48	45.03	40.00	9.31
mark	17.36	46.04	41.45	8.93
Total Code Generation Time (User System) (in seconds)				
grep	**9.16** 0.90	**12.07** 0.99	**11.63** 0.85	**8.67** 0.86
reader	**25.90** 2.39	**35.24** 2.52	**33.19** 2.61	**24.43** 2.54
knuth	**9.11** 0.67	**20.51** 1.53	**17.57** 1.47	**8.49** 0.65
puzzle	**7.56** 0.73	**10.28** 0.86	**9.87** 0.84	**7.50** 0.75
dh	**14.81** 1.27	**19.76** 1.40	**18.83** 1.32	**14.04** 1.26
mark	**8.97** 0.82	**12.17** 0.94	**11.57** 0.93	**8.49** 0.81

Code Generation Time for mot.ng

Figure 8.21

BURS is substantially faster than any other TPMSR. It easily outperforms BU and TD because it avoids handling costs explicitly. It is more surprising that BURS is even faster than GG. A careful comparison of the respective portions of code implementing UCODE showed several causes for the difference in speeds. Probably the biggest contribution lies in the representation of the automaton: GG uses a tight encoding and a cache, which loses in speed against the more efficient table folding. In addition, GG uses the normal technique (for parsing technology) of default transitions, which is slower than a simple lookup. Another contributor is that the relationship between the parser used in GG and the traversal of the tree providing the prefix traversal is not as simple as the tree traversal used by BURS. Finally, GG stores states and other information in a stack (the parse stack), while BURS uses (pre-allocated) slots associated with the tree; the stack requires extra checks for overflow and the like. GG also uses a few more indirect routine calls than BURS. Despite the difficulty in comparing the methods in the presence of these differences in implementation strategy, we think that the evidence shows that BURS is, at least, comparable in speed to GG. To reduce effects caused by compilation of the algorithms, the values shown in Figures 8.20 and 8.21 correspond to GG compiled using the peephole optimizer, and BURS without it; the values are more favorable to BURS otherwise.

If BURS is used, the time spent solving UCODE becomes a quite small percentage of the total time in the code generator. After achieving this reduction, the next goal is to increase it again, by

reducing the time spent in the other parts of the code generator. The main target for "reduction" are the IR transformer routines, which are currently quite inefficient. Work is underway by the author and other researchers to speed up this phase.

Quality of the Generated Code

For the purposes of this section, the "cost" of the generated code is the cost of its associated rewrite sequence as a 4-tuple. GG does "maximum munching" which may lead to non-optimal sequences in all cost components. TD, BU, and BURS are all capable of generating optimal code, but, due to the limitations in the current implementation of BURS-TG, BURS uses tables that minimize only the first component of the cost tuple. BURS-cg provides the best approximation to the lexicographic cost among BURS-tg, BURS-p, and BURS-cg and BURS-tg provides a slightly better value than BURS-p. The costs are shown in Figure 8.22 normalized to 100 as the optimal cost; the smaller an entry, the smaller its cost. The abnormality of the fourth component of BURS-cg in *mot.ng* is due to a strange bug in BU and TD that misleads the rewrite sequence from *Icon_w* into *dreg_temp_1*.

vax.ng.ne	GG	BURS-p	BURS-cg	BURS-tg
grep	102.3 108.5 97.4 100.9	100.0 103.4 103.3 100.0	100.0 100.2 101.6 100.0	100.0 103.4 103.3 100.0
reader	100.6 101.8 99.5 100.8	100.0 102.0 102.7 100.0	100.0 100.0 101.0 100.0	100.0 102.0 102.7 100.0
knuth	117.4 115.5 104.7 100.0	100.0 106.3 104.4 100.0	100.0 100.0 100.3 100.0	100.0 100.0 104.4 100.0
puzzle	100.8 101.4 100.0 110.0	100.0 100.0 100.7 100.0	100.0 100.0 100.7 100.0	100.0 100.0 100.7 100.0
dh	101.9 104.1 98.8 109.5	100.0 103.1 103.2 100.0	100.0 100.8 101.0 100.0	100.0 103.1 103.3 100.0
mark	100.4 101.0 99.4 100.0	100.0 101.8 106.1 100.0	100.0 100.0 101.5 100.0	100.0 101.8 106.1 100.0
mot.ng	GG	BURS-p	BURS-cg	BURS-tg
grep	101.4 100.0 103.1 95.8	100.0 100.0 100.0 100.0	100.0 100.0 100.0 98.8	100.0 100.0 100.0 100.0
reader	101.3 100.0 104.0 97.6	100.0 100.0 100.0 103.7	100.0 100.0 100.0 99.8	100.0 100.0 100.0 103.7
knuth	102.5 100.0 94.7 100.6	100.0 100.0 100.0 108.6	100.0 100.0 100.0 100.0	100.0 100.0 100.0 108.6
puzzle	104.5 100.0 112.2 95.7	100.0 100.0 100.0 102.8	100.0 100.0 100.0 99.8	100.0 100.0 100.0 102.8
dh	103.1 100.0 103.2 100.3	100.0 100.0 100.0 104.1	100.0 100.0 100.0 99.3	100.0 100.0 100.0 104.1
mark	101.8 100.0 102.5 98.3	100.0 100.0 100.0 102.2	100.0 100.0 100.0 99.8	100.0 100.0 100.0 102.2

