Extending Attribute Grammar and Type Inference Algorithms

Janet Ann Walz
Ph. D. Thesis

TR 89-968
February 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

EXTENDING ATTRIBUTE GRAMMAR AND TYPE INFERENCE ALGORITHMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Janet Ann Walz

January 1989

EXTENDING ATTRIBUTE GRAMMAR AND TYPE INFERENCE ALGORITHMS

Janet Ann Walz, Ph.D.
Cornell University 1989

Gated attribute grammars and error-tolerant unification expand upon the usual views of attribute grammars and unification. Normally, attribute grammars are constrained to be noncircular; gated attribute grammars allow fairly general circularities. Most unification algorithms do not behave well when given inconsistent input; the new unification paradigm proposed here not only tolerates inconsistencies but extracts information from them. The expanded views prove to be useful in interactive language-based programming environments. Generalized unification allows the environment to help the user find the sources of type errors in a program, while gated attribute grammars allow the environment to provide an interpreter for incremental reevaluation of programs after small changes to the code.

The defining feature of gated attribute grammars is the appearance of a gate attribute (indicating where cycle evaluation should begin and end) within every cycle. Attributes are ordered by collapsing strongly connected components in the dependency graph and topologically sorting the result. The smaller dependency graph for each component (ignoring edges leading to the gate) can be recursively collapsed to provide further ordering. Use of the evaluation order defined in this manner allows gated attribute grammars to do without the restrictions on functions within a component needed by the other varieties of circular attribute grammars. Initial and incremental evaluation algorithms are given, as well as a sample grammar allowing an editor for a small language to become an incremental interpreter.

Counting unification defines unique solutions to sets of input equations that contain conflicting type information. These solutions are derived from the potential variable constraints implied by the input equations. For each type variable, each branch (a portion of a constraint) is assigned a weight indicating the number of times the input set implied such a constraint. When the input equations are derived from the static analysis of a program, the relative branch weights for a conflicting variable give the overall pattern of uses of that variable and can direct attention to parts of the program that disagree with the majority of uses. A number of error-tolerant unification algorithms are presented.

Biographical Sketch

Janet Walz was born in Ann Arbor, Michigan in 1961, but she chose to attend Michigan State University in East Lansing. She graduated with a B.S. in computer science (with minors in geophysics and mathematics) in June 1983. She proceeded to Cornell University in Ithaca where she received an M.S. in computer science (with a minor in mathematics) in January 1986 and a Ph.D. in January 1989.

Acknowledgments

First, of course, I would like to thank my advisor, Greg Johnson, for his encouragement and for his many useful comments. I would also like to thank Chris Buckley for attempting to translate my dissertation into English, Jack Callahan for invaluable help with the text formatting system, and Bill Pugh for assorted comments. I also appreciate the facilities provided to me by the University of Maryland at College Park for the past two years.

This work was partially supported by a National Science Foundation Graduate Fellowship and a grant from the Air Force Office of Scientific Research. Portions of this work earlier appeared in conference proceedings of the Association for Computing Machinery.

Table of Contents

List Of Figures

Chapter One

Introduction

Gated attribute grammars and error-tolerant unification expand upon the usual views of attribute grammars and unification. Normally, attribute grammars are constrained to be noncircular; gated attribute grammars add a fairly general form of circularity to this standard approach. Most unification algorithms do not behave well when given inconsistent input; the new unification paradigm proposed here not only tolerates inconsistencies but extracts information from them. The expanded views prove to be useful in interactive language-based programming environments. The generalized unification allows the environment to help the user find the sources of type errors in a program, while gated attribute grammars allow the environment to provide an interpreter for incremental reevaluation of programs after small changes to the code.

The following sections provide something of the flavor of these extensions and their possible uses in such environments. Subsequent chapters provide details of the extensions and relate them to previous work. Chapter Two reviews previous work on attribute grammars, including the nonlocal productions which are used to implement the novel feature of gated attribute grammars; Chapter Three describes gated attribute grammars in detail and relates them to earlier versions of cyclic attribute grammars. Chapter Four provides an introduction to unification and type inference, while Chapter Five chronicles some attempts to produce error-tolerant unification algorithms. Chapter Six probes the meaning of type constraints on the way to a definition of conflict-tolerant counting unification. Chapter

Seven shows how to apply the ideas developed in Chapter Six to recursive type equations. Some concluding remarks then appear in Chapter Eight.

1.1. Gated Attribute Grammars

The strengths of attribute grammars as a basis for programming environments are well known. Several environment research projects rely on attribute grammars to support incremental semantic analysis ([ReT84], [FJM83], [BaS86]). Attribute grammars provide a high-level, declarative style for describing the static semantics of programming languages. Further, such descriptions are amenable to automated analysis and the production of programming environments that incrementally perform various semantic analyses of programs. Such automatically generated environments have several desirable properties; even though the writer of an attribute grammar need not be concerned with dynamic editor-time issues such as order of attribute evaluation, an editor generator can produce environments that incrementally reevaluate attribute values in an optimal fashion after user changes to the program ([RTD83]).

Attribute grammars have traditionally been required to be noncircular; that is, no parse tree derivable from the associated context-free grammar is allowed to give rise to cyclical or circular functional dependencies among attributes. In fact, the main result of the seminal paper on attribute grammars, [Knu68], is an algorithm for testing attribute grammars for noncircularity. However, several researchers have discovered applications in which relaxation of the requirement of noncircularity has given rise to natural, elegant solutions. Among these applications are instruction selection for code generation ([Ske78]), control and data flow analyses ([Far86]), and VLSI design problems ([JoS86]). Attention has recently been given to incremental attribute evaluation in the presence of circular functional dependencies in order to make it possible for programming environments to be based on such circular attribute grammars. Results obtained previously have imposed various restrictions such as monotonicity of evaluation functions and domains that are lattices of finite height ([JoS86]) in order to assure termination of the evaluation process. However, there are some important classes of functions, such as those found in interpreters faced with the possibility of infinite loops in user programs, for which termination is not assured. The work described in these chapters is motivated by a desire to obtain optimal or near optimal incremental reevaluation behavior for this important category of problems.

If a user is manipulating a fairly large or computationally expensive program and asks for it to be executed by an interpreter in the environment, then slightly modifies the program and asks for another interpretation, the program should be reinterpreted in a minimal way, preserving as much information as possible from the previous execution, rather than being reinterpreted from the beginning. This approach contrasts, for instance, with the technique of using a noncircular attribute grammar to produce code that is then interpreted or directly executed; in that technique, code is incrementally kept in agreement with the user's program, but interpretation or execution always starts from the beginning rather than making use of the results of previous interpretations. In another related approach ([BMS87]), interpreters and debuggers are automatically generated from denotational descriptions of programming language semantics, but subsequent reinterpretations of possibly slightly modified programs again result in reexecution from the beginning.

Incremental reevaluation can provide major savings when an application is organized as a sequence of phases. If the application has seven phases, for example, changing something in phase six means that only the last two phases need to be reexecuted, since the results from the first five phases are already on hand. Applications that are naturally organized in this manner include such diverse examples as compilers, numerical analysis programs, and combinations of independent tools into a single pipeline.

The gated attribute grammar approach provides a general, rigorous, generator-based framework that addresses many of the same issues in incremental reevaluation as the hand-coded system of [KaW87]. The approach gives rise to a very appealing spreadsheet-like style of programming, in which every program modification causes fast update of the contents of input/output windows that show the result of program execution on sample inputs.

The use of run-time stores (sets of <variable,value> pairs) as attributes in this new approach is very similar to the use of compile-time symbol tables (sets of <variable,type> pairs) in other attribute grammar systems. This similarity allows the multiple distinct store-valued attributes in a parse tree to be managed in the same way as aggregates like symbol tables have been ([Hoo87]), which relieves the system of the burden of maintaining many slightly different copies of the same information.

An implementation of these ideas as expressed in gated attribute grammars was built on the POE system ([FJM83]), an attribute-grammar-based programming environment. The addition of run-time semantics via circular attribute grammars permits automatically generated environments to be fairly complete, in that incremental static semantic checking and fast incremental execution are now available within a single framework.

1.2. Error-Tolerant Type Inference

A crucial aspect of a program intended for general use is its behavior in the presence of erroneous inputs. For instance, much attention has been devoted to the problems of error detection, reporting, and correction in compilers ([FMM79], [GHJ79]). As programming languages and systems based in one way or another on unification (originally proposed by [Rob65]) become more common, it becomes increasingly important to develop a theory of error detection and correction for unification-based systems.

In translating common programming languages like Pascal ([JeW78]), in which every user-declared constant, variable, or function may have exactly one fixed type, the common practice is to assume that the first occurrence of the identifier in the program -- the declaration, in Pascal -- gives the correct type for that identifier. All subsequent occurrences of the identifier in appropriate scopes are compared against the declaration to assure consistency. However, it may be that the programmer declared the identifier as one type, but during further development of the algorithm the programmer then started consistently "misusing" the identifier as a type more appropriate to the algorithm than the one which was anticipated at declaration time. In such a case, it is easy to argue that it is the declaration which is in error and the uses which are correct. Particularly when the program is being constructed in an interactive programming environment, it is easy to argue that the translating system should draw the programmer's attention to the declaration instead of accepting the declaration and marking all uses as incorrect. In the more general situation where the uses disagree among themselves, and possibly also with the declaration, the system should mark all the disagreeing type assertions, but the programmer's attention should be drawn more strongly to the occurrences which support minority type assertions as these are more likely to be incorrect.

To accomplish such a goal, we must define a notion of the number of times that an identifier is asserted to be of a given type. Defining such weights directly on a monomorphic type structure such as Pascal's would result in types such as (int $\times$ bool $\times$ int) $\rightarrow$ bool and (int $\times$ bool $\times$ bool) $\rightarrow$ bool being considered totally incompatible, in spite of clear similarities between them. Moving to a polymorphic type structure (as discussed by [Car85]), where identifier types can contain type variables instead of only type constants, we discover that both functional types are instantiations of the type (int $\times$ bool $\times$ α) $\rightarrow$ bool, where α is a new type variable. Thus we can consider the two types to be "nearly" compatible, with a conflict only in the third argument position. If these two types were both associated with the same identifier, the polymorphic view could direct attention immediately to the third argument position of the function instead of to the less specific function name, which would be the only option under the monomorphic view. By defining typing weights on a polymorphic type structure, we also gain the ability to type identifiers of openly polymorphic languages such as ML. A similar approach is found in [Wan86], where some conflicting constraints for type variables over such a structure are collected, while Snelting ([Sne86]) has considered the problems of typing expressions in a system using overloading as a more restricted form of polymorphism.

The basic paradigm for weighing type assertions holds that the weight of an assertion is the number of times that particular assertion can be independently derived from the program. To define these numbers, we collect a set of type equations which represent the constraints imposed by the various pieces of syntax in the program. We now wish to process this set to determine whether there are any conflicting type assertions and, if so, to which type variables the conflicts can be traced. Once we know which type variables participate in conflicts, we can map these variables back to the syntactic constructs which imposed the constraints. The amount of attention drawn to each such construct can be determined by whether the construct supported a majority or minority opinion in the conflict.

Consider a language-based editor for a variant of ML ([Mil78]) that uses this approach to the isolation of likely causes of user errors. As the user edits and manipulates his or her program, unification is incrementally applied to determine the program's type correctness. If a type inconsistency arises, the set of type equations can be analyzed to determine the most likely source of error. In this way a determination can be made as to the relative strengths with which the set of type equations asserts multiple contradictory hypotheses. In a language such as ML, the type of an object is inferred from patterns of usage. The type inference system for MOE ([JoW86]), an ML-oriented editor implemented using the editor-generating system PoeGen ([FJM84]), employs an extension of the ML type system in the style of [MPS84] which permits recursive types (*i.e.*, type expressions are allowed to be infinite regular trees as well as the more traditional finite trees). As noted in [MPS84], such recursive types were employed in a type inference system for Scheme. In addition to recursive types, however, MOE's type inference system provides robust error handling and the potential for much easier pinpointing of the sources of type errors.

Of paramount concern in the whole MOE project was the problem of providing helpful and exact responses to the user in the presence of type conflicts. This concern affected the design of the attribute grammar and led to a new class of error-tolerant unification algorithms. In providing error information to the user, two principles are observed:

(1) Error indications should be complete but parsimonious; the user should see highlighted on the screen everything that contributed directly to an error, but nothing more.

(2) The user's attention should be drawn to what appear to be the anomalies that are responsible for errors.

Often it is the case that most uses of a given object are mutually consistent, whereas one or a very small number of uses conflict with the general usage pattern. In MOE, if it is possible to discern that such a situation has arisen, likely errors are highlighted at high intensity and all other program components that contributed to the inferred type of the object are highlighted at a lower intensity.

Language-based editors permit a new level of quality in the process of helping users when inputs are, for one reason or another, invalid. Unification-based type inference in language-based editors appears to have been first considered by Meertens ([Mee83]). Snelting and Bahlke have more recently also explored this approach. In contrast to the approach of [BaS85], however, the unification-based type-inference scheme for MOE is expressed in the notation of attribute grammars.

The concepts of error-tolerant unification developed here are applicable outside the realm of language-based editors that perform incremental type inference; type-error detection and correction following these techniques should be easily adaptable to compilers and interpreters that perform unification-based type inference, and may have relevance more broadly in other systems that employ unification algorithms.

Chapter Two

Attribute Grammars

An *attribute grammar* is a context-free grammar in which each symbol has some number of associated attributes and each production has associated functions which define certain attributes of symbols in the production in terms of others. A example of an attribute grammar for a simple scoped expressional language is shown in Figure 2.1. In this example, the grammar symbol exp has two attributes, env, which contains the bindings for the environment in which the expression is evaluated, and val, which indicates the value of the expression.

An *attributed parse tree* is a parse tree of the context-free grammar in which each instance of a grammar symbol has instances of the attributes associated with that symbol, and the attribute instances are connected into a graph by the functional dependency relations imposed by the production

```
start ::= exp
      exp.env = λname.⊥

exp1 ::= exp2 op exp3
      exp2.env = exp1.env
      exp3.env = exp1.env
      exp1.val = exp2.val op exp3.val

exp1 ::= let id = exp2 in exp3
      exp2.env = exp1.env
      exp3.env = exp1.env[id.name←exp2.val]
      exp1.val = exp3.val

exp ::= id
      exp.val = lookup(id.name,exp.env)

exp ::= int
      exp.val = int.val
```

Figure 2.1. A Simple Attribute Grammar.

instances. When the value of each attribute instance in a tree is the same as the result of applying its evaluation function to the values of its predecessors in the dependency graph, the parse tree is said to be *consistent.* A sample consistently attributed parse tree from the expressional language can be found in Figure 2.2, with the values of the exp.env and exp.val attributes found in the boxes on the right.

2.1. Standard Attribute Grammars

Traditionally, work on attribute grammars has focused on those grammars for which every possible attribute dependency graph is acyclic. In these cases, each attribute can be classified as either synthesized or inherited. A *synthesized attribute* can be thought of as one whose value flows up the tree. More specifically, an attribute is synthesized if its evaluation functions appear in productions where its symbol is on the left-hand side. An *inherited attribute* can be thought of as one whose value flows down the tree, and its evaluation functions appear in productions with its symbol on the right-hand side. In the sample language, exp.env is inherited and exp.val is synthesized.

With an acyclic attribute grammar, a single traversal, in topological order, of the dependency graph can consistently attribute the entire tree. Each attribute's defining function will be evaluated exactly once in this initial traversal, guaranteeing that the attribution process will terminate, as long as each defining function itself terminates.

Global topological information for ordering attribute evaluation can be delivered to each instance of a grammar symbol by superior and inferior characteristic graphs. An *inferior characteristic graph* shows the projection of transitive dependencies from the portion of the parse tree below the symbol; a *superior characteristic graph* shows the projection of transitive dependencies from the remainder of the tree. These characteristic graphs, combined with