Quality of the Generated Code

(value of cost tuple (M I S O) relative to lexicographic optimum (100 100 100 100)

Figure 8.22

Table Generation Times

One of the main disadvantages of the current implementation of BURS-TG is that it generates tables very slowly. Direct comparisons of BURS-TG against the table generators of GG, TD, and BU are not available, but the tables of Figures 8.23 give some relevant information. The top of the figure shows times in seconds on a Sun-3/75 with 12 MB of main memory and no local disk. The bottom of the figure reproduces information from [HeD87] comparing the performance of the different table generators in UW-CODEGEN; values are in seconds on a DEC Microvax-II. There are

two columns for BU: the first column corresponds to the generation of tables without any effort to use cost information at table-generation time to reduce the number of alternatives to consider at code generation time; the second column corresponds to the tables used in our other comparisons, in which some elimination of alternatives is done based on costs. We emphasize again that the current implementation of BURS-TG was written with no special effort to generate tables fast.

vax.ng.ne	K	M	I	S	O
BURS	132.6	2361.3	398.8	368.4	148.1
BURS-cg	94.1	921.3	207.7	196.5	111.7
BURS-tg	204.7	4016.3	451.2	391.5	178.2
mot.ng	K	M	I	S	O
BURS	282.84	2582.	281.5	324.7	841.1
BURS-cg	172.4	1757.9	170.7	194.6	518.9
BURS-tg	216.7	19984.3	217.6	236.4	1158.3
					(Sun 3/75 seconds)

Machine	GG	TD	BU	BU-cost
vax.ng.ne	204.7	58.0	242.1	625.7
mot.ng	194.8	61.0	442.9	1753.1
				(μVax-II seconds)

Table Generation Times for several TPMSR

Figure 8.23

8.4. Conclusions and Further Work

BURS-based code generators seem to be the method of choice for generating code in the compilation model supported by UW-CODEGEN. They generate code as good as the best methods available, namely any of the dynamic programming-based techniques, with tables of competitive size and with code generation speed faster than the previously fastest techniques (PCC and GG in UW-CODEGEN). The inflexibility in changing cost functions "on-the-fly" does not seem a real disadvantage, but, given the table size, it could be supported, if desired, by changing tables.

Two other well-known systems that have not been compared directly to BURS are PCC2 and *twig*. We did not have access to the systems for direct measurements, but there are a few published numbers. [AGT86] indicates that a compiler built using *twig* had a speed improvement of 25% over one built using PCC2; PCC2 is normally considered to be twice as slow as PCC1, (see [HeD86] which cites a technical report by Aho, Ganapathi, and Tjiang), and we know that GG can run as fast as PCC1 [Hen84]. Hence, *twig* is substantially slower than GG and, hence, than BURS. The portion of *twig* that solves a subproblem similar to UCODE is based on the same theory as TD, and one would expect their running times to be similar.

There are several questions left open after this experiment. Most of the questions are related to the current implementation of BURS-TG. Briefly the approach used now is to generate a large number of states and then find and force equivalent states among them. This is both slow and, more important, it consumes a large amount of memory, so much so that it has been possible to generate tables using the lexicographic cost function only for *risc.bwl*. It is of theoretical (although probably not practical) interest to know what will happen with the descriptions for the Mc68000 and VAX-11. The current implementation of BURS-TG grew with BURS theory and it is very "exploratory"; a better selection of data structures will improve performance significantly, but a drastic reduction awaits new approaches to the generation of the states.

Another question related to the current implementation is what is the effect of the input tree language on the table generation time and the size of the generated tables. Comparing *vax.bwl* and *vax.ng* shows a non-trivial reduction in the number of states and table size. This is, to a large extent, due to the removal of useless rewrite rules. Chapter 6 already has the theory required to examine this issue. Implementing it would allow quantifying its benefit.

Finally, is is probable that better heuristics can be found to find the best δ-UI LR graph and to lay out the GOAL array. Such heuristics could further reduce the size of the generated tables.

Another direction of research includes more experiments with different target machines, and also with different descriptions of them, since the ones used for this experiment where developed considering the idiosyncrasies of the Graham-Glanville techniques.

8.5. Acknowledgements on this Chapter

This implementation of BURS-CG and BURS-TG has used previous work from David Chase and Robert Henry.

Chase gracefully made available to us the code implementing his algorithms to build match set B-fsa. I had avoiding "bitting the bullet" of implementing my algorithms for solving C-REACHABILITY and UCODE for several months, but the opportunity provided by the presence of his code prompted me to build my table generator in top of it. My original approach was to construct the first a table representation of the BURS-state B-fsa and then find row and column foldings of the tables. Given the large initial number of states this approach would have been very inefficient, if at all feasible. The timely appearance of Chase's algorithms (late 1986) saved me from, at best, having to re-discover them, or, at worst, giving up on the implementation altogether. Overall, I increased the original 2700 lines of finely tuned code from Chase to more than 10000 lines of "exploratory" code, in the process slowing down his code by an order of magnitude.

UW-CODEGEN is the result of the effort of many people at UC Berkeley and at UW, and, foremost among them is Robert Henry. Its availability was crucial in order to measure the performance of the code generators. In some sense, my work can be seen as an outgrowth of the CODEGEN effort.

CHAPTER 9

Conclusions

I do not think we can hope for any better things now.
We shall stick it to the end,
but we are getting weaker, of course,
and the end cannot be far.

It seems a pity, but I do not think I can write more.