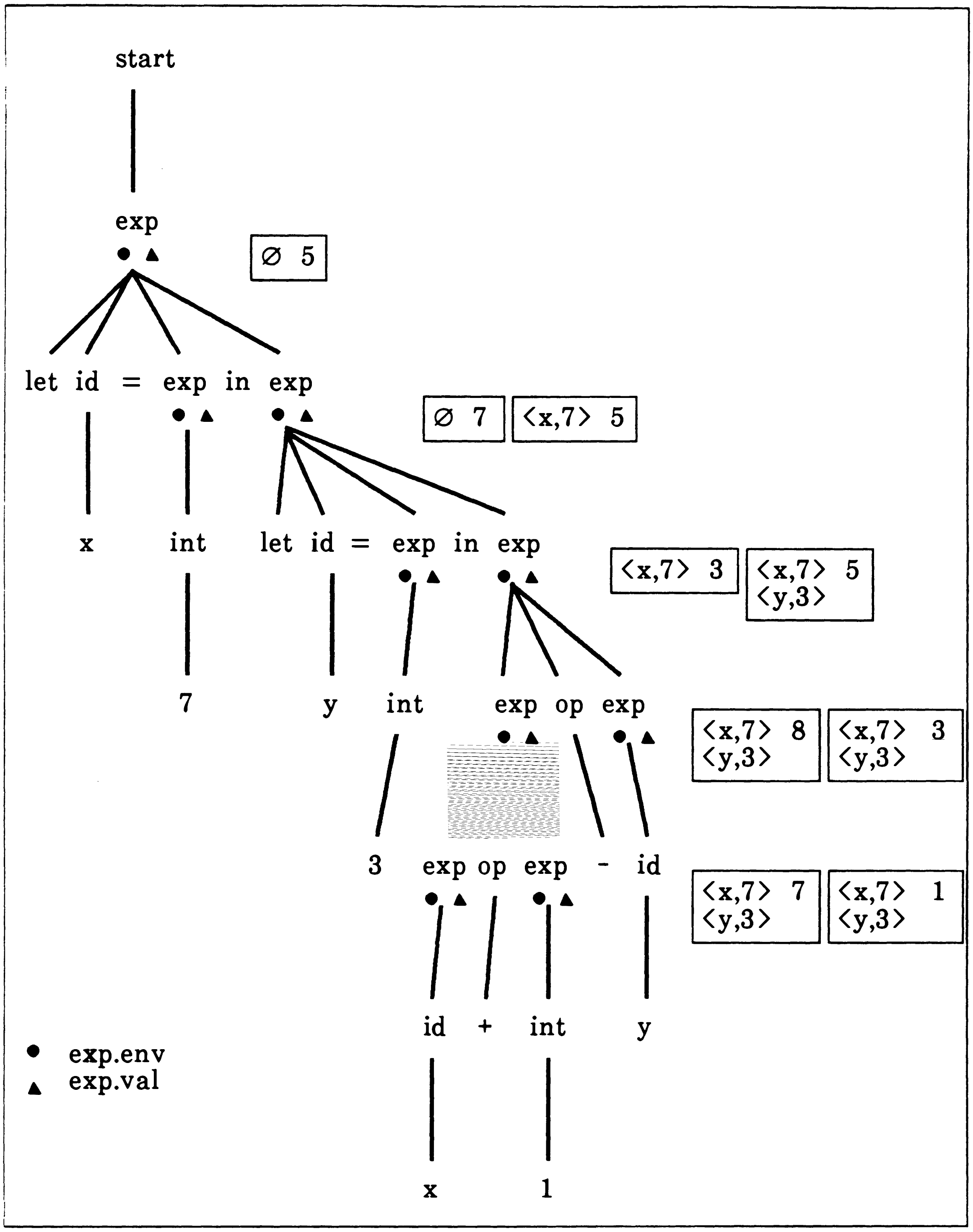


Figure 2.2. An Attributed Parse Tree.

the local dependencies imposed by production instances, allow local evaluation to proceed in accordance with global topological order.

The paradigm for changing a consistently attributed parse tree calls for replacing some subtree of the parse tree by another subtree. The two subtrees must be rooted by the same grammar symbol, and the position of that symbol instance in the new tree is called the point of intersection. The second subtree is assumed to also have internally consistent attribute values. Therefore, after the replacement, all inconsistent attributes occur in the productions immediately above and below the point of intersection. By propagating changes along the dependencies leaving these attributes, we will reach every attribute whose value in the updated parse tree, when consistently attributed, differs from that in the old tree.

Characteristic graphs allow optimal incremental updating after some portion of a tree has been changed. When superior characteristic graphs are kept for symbol instances between the cursor position, which is the point of change, and the root of the parse tree, and inferior characteristic graphs are kept for all other instances, [RTD83] show how updating after a subtree replacement can be accomplished in O(|AFFECTED|), where AFFECTED is defined as the set of attributes whose values after updating quiesces differ from their values before the editing change. (In other words, the information in the characteristic graphs can be used to avoid problems with naive change propagation, such as having to redo the propagation from an intersection attribute after discovering that one of its predecessors got a different value in the course of change propagation from another of the intersection attributes.)

If an attribute grammar is partitionable (the class of ordered attribute grammars being a polynomially-recognizable subset of the partitionable attribute grammars [Kas80]), then these incremental optimality results are obtainable without the expense of maintaining characteristic graphs at evaluation time. In this case, it is possible to wire a strategy for optimal incremental reevaluation after subtree replacement into the editor as the editor is being created.

2.2. Attribute Grammars with Nonlocal Productions

In the discussion so far, attribute grammar dependencies have followed the structure of the parse tree. Each evaluation function used inputs available in a single production to determine the value of another attribute in the same production. However, attribute grammars following this framework will typically generate long "copy chains" in the dependency graph, where each evaluation function along the chain is an identity which merely moves the value one step further along the tree structure. This situation occurs in the sample grammar when there is a large, complicated arithmetic expression inside the deepest let clause. In such a case, the environment from the deepest let clause is propagated through all the interior nodes of the arithmetic expression only so that its value can reach those leaves which are identifiers. If the value bound to an identifier via a **let** clause changes, the attributes of all these interior nodes must also be updated to get the update to the leaves where it may be used.

Motivated by a desire to remove such inefficiency, several people have developed ways of easing the restriction on dependency links. Johnson and Fischer ([JoF85]) add new symbols, called interface symbols, to some of the productions of a grammar. They also add nonlocal productions, which turn collections of these interface symbols into the empty string so that the addition of the interface symbols does not change the accepted language. These nonlocal productions, with the aid of information flowing along the tree between the interface symbols, set up and maintain nonlocal dependency links. With the aid of a priority relation on attribute instances, which is calculated at editor generation time, attribute evaluation can still be ordered to avoid evaluating an attribute more than once during an incremental update. This scheme works well when linking variable definitions to their uses, as a change in the type or value of a variable can then be propagated directly to the uses of that variable, without passing through the intermediate tree structure. The treatment of nonlocal productions was formalized in [JoF87], where the special interface symbols and productions are dropped in favor of something that looks like a traditional attribute grammar. Nonlocal dependencies are represented by chains of local dependencies through designated *imaginary* attributes, which allows global properties of the grammar to be analyzed at editor generation time by local algorithms. Projecting onto the real attributes reveals the nonlocal dependencies which are then used during the actual evaluation process.

Another approach is found in [RMT86], where an upward remote reference can replace a chain of copy attributes from a symbol's attribute to an attribute of a descendant symbol in the parse tree. More specifically, an evaluation function is allowed to reach back up the tree until it finds the first instance of a desired grammar symbol, from which it can take the value of an attribute. [Hoo86] provides yet another approach, where any copy chain can be replaced by a copy bypass dependency which provides an approximate topological ordering for attribute evaluation. Both of these latter approaches can avoid propagating the environment throughout the body of a let clause by making each use of a variable inside the body refer back to the environment produced in the let production, although for an upward remote reference the grammar must be changed slightly to give the expression that is the let body a different symbol.

In [Hoo87], Hoover provides a different way of thinking about aggregate attributes such as environments or states. Such aggregates consist of a number of independent keyed records and are manipulated by five aggregate operators: EMPTY, ADD, LOOKUP, COPY, and UPDATE. Since each of these operators has a clear effect on an input aggregate, dependency chains in which each aggregate is the result of an operator applied to the previous aggregate do not require recalculating and representing an entire new aggregate at each step. With an aggregate environment, each use of an identifier depends only on the appropriate record, not the rest of the environment. Through the use of copy bypass or copy structure trees, each use is, in effect, connected via a nonlocal dependency to the appropriate definition.

2.3. Circular Attribute Grammars

Another restriction that has recently been relaxed is the total ban on cycles in the attribute dependency graph. One of the first researchers to explore this area was Skedzeleski ([Ske78]). More recently, Jones and Simon ([JoS86]) allow arbitrary dependency graphs, but restrict the defining functions for attributes that may appear in strongly connected components (SCCs) of a dependency graph. These functions are required to be monotonic and to take values from a lattice of finite height. Under these conditions, finding a fixed point of an SCC requires only a finite number of attribute evaluations -- the value of an attribute can only change by increasing, and there are only a finite number of steps to go up in the lattice. By treating each maximal strongly connected component as an attribute in a collapsed graph for SCC scheduling, they maintain a worst-case incremental update time of $O(hk|\text{AFFECTEDSCC}|)$, analogous to that in [RTD83], where h is the height of the highest lattice and k is the largest number of nodes in an affected SCC. They also show that the local portion of the collapsed dependency graph can be constructed from the locally available characteristic graphs and production dependencies, and note the care that must be taken to ensure that the SCC fixed point found by incremental evaluation is indeed the least fixed point for that SCC.

Farrow ([Far86]) also allows arbitrary dependency graphs and restricts the defining functions in strongly connected components to be monotonic, but instead of requiring these functions to operate on finite-height lattices, he requires them to satisfy an ascending chain condition. This condition, that for every ascending chain $s_0 < s_1 < \ldots < s_k$ some $f(s_k) = f(s_{k+1})$, also guarantees that evaluation of each individual SCC will terminate, and thus that the entire evaluation process will terminate.