Diary of the Terra Nova Expedition to the Antarctic
March 29, 1912 (last entry)

[Robert Falcon Scott [1868-1912]]

In this dissertation, we have studied some tree transformation problems using a descriptive mechanism that can be characterized as "natural" extensions of "pure" tree rewrite systems, with as little intrusion as possible from other techniques such as attribute grammars. The dissertation explores two different areas: it contains some foundational work to support the thesis that such a descriptive mechanism can be made expressive enough for many applications, and, it shows that even simple tree rewrite systems are very useful. The increase to the descriptive power of the mechanism is based on extending the notion of pattern. Although the dissertation does not provide the "last" word in this aspect, it does show several useful extensions and how to implement them efficiently. The usefulness of simple tree rewrite systems is based on variations of the notion of BURS.

9.1. Extending Patterns

"Traditional" tree patterns are what have been called linear N-patterns: patterns where variables have arity 0 and are not repeated. They are studied in Chapter 3, where a unified description of several pattern matching algorithms based on bottom-up automata is presented. The algorithms cover a range of alternatives from fast algorithms with large tables to slower algorithms with small tables. Some of the algorithms are well known [HoO82], and others are simple, but useful, improvements to known ones (Section 3.5 improves an algorithm by Chase [Cha87]), and yet others, like the CTF subpattern B-fsa algorithm (Section 3.4), are new and could be of interest for applications requiring small tables and reasonable matching time. The exact table and size comparisons are application-dependent and will be the subject of future research. Chapter 3 also contains some results providing a better understanding of the notions involved in bottom-up pattern matching and shows how to determine which match sets will be useful when the input tree belongs to a given recognizable set, which seems a new result and is quite useful.

Linear N-patterns can only describe very local conditions. This dissertation contains several ways in which to extend this notion. A first extension are the non-linear N-patterns, in which variables are still restricted to have arity 0 but may be repeated. Chapter 4 considers these patterns and reports on a family of algorithms that solve non-linear pattern matching by combining linear pattern matching with testing of semantic routines. Linear pattern matching is done using the algorithms of Chapter 3; the result of testing semantic routines can be "folded" into the

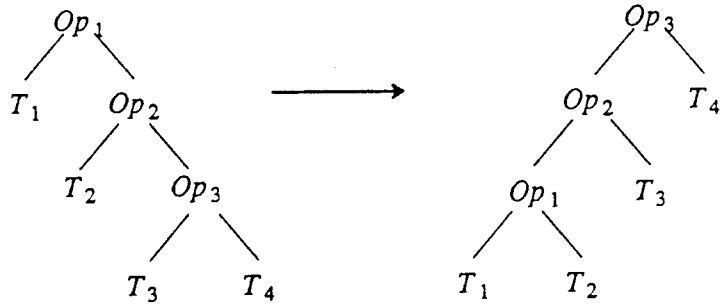
states used by these algorithms to avoid recomputing some predicates. This folding allows the implementer to trade table size against algorithm speed. The results in this chapter provide a practical approach to non-linear pattern matching, but they have not been implemented and it is not known how well they perform in practice. None of the algorithms attain the goal of a practical algorithm that does at most as many semantic tests as edges are present in a dag representation of the input tree.

The algorithms in Chapter 4 can either assume a dag representation of the input tree and perform tests for subtree equality by pointer comparison, or do full subtree equality tests. A problem with the dag representation is that local changes to the tree may produce global changes to its corresponding dag. This might preclude the use of a dag representation in an application such as a rewrite system. Performance measurements would be needed to make a final decision on this issue.

The extension to the notion of pattern is a typed N-pattern: a normal N-pattern extended with a restriction on the valid values that can be assigned to the variables. Section 7.1 shows how to perform pattern matching for linear typed N-patterns by modifying the B-fsa used in Chapter 3. The techniques of Chapter 3 can be used to restrict the input set to be a recognizable set.

The final extension considered in the dissertation are X-patterns, patterns in which variables may have non-zero arity. X-patterns are closely related to some special typed N-patterns, but they name and identify portions of the pattern; they can also be typed. Section 7.1 shows how to do pattern matching for typed linear X-patterns by using the techniques of Chapter 5 and solving a REACHABILITY problem. Recognizable sets as input sets can be taken into account using the same techniques.

The extensions of patterns considered above are necessary to describe non-trivial tree transformations, but may not be enough in some cases. For example, a tree transformation such as:



cannot be described in a single transformation with the extensions to patterns proposed here. Moreover, a description with several individual transformations requires complicated interactions that will reduce the readability of the description. Future research will try to extend the notion further, but note that the above transformation could be described using an input pattern typed so as to correspond to the input tree and extended so that one could name an undeterminate number of variables, which then would be composed, in a different way, in the output pattern. [Pel84] contains a sketch for several possible extensions along these lines.

9.2. BURS and Reachability

A major contribution of this dissertation is the definition of the subclass of rewrite systems called bottom-up rewrite systems, BURS, and the definition and solving of the REACHABILITY problem for them. The rewrite systems in BURS use only linear N-patterns, but they can be used to solve several useful problems.

Chapter 5 introduced the notion of a BURS and the basic algorithms for REACHABILITY for them, as well as the faster specialized versions for finite BURS. The idea behind the algorithm for REACHABILITY has been around for a while; maybe the earliest references in the context of compiler systems are the dynamic programming algorithms of [AUJ77] and [Rip78] for code generation. BURS theory differs from these early proposals in that it is based on rewrite systems, it can handle a larger class of rewrite systems, and it emphasizes the computability of the states by a bottom-up automaton. Also, the emphasis in solving fixed-goal REACHABILITY (as opposed to tracking the behavior of some rewrite system as in [Moe87]) produces fast algorithms.

Chapter 7 contains two direct applications of REACHABILITY problems. Section 7.1 shows how to solve pattern matching for typed X-patterns. The resulting algorithm seems quite fast, but it has not been implemented and measured yet. Another application of the REACHABILITY algorithm is in finding "tilings" of trees. This is the basic process underlying solving the forward application problems of the projection systems defined in Section 7.2. Inverse application problems for projection systems can also be solved using REACHABILITY. The application of REACHABILITY explored in most detail in this dissertation is code generation, which is based on solving C-REACHABILITY, which is solved in Chapter 6.

Some questions remain unanswered. The most important are faster algorithms for the generation of the LR graphs and UI-LR graphs, and algorithms for membership in finite BURS for the case that the input set is a general recognizable set.

9.3. Code Generation

The idea of using some type of dynamic programming for code generation has been around for a while (see Section 9.2 above), but it is only recently that practical proposals have been made based on this approach. Suddenly, a number of researchers independently have proposed similar algorithms. This dissertation contains another proposal but with several significant advantages. Chapter 6 shows how to modify the algorithms of Chapter 5 for code generation. To test the applicability of the theory a code generator generator was implemented, which is described in Chapter 8. The results of that chapter show that, in the code generation model of UW-CODEGEN, a BURS-based code generator is at least as fast as a non-optimal technique like Graham-Glanville, much faster than an optimal technique based on dynamic programming with an explicit manipulation of costs, and competitive in table size with all but some substantially slower approaches.

The results presented in this dissertation show the potential for BURS-based fast optimal code generation for expression trees. It is important to emphasize that the main advantage of optimality is that it frees the machine description writer from having to understand the theory used to generate the code generator. A non-optimal technique like GG may generate quite good code (see Figure 8.22) if the machine description is fine-tuned for that particular technique [Hen84].

The BURS theory was developed independently of [WeW86], [HeD87]; it differs from them in its ability to encode cost information into the δ-BURS states and in handling a larger class of rewrite systems. The results obtained are similar to those claimed by Hatcher and Christopher [HaC] and [Hat85] but while the Hatcher/Christopher technique requires modifying some parts of the machine description to retain optimality, the approach described in Chapter 6 is always optimal, provided that a finite number of states exist. It is likely that the Hatcher/Christopher technique can be explained as a simplification of BURS-theory, but descriptions available have been inadequate to do so.

Probably the best-known implementation for optimal code generation is the one based on *twig* and reported in [AGT86]. The theory behind it is quite similar to the one used in the TD code generator reported in Chapter 8 with two differences. The first difference is that the implementation of *twig* reported in [AGT86] does more computation at solving time than TD. Thus, *twig* has smaller tables and smaller table generation times, but larger code generation times. The

second difference is in the phase organization. Both *twig* and UW-CODEGEN perform two types of transformations: some transformations do normalization and simplification, as in the mapping of short-circuit booleans into compare and jumps, the others are the ones discussed in this article and correspond to the machine instructions. *Twig* deals with both types of transformations together in a single mechanism, but the interaction of the machine rewrite rules with the simplification routines makes it possible to write looping and non-optimal transformations. UW-CODEGEN first performs the normalization and simplification and then the machine rewrite rules, but the simplification routines can query the machine description to make decisions. Although there are no specific measures comparing our approach and *twig*, it is safe to say that BURS-based code generation is substantially faster than one based on *twig*.

In addition to just solving the code generation problem, the δ-LR states contain information that could be used to help the design of some parts of the instruction set of target machines. In particular, one can determine whether the addition of a feature will affect the quality of the generated code for any valid input tree.

There are several open questions in this area. One important experiment that we did not perform would be to quantify the effect of the input set on table size. This could be done by implementing the algorithm presented in Section 5.5. Another experiment of interest would be to generate tables to minimize lexicographic cost and compare their size with the ones currently generated for the approximation to lexicographic cost; probably the benefits in the cost of the generated code are not going to be worth the increase in table size. Finally, it would be very useful to have a faster table generator. This would require a better implementation and, most likely, new algorithms to generate the states.

9.4. Bottom-Up and Top-Down Pattern Matching

At first observation, it is not clear whether top-down or bottom-up pattern matching is the better approach. Proponents of top-down pattern matching point out its small table size and its table-generation speed, those of bottom-up pattern matching, its solving-time speed.

The results of [HeD87] suggest that, at least in the context of code generation, the resulting table sizes of fast top-down pattern matchers are not significantly smaller than what we can obtain for bottom-up pattern matchers using recent technology (such as Chase's algorithm [Cha87] as modified in Section 3.5). Similarly, Chase's algorithm to generate bottom-up pattern matchers is fast enough for all the applications with which we are familiar.

Still in the context of code generation, this dissertation has shown how to combine the information required for bottom-up pattern matching together with information encoding rewrite sequences and cost information into a single state, but we do not know how to do this for top-down pattern matching. Combining all this information into a single state allows for very fast code generation solving times.

If table size is a big concern, it appears certain that an implementation using either subpattern B-fsa, or a faster variation like CTF subpattern B-fsa (Chapter 3) will produce very small tables, maybe comparable to the sizes of top-down pattern matchers.