There is some similarity between these approaches and the data-flow technique of interval analysis as described in [ASU86]. All of them solve equations involving monotonic functions by iterating in the manner prescribed by a flow graph. The granularity of the data-flow graph, however, is much coarser than that of the attribute dependency graph. A node in a data-flow graph represents a statement or an entire basic block; a node in a dependency graph represents an attribute associated with a single grammar symbol. The edges in a data-flow graph actually represent control dependencies among the basic blocks, so there is always a unique source node where program execution begins. Similarly, natural loops have a single entry point at the top which dominates the rest of the loop. With the finer granularity of a dependency graph, these regularities no longer hold (*e.g.*, every int.val node from the sample grammar will be a source node). Therefore, connected components provide a better partitioning of the graph into areas corresponding to loops than do the dominator relationships used in interval analysis.

Further discussion of these approaches to circular attribute grammars is postponed to sections of the next chapter, after another way of approaching circularity in attribute grammars has been introduced.

Chapter Three

Gated Attribute Grammars

A new category of attribute grammars called *gated attribute grammars* provides another approach to cyclic computations. In contrast to the earlier approaches, evaluation functions do not need to be specially restricted when they occur in a cycle; the more powerful partial functions used in applications such as interpreters can be expressed directly in the grammar. In a gated attribute grammar, every cycle in an attribute dependency graph must have a *gate* attribute that specifies the location in the cycle at which evaluation is to start. To evaluate the attributes in a parse tree, one temporarily views strongly connected components as single nodes and evaluates the nodes of the collapsed graph in topological order. Evaluation of a strongly connected component involves an inductive application of this scheme: one removes incoming arcs to the gate attribute of the outer SCC (so that at least the gate attribute is no longer in the same SCC as the other nodes), identifies any nested strongly connected components, and considers these nested SCCs to be individual nodes. The evaluation of this graph then proceeds in accordance with a topological ordering on its nodes.

The distinguishing characteristic of a gate attribute is that it has two evaluation functions. The first evaluation function has as inputs only attributes outside the gate attribute's SCC and is used to provide an initial value to the gate. The second evaluation function depends on attributes in the SCC, and is used to obtain the succession of subsequent gate values. A boolean valued pseudo-attribute named *start* is associated with a gate attribute to determine which of the gate's evaluation functions should be invoked when the gate needs to be evaluated. Evaluation of the SCC is complete when (and if) it reaches a fixed point; that is, when every attribute in the SCC contains a value that is the same as the result produced by its evaluation function when it is applied to the values of the attribute's predecessors.

As with [JoS86], incremental evaluation at the outermost level can be accomplished using results from [RTD83] and subsequent refinements. A major difference, however, arises when an input to a strongly connected component "node" changes value. We do not merely want to schedule the attribute that is the successor of the changed attribute for reevaluation, since that node might be in the middle of the SCC. Rather, we also schedule the gate of the SCC for reevaluation, since that is the attribute at which evaluation of the SCC is supposed to start. At evaluator time, nonlocal productions are employed to attach inputs of an SCC to the start attribute of the SCC. The purpose of these connections is to assert that the start attribute of the SCC is an evaluator-time successor of each input to the SCC. The writer of an attribute grammar designates certain attributes as gates and gives them two evaluation functions. The whole technique results in an extended attribute grammar notation that is applicable to a large and important class of problems, including interpreters, that are most naturally expressed using circular attribute grammars, and from which efficient programming environments can be automatically created.

In the following sections the class of gated attribute grammars and the notion of gated strongly connected components are defined; this latter concept is critical to the approach presented herein. Initial and incremental evaluation algorithms for gated attribute grammars are presented; the approach is quite general, and should be readily implementable in any of a variety of attribute-grammar-based editor and compiler generating tools.

3.1. Philosophy

While there are application areas whose natural semantic functions satisfy one of the previously-imposed conditions on circular attribute grammars discussed in the preceding chapter, there are also some natural functions that are not so well-behaved. An example of such functions arises when we try to use attribute evaluation to model execution of a **while** loop. Adding an indefinitely looping construct to model while execution will give the attribute evaluation mechanism full partial recursive power to apply to other, less easily visualized problems. The small grammar found in Figure 3.1 is a natural way to express interpretation via attribute evaluation. (In this grammar, a value of not-reached for a state means that the corresponding statement would not be executed if the program were run.)

If the **while** production is ignored, this attribute grammar is acyclic. With the **while** production, a cycle is formed by the while.gate, B.state, B.val, S_2.initst, and S_2.finalst attributes, as shown in Figure 3.2, where dashed links indicate indirect dependencies. (The while.start attribute will be discussed later in this section.)

For the class of problems and algorithms considered here, it is appropriate to use the standard total ordering on the integers, rather than creating a flat cpo out of the integers. (Circular attribute grammars give rise to sets of simultaneous equations over the integers, and, as expanded upon in the next section, solutions of such systems with respect to the flat cpo would give the value $\perp$ to many of the variables.) Since there is no reason, under this ordering, to expect that the body of the **while** loop represents a monotonic function from initial states to final states, the previous work on circular attribute grammars can not define the desired final state of this loop. Execution of this loop, however, would result in a well-defined final state, provided that

```
Start ::= S
    S.initst = λname.⊥
    Start.finalst = S.finalst

S ::= id ← exp
    exp.state = S.initst
    S.finalst = if S.initst = not-reached then
                    not-reached
                else
                    S.initst[id.name←exp.val]

exp ::= id
    exp.val = lookup(id.name,exp.state)

exp ::= int
    exp.val = int.val

exp1 ::= exp2 op exp3
    exp2.state = exp1.state
    exp3.state = exp1.state
    exp1.val = exp2.val op exp3.val

S1 ::= S2 ; S3
    S2.initst = S1.initst
    S3.initst = S2.finalst
    S1.finalst = S3.finalst

S1 ::= while exp do S2
    while.gate = if while.start then
                    <1,S1.initst>     (initial value)
                 else
                    <0,S2.finalst>    (subsequent values)
    exp.state = second(while.gate)
    S2.initst = if exp.val ≠ 0 then
                    second(while.gate)
                else if first(while.gate) = 1 then
                    not-reached
                else
                    S2.initst
    S1.finalst = if exp.val = 0 then
                    second(while.gate)
                 else
                    not-reached
```

Figure 3.1. A Gated Attribute Grammar.

the loop terminates. In the next section, the usual attribute evaluation algorithm is extended in such a way that the attribution final state coincides with the execution final state.

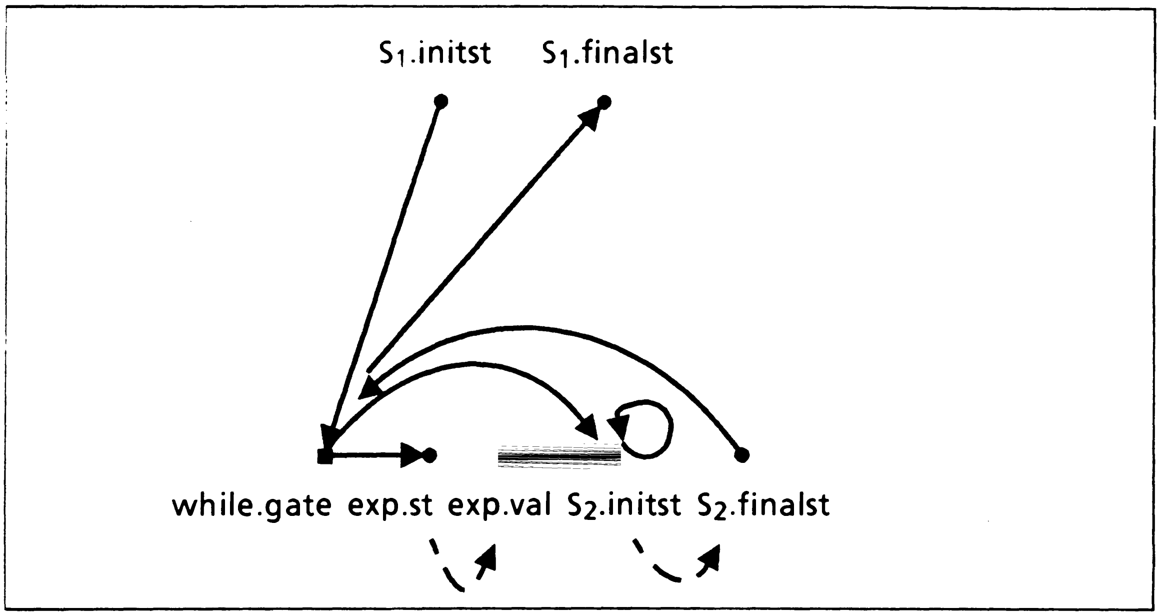


Figure 3.2. while dependencies.