Another significant advantage of bottom-up pattern matchers is that they mix well with recognizable sets (Def. 2.5). Chapter 3 showed how to restrict the match set to those that are used in trees of a recognizable set, and Chapter 5 did a similar thing for BURS, but it is not clear how a top-down pattern matcher could exploit such a restriction on the input set.

9.5. Compiler Phases

The dissertation has shown how to describe several problems in compiler systems as a few problems involving transformation on trees. It has also shown that some of these tree transformations can be described using BURS and solved very efficiently. The application to code

generation has been explored quite extensively in this dissertation. The use of BURS in solving application problems in projection systems (Def. 7.3) needs additional work. Currently, the most appealing application is to defining the the mapping back and forth between parse trees and abstract syntax trees. Projection systems are quite similar to the tree-to-tree grammars of [KMP84] but with a cleaner definition and with the ability for inversion. They are not as powerful, but some of the extensions suggested in Section 7.2 seem to fill the gap.

Another important open question is how useful are the extensions of patterns introduced (typed N-patterns and typed X-patterns) for defining complex tree transformations. A specific example for future investigation are the machine independent and machine dependent rewrites of TTS in Codegen [AGH84]. It seems likely that a proper description will require the use of further extensions to the notion of pattern.

Bibliography

The best ideas are common property

[Lucius Annaeus Seneca [4 B.C.-A.D.65]]

[AhU73]
A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation and Compiling, Volumes 1 and 2*, Prentice Hall, Englewood Cliffs, NJ, 1973.

[AHU75]
A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, MA, 1975.

[AhJ76]
A. V. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees", *Journal of the ACM 23*, 3 (July 1976), 488-501.

[AhU77]
A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison Wesley, Reading, MA, 1977.

[AUJ77]
A. V. Aho, J. D. Ullman and S. C. Johnson, "Code Generation for Expressions with Common Subexpressions", *Journal of the ACM 24*, 1 (January 1977), 146-160.

[AJU77]
A. V. Aho, S. C. Johnson and J. D. Ullman, "Code generation for machines with multiregister operations", in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, January 1977, 21-28pp.

[AGT86]
A. V. Aho, M. Ganapathi and S. W. K. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming", Submitted to ACM Transactions on Programming Languages and Systems, January 1986.

[AGH84]
P. Aigrain, S. L. Graham, R. R. Henry, M. K. McKusick and E. Pelegri-Llopart, "Experience with a Graham-Glanville Style Code Generator", *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices 19*, 6 (June 1984).

[Bal83]
R. A. Ballance, Generalized Language-Based Editors. Dissertation Research Proposal, May 1983.

[BVG87]
R. A. Ballance, M. L. Van De Vanter and S. L. Graham, "The Architecture of Pan I", PIPER Working Paper, Computer Science Division, EECS, University of California, Berkeley, CA, Summer 1987.

[BBG87]
R. A. Ballance, J. Butcher and S. L. Graham, "Incremental Syntax Analysis in a Language Based Editor", Unpublished Document, Computer Science Division, EECS, University of California, Berkeley, CA, November 1987.

[BDK86]
R. Brayton, E. Detjens, S. Krishna, P. McGeer, T. Ma, L. Pei, N. Phillips, R. Rudell, R. Segal, A. Wang, R. Yung and A. Sangiovanni-Vincentelli, "Multiple-Level Logic Optimization System", in *Proceedings of IEEE International Conference on CAD*, Santa Clara, California, November 1986.

[Cha82]
G. J. Chaitin, "Register Allocation and Spilling via Graph Coloring", *Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices 17*, 6 (June 1982), 98-105.

[Cha87]
D. R. Chase, "An Improvement to Bottom-up Tree Pattern Matching", in *Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages*, Munich, Germany, January 1987.

[Chu41]
A. Church, "The Calculi of Lambda Conversion", *Annals of Mathematical Studies #6*, Princeton, New York, 1941.

[CoS70]
J. Cocke and J. T. Schwartz, *Programming Languages and Their Compilers*, Courant Institute of Mathematical Sciences, New York University, April 1970.

[DEC81]
Digital Equipment Corporation, *VAX-11/780 Architecture Handbook*, Digital Equipment Corporation, 1981.

[DaK77]
R. Davis and J. King, "An Overview of Production Systems", in *Machine Intelligence 8*, E. W. E. D. Michie (editor), Ellis Horwood Limited, Edinbourgh, 1977, 300-332.

[DeR69]
F. L. DeRemer, "Transformational grammars for languages and compilers", TR50, Computing Laboratory, University of Newcastle-on-Tyne, 1969.

[Der83]
P. Deransart, "Logical Attribute Grammars", in *Information Processing 83*, R. E. A. Mason (editor), Elsevier Science Publishers, 1983, 463-469.

[DGR87]
E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang, "Technology Mapping in MIS", University of California, Berkeley-EECS Technical Report, September 1987.

[Eng75]
J. Engelfriet, "Tree Automata and Tree Grammars", DAIMI Report FN-10, Department of Computer Science, University of Aarhus, Denmark, April 1975.

[For79]
C. L. Forgy, *On the Efficient Implementation of Production Systems*, PhD Dissertation Carnegie-Mellon University, February 1979.

[GaF82]
M. Ganapathi and C. N. Fischer, "Description-Driven Code Generation Using Attribute Grammars", *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages*, Albuquerque, NM, January 1982, 108-119.

[GaG84]
H. Ganzinger and R. Giegerich, "Attribute Coupled Grammars", *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices 19*, 6 (June 1984).

[GaJ80]
M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., 1980.

[GlG78]
R. S. Glanville and S. L. Graham, "A New Method for Compiler Code Generation (Extended Abstract)", *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, Tucson, AZ, January 1978, 231-240.

[GMW80]
I. Glasner, U. Moencke and R. Wilhelm, "OPTRAN, A Language for the Specification of Program Transformations", *6th Workshop on Programming Languages and Program Development*, Universitat des Saarlandes, 1980.

[GKM83]
S. L. Graham, P. B. Kessler and M. K. McKusick, "An Execution Profiler for Modular Programs", *Software—Practice & Experience 13* (August 1983), 671-685.

[Gue81]
I. Guessarian, "On Push Down Tree Automata", in *Proceedings 6th Colloquium on Algebra, Automata, and Programming*, E. Astesiano and C. Bohm (editor), Genoa, 1981, 211-223.

[Har78]
M. A. Harrison, *Introduction to Formal Language Theory*, Addison Wesley, Reading, MA, 1978.

[Hat85]
P. J. Hatcher, *A Tool for High-Quality Code Generation*, PhD Dissertation, Illinois Institute of Technology, Chicago, Illinois, December 1985.

[HaC]
P. J. Hatcher and T. W. Christopher, "High-Quality Code Generation Via Bottom-Up Tree Pattern Matching", *POPL86*, .

[Hen82]
R. R. Henry, "Building Compilers with pcc2", Unpublished manuscript, Computer Science Division, EECS, University of California, Berkeley, CA, April 1982.

[Hen84]
R. R. Henry, "Graham-Glanville Code Generators", Phd Dissertation, Report No. 84/184, Computer Science Division, EECS, University of California, Berkeley, CA, May 1984.

[HeD86]
R. R. Henry and P. C. Damron, *An Experiment in Code Generation*, University of Washington Technical Report, June 21st, 1986.

[HeD87]
R. R. Henry and P. C. Damron, "Code Generation Using Tree Pattern Matchers", Technical Report 87-02-04, University of Washington, February 10, 1987.

[HoO82]
C. M. Hoffman and M. J. O'Donnell, "Pattern Matching in Trees", *Journal of the ACM 29*, 1 (January 1982), 68-95.

[Hoo86]
R. Hoover, "Dynamically bypassing copy rule chains in attribute grammars", *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages*, Tampa Bay, FL, January 1986, 14-25.

[HuO80]
G. Huet and D. C. Oppen, "Equations and Rewrite Rules: A Survey", in *Formal Languages: Perspectives and Open Problems*, B. R. (editor), Academic Press, 1980, 349-405.

[Hue80]
G. Huet, "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems", *Journal of the ACM 27*, 4 (October 1980), 797-821.

[Joh77]
S. C. Johnson, *A Tour Through the Portable C Compiler*, Bell Laboratories, Murray Hill, NJ, 1977.

[Joh78]
S. C. Johnson, *YACC: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, NJ, July 1978.

[Johes]
S. C. Johnson, A portable Compiler: Theory and Practice, January 1978.

[Kam3]
T. Kamimura, "Tree Automata and Attribute Grammars", *Proceedings of the Conference on Automata, Languages, and Programming 17*, 4 (March 1982), 374-384.

[KMR72]
R. M. Karp, R. E. Miller and A. L. Rosenberg, "Rapid Identification of Repeated Patterns in Strings, Trees and Arrays", *Proceedings 4th Annual ACM Symposium on Theory of Computing*, 1972, 125-136.

[Kas80]
U. Kastens, "Ordered Attributed Grammars", *Acta Informatica 13* (1980), 229-256.

[KSP83]
M. Katevenis, R. Sherburne, D. Patterson and C. Sequin, "The RISC II Micro-Architecture", in *Proceedings of International Conference on VLSI '83*, North Holland, Trondheim, Norway, August 1983, 349-359.

[KMP84]
S. E. Keller, S. P. Mardinly, T. F. Payton and J. A. Perkins, "Tree Transformation Techniques and Experiences", *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices 19*, 6 (June 1984).

[Kes84]
P. B. Kessler, "Automated Discovery of Machine-Specific Code Improvements", Phd Dissertation, Report No. 84/213, Computer Science Division, EECS, University of California, Berkeley, CA, November 1984.

[KDR78]
J. Kleer, J. Doyle, C. Rich, G. L. Steele and G. J. Sussman, "AMORD, A Deductive Procedure System. -A Rule Based System for Problem Solving Writing-", MIT AI Lab 435, January 78.

[Knu68]
D. E. Knuth, "Semantics of context-free languages", *Mathematics Systems Theory 2* (1968), 127-145.

[KnB70]
D. E. Knuth and P. B. Bendix, "Simple Word Problems in Universal Algebras", in *Computational Problems in Abstract Algebra*, J. Leech (editor), Pergamon Press, 1970, 263-297.

[Knu73]
D. Knuth, *The Art of Computer Programming, Vol. 1*, Addison-Wesley, Reading, Mass., 1973.

[Kro75]
H. H. Kron, "Tree Templates and Subtree Transformational Grammars", PhD Dissertation, Information Sciences Department, UC Santa Cruz, December 1975.