This extended attribution algorithm does not require an attribute grammar to be noncircular, but it still requires some order in the attribute grammar. Every nontrivial dependency-graph cycle that can be generated by such an attribute grammar must contain at least one attribute designated as a *gate*. (Cycles such as the one for S_2.initst in Figure 3.2 with only one node in the cycle, corresponding to an attribute depending on its prior value, are considered trivial.) A gate attribute represents the point at which attribution of a strongly-connected component in the dependency graph should begin. Each gate attribute has an automatically associated *start* attribute that will indicate when the gate attribute should take its value from outside its SCC rather than from inside. In the example, while.gate is the only designated gate and while.start is its start attribute.

A gate attribute provides the means of controlling the evaluation of attributes within a strongly connected component. If all functions in the SCC are monotonic and all attributes are initially $\perp$, starting propagation from any attribute in the SCC will yield the least fixed point of the SCC; if some functions are nonmonotonic, starting propagation from different attributes can yield different fixed points, so the desired fixed point is defined to be the one resulting from propagation starting at the gate attribute. By guaranteeing that each SCC has a gate attribute, we also guarantee that each SCC has a well-defined fixed point.

Given an attribute grammar with designated gate attributes, the algorithms previously used to detect circularities can now be used with only minor modifications to detect circularities that do not pass through gate attributes. If no such circularities can be found, the attribute grammar is called a *gated attribute grammar* and can be evaluated with the techniques of this chapter.

Each gate in a dependency graph defines a *gated strongly connected component (GSCC)*, which can be considered as the region of the graph under the control of the gate. Those attributes used to calculate the gate value when its start is **true** are outside the gate's control, so edges from them to the gate are not used in determining the extent of the GSCC. Any node that remains strongly connected to the gate without these edges in the graph is in the gate's GSCC.

As a consequence of this definition, every nontrivial maximal SCC in a dependency graph of a gated attribute grammar is a GSCC and, for any gate that is not nested inside another's GSCC, the GSCC is the maximal SCC containing that gate.

The *sub-GSCC* for an interior gate node consists of those nodes in the SCC that are strongly connected to the interior gate after removing arcs to this gate from its start node and whichever other nodes contribute to the gate value when its start is **true**. This sub-GSCC is the same as the maximal SCC for the interior gate node would be if there were not an outer gate. The portion of a GSCC which is not in any sub-GSCC is called its *core*. (In the example, GSCCs correspond to **while** loops, sub-GSCCs to nested **while** loops, and the core of a GSCC to that portion of the loop outside of any nested loops.)

3.2. Initial Evaluation

Initial evaluation starts with an attributed parse tree where no attributes are guaranteed to have consistent values. Therefore, every attribute needs to be evaluated, those in nontrivial SCCs possibly more than once, for the tree to become consistently attributed.

A parse tree is attributed by repeatedly selecting a strongly connected component (potentially a single attribute) of its dependency graph and evaluating the attributes in that SCC until they reach final consistent values. SCCs can be selected in any way consistent with a topological ordering of the collapsed dependency graph, where each maximal SCC has been reduced to a single node. Thus, if there are no nontrivial SCCs in the graph, this evaluation process proceeds in the same way as the usual optimal evaluation process.

Before evaluation of the attributes mentioned by the grammar writer begins, certain additional links are added to the dependency graph by non-local productions. These additional links, indicated by dotted lines in Figure 3.3, make each attribute (outside a GSCC) that is a predecessor of a node in the GSCC also a predecessor of the start attribute of that GSCC. These links allow the evaluator to always begin evaluation of a GSCC at its gate attribute. The evaluation function for a start attribute returns **true**, indicating that the gate attribute should take its value from outside the GSCC to begin GSCC evaluation, iff any of its predecessors from these additional links were newly set or changed value since the gate attribute last took a value

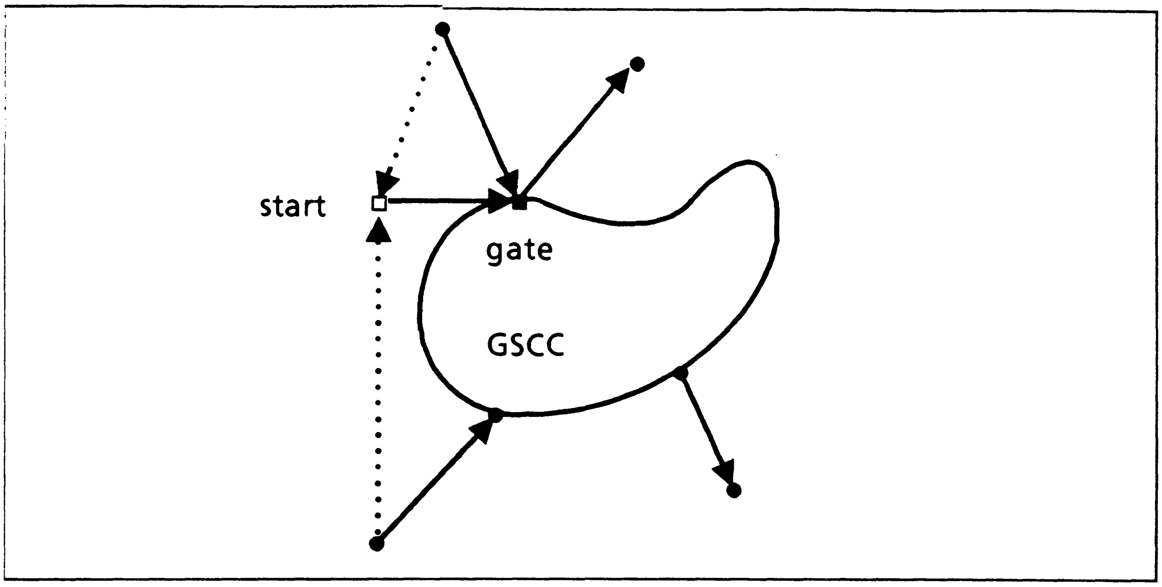


Figure 3.3. Adding Nonlocal Links.

from outside. Since the only successor of a start attribute is its gate attribute, adding these additional links does not change the SCC composition of the dependency graph.

These links are useful primarily for incremental reevaluation of a GSCC, but even during initial evaluation sub-GSCCs may require multiple evaluations, and any GSCC may require multiple passes to reach a fixed point. In practice, subsequent passes and evaluations of a GSCC would be handled by the more efficient incremental evaluation of the next section, but for the moment we consider a simplified algorithm where all the attributes of a GSCC are evaluated on each pass.

Since there may be nonmonotonic functions inside a GSCC, attribute evaluation order influences correctness as well as efficiency. An example where differing evaluation order results in different fixed points for the GSCC is shown in Figure 3.4. The attributes are shown with consistent values that could be left over from an earlier evaluation of the GSCC. Now assume that the GSCC must be evaluated again, and that the gate g gets the new initial value 20. If the GSCC is evaluated in the order acbgabcg, the

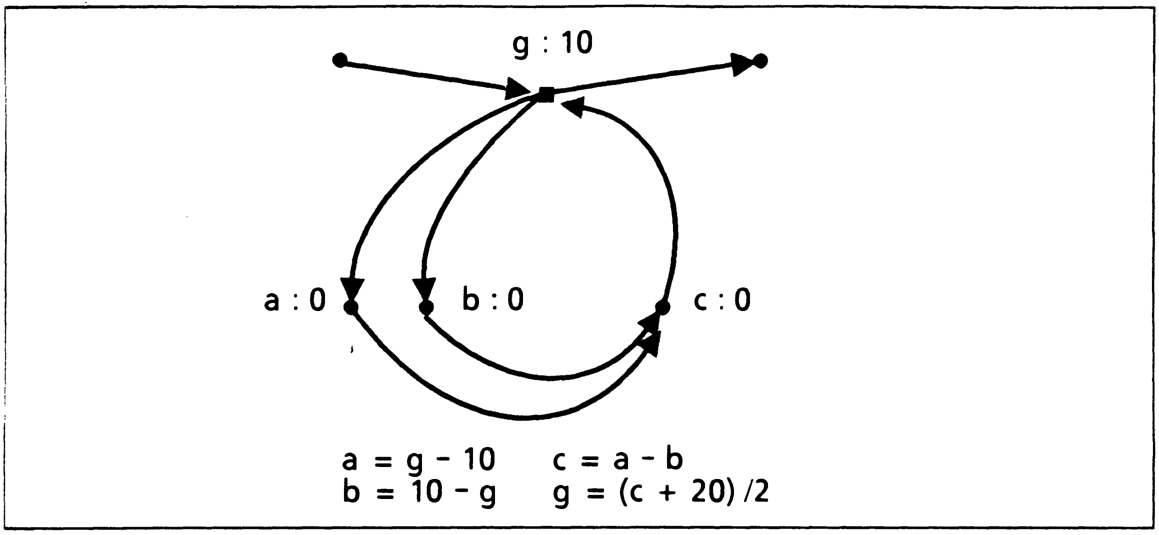


Figure 3.4. A Nonmonotonic GSCC.

fixed point (a:5, b:-5, c:10, g:15) is reached. If it is evaluated in the order abcg, the fixed point (a:10, b:-10, c:20, g:20) is reached.