[LeS75]
M. E. Lesk and E. Schmidt, "LEX - A Lexical Analyzer Generator", Computer Science Technical Report TR-39, Bell Laboratories, Murray Hill, NJ, October 1975.

[Mck84]
K. M. Mckusick, "Register Allocation in Table Driven Code Generators", PhD Dissertation, Report No. 84/214, Computer Science Division, EECS, University of California, Berkeley, CA, November 1984.

[MWW84]
U. Moencke, B. Weisgerber and R. Wilhelm, "How to Implement a System for the Manipulation of Attributed Trees", *Workshop on Programming Languages and Programming Development*, Zurich, 84.

[Moe87]
U. Moencke, "Simulating Automata for Weighted Tree Reductions", Universitat des Saarlandes, Technical Report A 10/87., August 1987.

[Mot82]
Motorola, *MC68000 16 Bit Microprocessor User's Manual, 3rd Edition*, Prentice Hall, Englewood Cliffs, NJ, 1982.

[Moz83]
M. Mozota-Coloma, "Certaines Methodes de Preuve de Terminaison des Systemes de Re ecriture", Technical Report 362, IMAG, January 1983.

[NMS83]
J. R. Nestor, B. Mishra, W. L. Scherlis and W. A. Wulf, "Extensions to Attribute Grammars", TL 83-36, Tartan Laboratories, April 1983.

[PaW78]
M. S. Paterson and M. N. Wegman, "Linear Unification", *Journal of Computer and System Sciences 16* (1978), 158-167, Academic Press.

[Pel80]
E. Pelegri-Llopart, "A Model for Attribute Evaluation", Internal Report, Universidad Simon Bolivar, April 1980.

[Pel84]
E. Pelegri-Llopart, "Describing Tree Transformations in Compiler Systems", Unpublished Ph.D. research proposal, University of California, Berkeley-EECS, November 1984.

[Pel] E. Pelegri-Llopart, "Optimal Plan Assignment in Attribute Grammar Evaluators", Submitted for publication in ACM Transactions on Programming Languages and Systems, ACM.

[PuB87]
P. W. Purdom, Jr. and C. A. Brown, "Tree Matching and Simplification", *Software—Practice & Experience 17*, 2 (February 1987), 105-115.

[Rai80]
K. J. Raiha, "Bibliography on Attribute Grammars", *SIGPLAN Notices 15*, 3 (March 1980), 35-44.

[Rai81]
K. Raiha, "A Space Management Technique for Multi-Pass Attribute Evaluators", Technical Report A-1981-4, Department of Computer Science, University of Helsinki, Finland, Oct, 2, 1981.

[RND77]
E. M. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall Inc., 1977.

[RMT86]
T. Reps, C. Marceau and T. Teitelbaum, "Remote Attribute Updating for Language-Based Editors", *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages*, Tampa Bay, FL, January 1986.

[Rim85]
K. Rimey, "Incremental Preprocessing for Bottom-Up Tree Pattern Matching", Term Paper, CS265, University of California, Berkeley, May 1985.

[Rip78]
K. Ripken, "A Formal Method for Describing Machine Code Generation with Local Optimizations.", in *Le Point sur la Compilation*, Montpellier, IRIA., 1978, 247-306.

[Rob65]
J. A. Robinson, "A Machine Oriented Logic Based on the Resolution Principle", *Journal of the ACM 12*, 1 (January 1965), 23-41.

[Ros73]
B. K. Rosen, "Tree-Manipulating Systems and Church-Rosser Theorems.", *Journal of the ACM 20*, 1 (January 1973), 160-187.

[Sny82]
L. Snyder, "Recognition and Selection of Idioms for Code Generation", *Acta Informatica 17* (1982), 327-348.

[Tha67]
J. W. Thatcher, "Characterizing Derivation Trees of Context-Free Grammars through a Generalization of Finite Automata Theory", *Journal of Computer and System Sciences 1* (1967), 317-322.

[Tha75]
J. W. Thatcher, "Tree Automata: An Informal Survey", in *Recent contributions to the theory of computing.*, A. Aho (editor), Prentice–Hall., 1975, 143-172.

[Tji] S. W. K. Tjiang, "Twig Reference Manual", Technical Report, AT&T Bell Laboratories, Murray Hill, NJ.

[WeW86]
B. Weisgerber and R. Wilhelm, *Two Tree Pattern Matcher for Code Selection (Including Targeting)*, Technical Report, Universitat des Saarlandes, Saarbrucken, W. Germany, February 1986.

[Wir71]
N. Wirth, "The Design of a PASCAL Compiler", *Software-Practice and Experience 1* (1971), 309-333.

[WJW75]
W. Wulf, R. Johnsson, C. Weinstock, S. Hobbs and C. Geschke, *The Design of an Optimizing Compiler*, American Elsevier Computer Science Library, 1975.