This example also justifies the choice of ordering for the integers made at the beginning of Section 3.1. Using a flat cpo, we would find that the least fixed point of this GSCC is (a:$\perp$, b:$\perp$, c:$\perp$, g:$\perp$). While this is a solution to the set of simultaneous equations representing the GSCC, the writer of the attribute grammar would probably prefer to get the integer solution (a:10, b:-10, c:20, g:20). Similarly, if the cpo for states is constructed from the flat cpo for the integers in order to guarantee monotonicity of functions, the least fixed point of a while loop would have each attribute in the SCC at its bottom value, meaning that the final state of every while loop would have no identifiers defined.

The correct evaluation order within a GSCC is defined by a topological ordering on the GSCC graph with edges leading to the gate removed (and any sub-GSCCs collapsed to single nodes). In other words, an attribute in the interior of a GSCC cannot be evaluated unless all attributes on paths from the gate to it are consistent with the present gate value. There are cases, such as Figure 3.5, where such an ordering cannot be obtained due to a cycle interior to a GSCC, but such cycles are ruled out in gated attribute grammars by the requirement that each nontrivial cycle contain a gate attribute.

A nontrivial GSCC is evaluated by alternating between evaluating the gate and evaluating the other nodes of the GSCC in internal topological order, with possible recursive evaluations of sub-GSCCs, until one pass is made over the GSCC with no changes in attribute values. (The simplified algorithm of this section ignores the possibility of aborting the last pass over the GSCC once the gate has been evaluated without changing value. The incremental evaluation algorithm of the next section automatically incorporates this by using change propagation.)

Even with monotonic functions, Jones and Simon ([JoS86]) for efficiency restrict evaluation order to agree with a depth-first ordering starting at a defined set of SCC attributes. For the first pass over the SCC, the set consists of those SCC attributes whose predecessors outside the SCC have changed. On succeeding passes, the set consists of the successors of leaves of the depth-first spanning tree from the previous pass. Since all their functions are monotonic, the evaluation ordering from these spanning trees could, for comparison with the GSCC method, be replaced by one derived from a topological ordering on the SCC with incoming arcs to the defined set

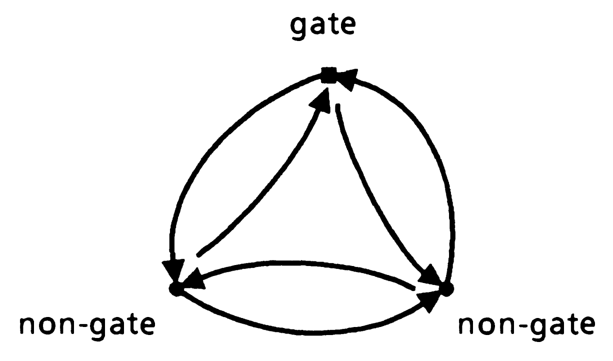


Figure 3.5. Inevaluable GSCC.

temporarily deleted. With such a replacement, the defined set for each pass would be a subset of that for the previous pass, with nodes being dropped from the set if their predecessors did not change on the previous pass. The GSCC algorithm starts with only the gate attribute in the defined set, and SCC evaluation terminates when the gate's predecessors in the SCC do not change on a pass.

3.3. Incremental Evaluation

At the beginning of incremental evaluation, a parse tree is consistently attributed except for a small number of attributes corresponding to places where changes have recently been made to the parse tree. As with noncircular attribute grammars, these attributes are just those around the root of the subtree if there has been a single subtree replacement. If more than one subtree has been replaced since the tree was last consistently attributed, we also need the attributes of the symbols along the branches that connect the replacement sites, as illustrated in Figure 3.6. These additional attributes provide the attribute evaluation sequencing between the replacement sites.

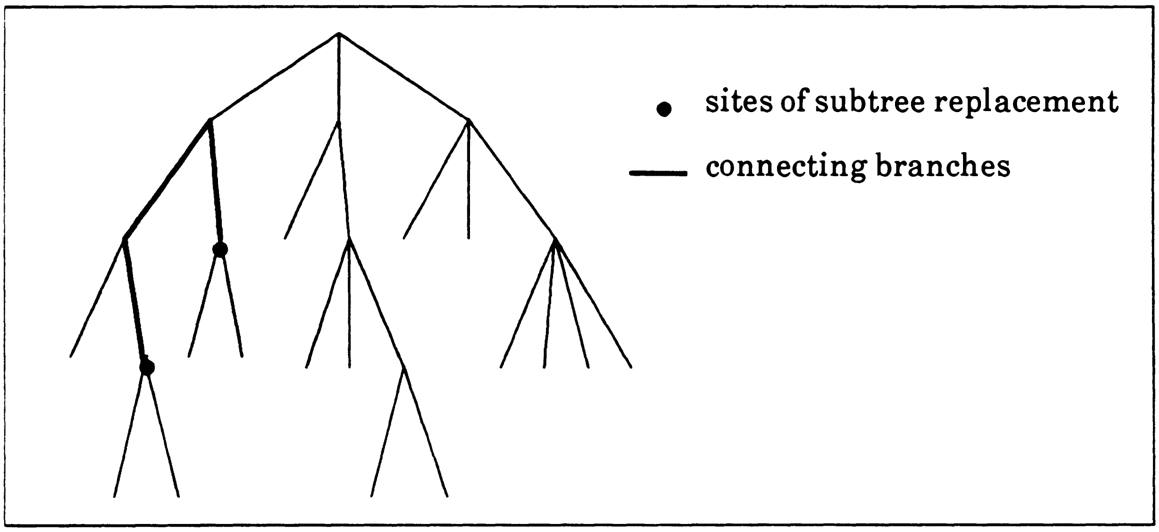


Figure 3.6. Connecting Sites of Subtree Replacement.

The strongly connected components containing the appropriate attributes are placed in the set NeedSCCEval, which throughout incremental evaluation contains those SCCs known to need evaluation. Another set, NeedEval(S), which contains those attributes of the GSCC S that are known to need evaluation, is kept for each GSCC (*i.e.* nontrivial SCC) in NeedSCCEval. While NeedSCCEval is not empty, an SCC is removed and evaluated. SCCs are selected for evaluation in accordance with the topological order imposed by the collapsed dependency graph.

A trivial SCC is evaluated by evaluating its node (repeatedly if the SCC is a trivial cycle) and adding its SCC-successors to NeedSCCEval if the value of the node changed. A nontrivial SCC is evaluated in the same spirit, as shown in Figure 3.7. Starting from the gate, changes are propagated around the GSCC in internal topological order until attributes settle to consistent values. Then SCC-successors are added to NeedSCCEval if their predecessor attributes have values different from before SCC evaluation.

Evaluating a sub-GSCC is identical to evaluating a GSCC, except that old attribute values are not saved and checked to add SCC-successors, since a sub-GSCC cannot know if its attribute values are final when they settle to consistent values. The sub-GSCC may be reevaluated, changing its attribute values further, in the process of evaluating the GSCC as a whole.

Whether evaluating a GSCC or a sub-GSCC, it suffices to evaluate attributes that are in the core and are known to need evaluation. If an attribute in a further nested GSCC needs evaluation, so does the start attribute of that nested GSCC, which is in the core of the next outer GSCC. Evaluating that start attribute will trigger evaluation of the nested GSCC, including the needy attribute.

```
for each node n in S with a successor outside S,
    save the present value of n
evaluate gate(S), setting start(S) false
if gate(S) changed, add successors of gate(S) in S to NeedEval(S)
while NeedEval(S) ∩ core(S) is not empty
    select n from NeedEval(S) ∩ core(S) in
        accordance with internal topological order
    if n is a start node,
        evaluate n
        if n is true, recursively evaluate the associated sub-GSCC, T
        add successors of T in S-T to NeedEval(S)
    else
        evaluate n
        if n changed, add successors of n in S to NeedEval(S)
for each node n in S with a successor outside S,
    if its present value differs from its saved value,
        add its SCC-successors to NeedSCCEval
        add its successors to NeedEval sets
```

Figure 3.7. Evaluating the Nontrivial SCC S.