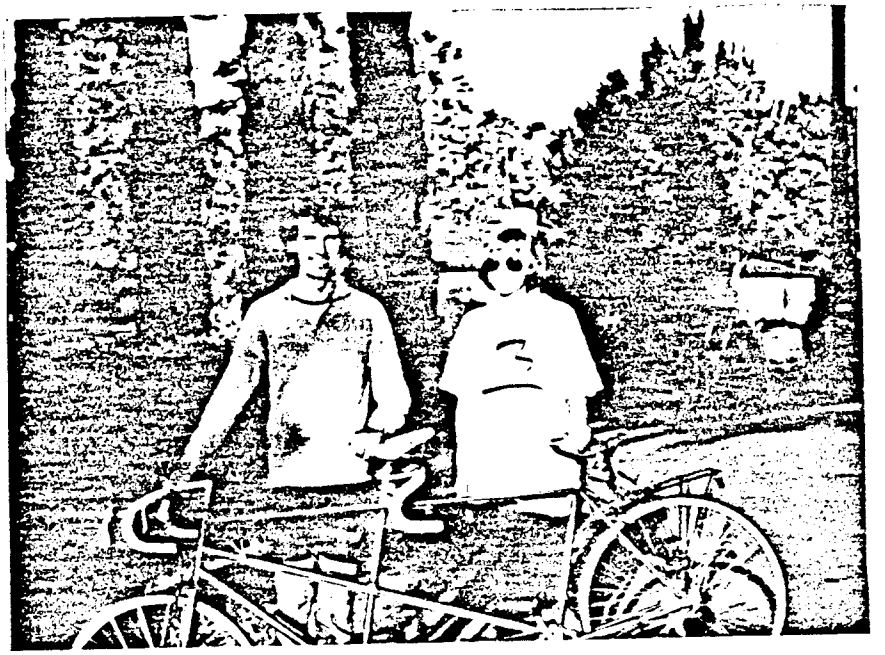
[Wul81]
W. A. Wulf, "BONSAI, A Tree Transformation Language", Unpublished Technical Report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, May 1981.

Glossary of Symbols

Symbol	Meaning	Page
$a_1 \bullet \cdots \bullet a_n$	Individual components of a sequence	18
ε	Empty sequence	18
$s_1 // s_2$	Concatenation	18
$length\,(s\,1)$	Length of a sequence	18
$T_{@p}$	Subtree of T at position p	20
$fr\,(T)$	Frontier of T	20
$height\,(T)$	Height of T	20
$Vars(\rho)$	Set of variables of ρ	22
$T_{@p \leftarrow t}$	Replacement in T of position p by t	23
$T_{X@p \leftarrow t}$	Replacement of X at p in T by p	23
$T_{X \leftarrow t}$	Replacement of X in T by t	23
$\sigma(\rho)$	Application of the substitution σ to the pattern ρ	24
$\rho_1 \equiv \rho_2$	ρ_1 is equivalent to ρ_2	24
$\rho_1 \sim \rho_2$	ρ_1 is independent of ρ_2	43
$\rho_1 // \rho_2$	ρ_1 is inconsistent with ρ_2	43
$\rho_1 \geq \rho_2$	ρ_1 subsumes ρ_2	43
Π_F	$\equiv$-reduction of the set of all subpatterns of F	43
$\rho_1 >_i \rho_2$	ρ_1 immediately subsumes ρ_2	43
I_ρ	Immediate subsumption set of ρ	43
$\rho_1 \oplus \rho_2$	Meet of ρ_1 and ρ_2	44
$M(\rho)$	Set of subpatterns subsumed by ρ	45
ρ_σ	Meet of all patterns in match set σ	45
$G >_i$	Immediate subsumption graph	46
$P_{op,i}$	Auxiliary notion used for Chase's algorithm	61
$R_{\sigma,op,i}$	$\sigma \cap P_{op,i}$	61
$P_\rho(T)$	Binding predicate of pattern ρ	69
Z_F	$\equiv$-reduction of the set of all structures of F	70
ΠZ_F	Equivalent to Π_{Z_F}	70
Σ_F	$\equiv$-reduction of the set of the meets of all match sets in F	70
$Z\Sigma_F$	$\equiv$-reduction of the set of all structures of patterns in Σ_F	70
ζ_ρ	Structure of pattern ρ	70
ζ_σ	Structure of match set σ	70
$Z\Sigma_F^{\oplus}$	Closure of $Z\Sigma_F$ under $\oplus$	74
$DS\,(\gamma)$	Discrimination set of the structure γ	74
$<\zeta,E>$	P-pattern with structure ζ and set of equations E	80

$P_{\rho,\phi}(T)$	Incremental binding predicate of ρ and ϕ	81
$DS(\phi)$	Discrimination set of the p-pattern ϕ	81
$\boldsymbol{EF}_{R,G}$	Extended pattern set of $\boldsymbol{R}$ and G	103
$\boldsymbol{I}_{R,G}$	Input patterns of $\boldsymbol{R}$ and G	103
$\boldsymbol{O}_{R,G}$	Output patterns of $\boldsymbol{R}$ and G	103
$\boldsymbol{M}_{R,G}$	Intermediate patterns of $\boldsymbol{R}$ and G	103

Index



Daisy, Daisy, give me your answer, do!
I'm half crazy, all for the love of you!
It won't be a stylish marriage,
I can't afford a carriage,
But you'll look sweet upon the seat
Of a bicycle built for two!

[Harry Dacre d. 1922]

Prepared by the author using the text processing tools *vi*, *(gnu)-emacs*, *spell*, *awk*, *sed*, *lpr*, *m4*, *bib*, *dtbl*, *deqn*, and *dtroff* with the *–me* macro package, coordinated by *csh*, *soelim*, and *nmake*. Printed in Adobe Inc.'s Times, on an Apple LaserWriter Plus. Processed mostly from pine.Berkeley.EDU, a Sun-3/75.