Also note that the NeedEval set of the maximal GSCC can be used by the sub-GSCC evaluation, which will look only for attributes in the core of the sub-GSCC. Thus it is not necessary to worry about multiple NeedEval sets for different parts of the same maximal SCC.

This incremental evaluation algorithm follows the same principles that were developed for initial evaluation. The maximal GSCCs are selected in accordance with a topological ordering on the collapsed dependency graph; nodes within a GSCC are selected in accordance with the internal topological ordering. Since every other predecessor of a sub-GSCC is also a predecessor of the sub-GSCC's start node, the entire sub-GSCC (a "node" in the internal collapsed graph) can immediately follow its start node in the internal topological order. The incremental algorithm, however, adds the idea of change propagation in the hope that not every GSCC that descends from those determined by the subtree replacement sites will have its value affected by the updates. This change propagation allows the algorithm to reevaluate only those GSCCs that occur in the set of affected GSCCs, as was shown in [JoS86].

3.4. Avoiding Reevaluation of Loops

Often when an input to a loop changes, it is necessary to entirely reevaluate the loop. Where possible, however, we would like to avoid this expense by identifying those portions of a loop that do not affect the computation of the loop in such a way as to require an entire reevaluation. This basic idea is somewhat reminiscent of code hoisting in optimization.

This discussion is necessarily specific to the continuing example of an attribute-grammar-based interpreter in which attributes containing states are passed around the tree.

Just as in standard incremental evaluation, we hope that at some point short of reevaluating all the attributes in the tree either attributes stop changing value or they change value in such a way as not to require further reevaluation. To make this determination, it is useful to define *input* and *output variables* of a loop. An input variable is one whose value is queried somewhere inside the loop, and an output variable is one whose value is modified somewhere inside the loop. Variables may, of course, be both input and output variables for a given loop. For the example language, the input variables are those that appear on the right-hand side of an instance of the production exp ::= id and the output variables are those that appear as left-hand sides of assignment statements. The concept of the state value not-reached is extended to include an extra pair of integers in each state attribute. These integers will record the number of the loop iteration at which the state is first evaluated and the number of the loop iteration at which it is last evaluated.

During change propagation, say we find that a predecessor of a state-valued attribute in a GSCC corresponding to a loop has changed. We evaluate the state inside the GSCC and note which variables inside it change value. We consider each changed variable in turn. The first iteration in the loop during which the state was evaluated represents the first time that the variable being considered would actually receive the new value. If that is after the last use (if any) of the variable in the loop, then we do not need to reevaluate the loop; we need only update those states evaluated after the given one to reflect the new "final value" of the variable. (Variables that are independent of the loop are handled similarly.)

The above relatively inexpensive technique formalizes and captures such common situations as changing the "clean-up" code in a loop such as that in Figure 3.8, which intuitively should not require reevaluation of the entire loop. If the loop iterated 17 times, the first and last evaluation of states in the clean-up code occurred on iteration 17. Giving a different value to the variable clean in the clean-up code does not require reevaluating the loop, as no state in the loop will query the value of clean after iteration 17.

```
while not done do
    { main loop body }
    if condition then
        { clean-up code };
        done := true
    else
        { continue iterating }
```

Figure 3.8. Typical Loop.

3.5. Nonlocal Predecessors of Start Attributes

It is necessary to create links from predecessors of a GSCC to the start attribute of that GSCC. These links are used to make the start attribute **true** and schedule the GSCC for reevaluation whenever an input to the GSCC changes value in such a way as to require a full reevaluation of the GSCC. Superior and inferior characteristic graphs are used as necessary to find attributes that are in a GSCC and identify their predecessors that are not in the GSCC. The first step in creating the needed links is to associate a pointer to the appropriate gate attribute with each member of a GSCC. In the case of nested GSCCs, each member gets a pointer to the gate of the smallest containing GSCC. In this way, all exterior gates can be found by following the chain of gate pointers outward.

For the example grammar, gate pointers can be calculated by adding the evaluation functions in Figure 3.9 to the earlier evaluation functions from Figure 3.1. (The notation *@X.a* indicates that a pointer to attribute X.a is produced.) Statement nonterminals have initial state and final state attributes, and the associated gate pointer attributes are abbreviated isgp and fsgp respectively. A similar naming convention is used for gate pointers associated with state and value attributes of expression nonterminals and gate attributes for **while** symbols. Some of these gate pointers are calculated with respect to the inferior characteristic graphs contained in the infchar attributes of their associated grammar symbols.

Once we have the gate pointers, we can add the necessary links. An attribute is a predecessor of a GSCC if the gate pointer of a successor does not appear in the chain of gate pointers from the attribute (*i.e.*, the successor is not in a containing GSCC). Since the successor may be in several nested GSCCs, a link is added from the predecessor attribute to the start pseudo-

```
Start ::= S
    S.isgp = nil

S1 ::= while exp do S2
    while.categp = S1.isgp
    S1.fsgp = S1.isgp
    S2.isgp = @while.gate
    exp.sgp = if exp.infchar = <state⇒val> then
                    @while.gate
              else nil

S ::= id ← exp
    S.fsgp = S.isgp
    exp.sgp = if exp.infchar = <state⇒val> then
                    S.isgp
              else nil

S1 ::= S2 ; S3
    S1.fsgp = S1.isgp
    S2.isgp = S1.isgp
    S3.isgp = S1.isgp

exp ::= id
    exp.valgp = exp.sgp

exp ::= int
    exp.valgp = exp.sgp

exp1 ::= exp2 op exp3
    exp2.sgp = if exp2.infchar = <state⇒val> then
                    exp1.sgp
               else nil
    exp3.sgp = if exp3.infchar = <state⇒val> then
                    exp1.sgp
               else nil
    exp1.valgp = exp1.sgp
```

Figure 3.9. Added Flow Rules.

attribute associated with each gate pointer in the trace-back chain from the successor until this trace-back chain joins the one from the predecessor. A diagram of these trace-back chains and resulting links for nested GSCCs is found in Figure 3.10. Figure 3.11 shows a parse tree with the dotted links added after execution of this first phase of attribute evaluation. (The GSCC in this figure has been circled so that its predecessors are easier to identify.)

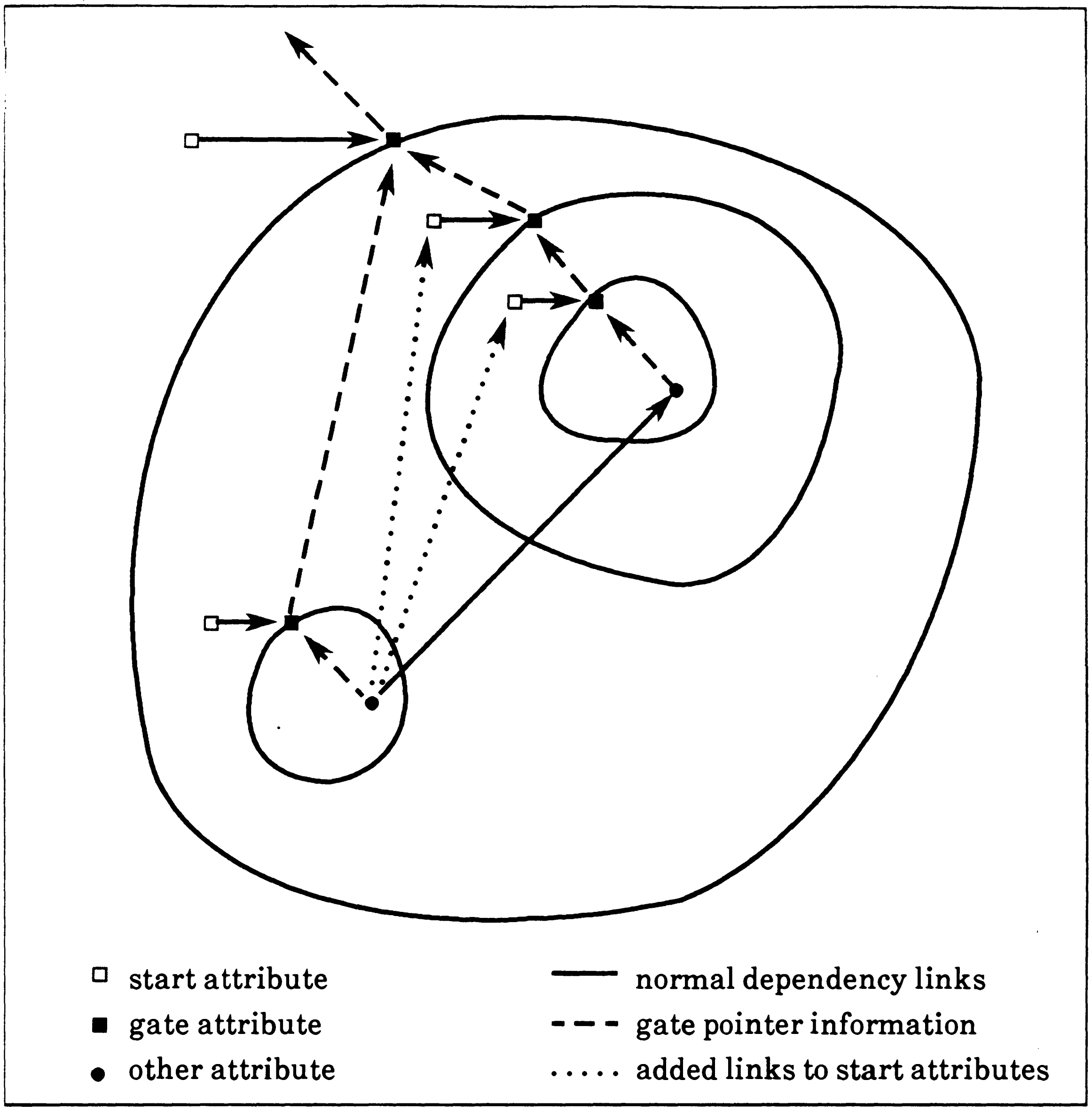


Figure 3.10. Using Gate Pointers to Add Nonlocal Links.

3.6. Discussion

As in [JoS86], these gated attribute grammar techniques guarantee that each maximal SCC evaluated will be in AFFECTEDSCC, and that no SCC will be evaluated more than once in a single incremental evaluation. However, due to the lack of restriction on the functions within an SCC, there cannot be a bound on the number of attribute evaluations required to evaluate a single SCC. To gain the power to model execution of a while loop

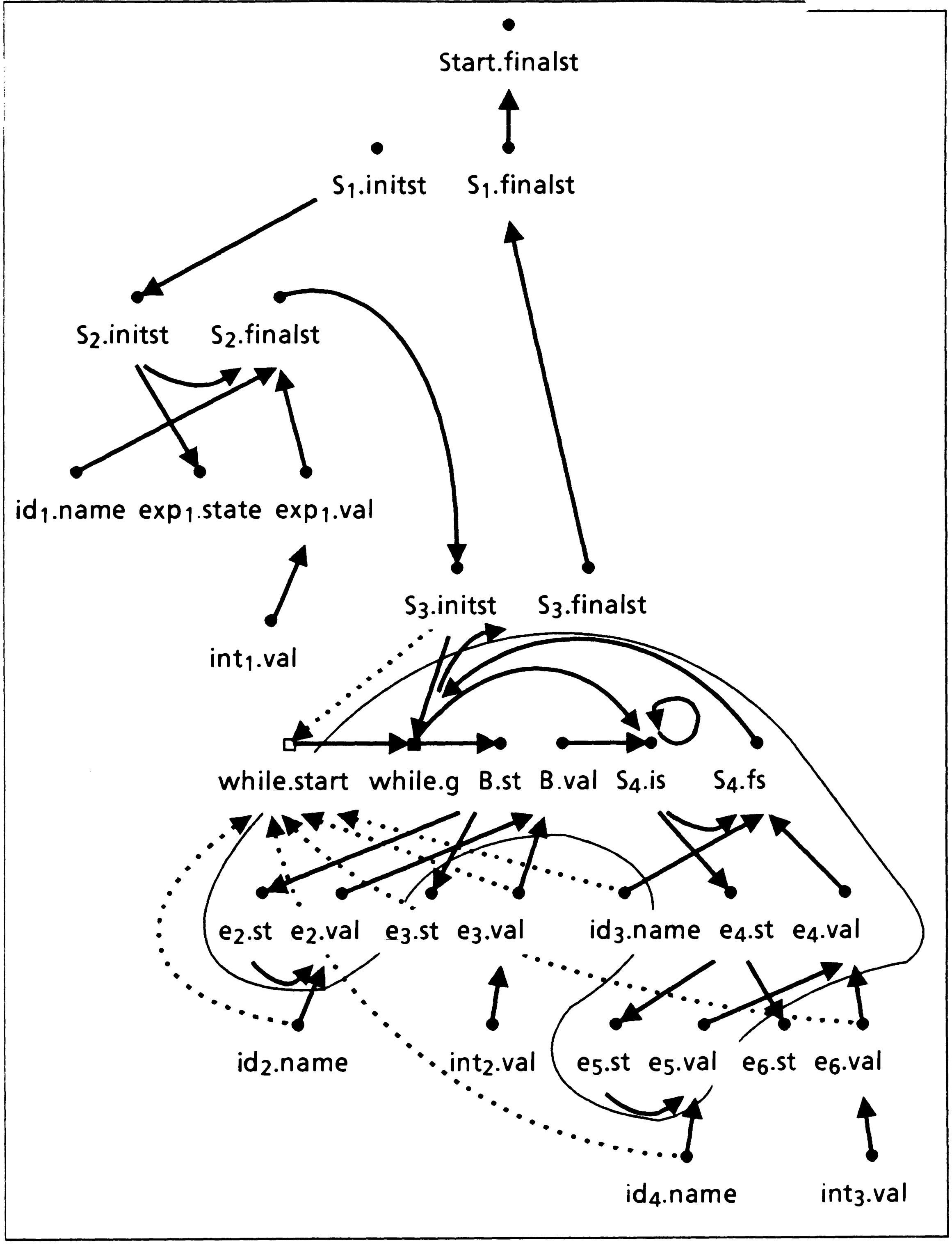


Figure 3.11. Dependency Graph for i ← 10 ; while i > 0 **do** i←i − 1.

in attributes, we are forced to accept that evaluation of an SCC may not terminate if its corresponding while loop diverges. (Evaluation of an SCC representing a divergent **while** loop will in fact terminate if the states in the **while** loop stop changing, as in the loop **while** 1 **do** i ← 2. In such a case, any subsequent states in the program are given the not-reached value.)

The recursive synth-function evaluator described in [Far86] addresses evaluation-process-termination problems in recursive attribute grammars by pulling the evaluation of an SCC into a single function and then pointing out that only a finite number of function evaluations will be attempted. For Farrow's finitely recursive attribute grammars, this transformation does not generate partial evaluation functions unless some of the SCC functions were already partial. (Most functions in traditional attribute grammars are simple total functions, but there is no way in general for an evaluator generator to guarantee these functions will always terminate.) However, for general circular attribute grammars this transformation would not affect the actual termination of the evaluation process, but would merely move potential nontermination from the attribute selection phase to the attribute evaluation phase.

Thus, partial functions are now allowed to be explicitly spread over the computation of the attributes of a GSCC instead of restricted to the computation of a single attribute. This choice allows computation to be expressed via attribute evaluation functions in whichever way seems most convenient.

By forcing attribute evaluation to occur in accordance with a topological ordering of the dependency graph, attribute evaluation is guaranteed to terminate if interpretation of the corresponding program would terminate. In addition to allowing a GSCC to converge to the wrong fixed point, evaluation out of topological order can result in undesired nontermination. If a GSCC were to be evaluated when some of its predecessors had yet to reach their final values, the invalid combination of inputs might be such that the

GSCC has no reachable fixed point. Yet GSCC evaluation cannot stop before it reaches a fixed point, for on valid input the evaluation must continue until it determines that the corresponding loop converges. However, topological ordering for the sake of efficiency will also guarantee that all GSCC inputs are valid when the GSCC is evaluated.

To gain the ability to evaluate arbitrary functions in dependency graph cycles, one localized timing constraint on attribute evaluation has been accepted. A start attribute is required to know if any of its predecessors have changed *since the last time its gate attribute took a value from outside the GSCC*. This constraint is met by setting the start attribute false immediately after, but conceptually at the same time as, taking a gate value from outside the GSCC. Without some way of telling whether GSCC evaluation is just beginning, a gate cannot know which of its evaluation functions to use. Using the internal evaluation function at the beginning of GSCC evaluation may abruptly end that evaluation if the gate is the only GSCC attribute with modified predecessors; using the external function in the midst of GSCC evaluation again brings up the problem of possible divergence due to invalid combinations of values. Jones and Simon face a somewhat similar problem in incrementally evaluating SCCs with monotonic functions. They must keep track of whether an attribute has been evaluated in the current round of SCC traversal to ensure that a left-over value does not cause SCC evaluation to miss its new least fixed point.

With the use of an efficient representation, such as may be found in [Hoo87], for aggregate attributes like states that differ only slightly from one another, the performance penalty for propagating state information around a dependency graph may be substantially reduced from that of a naive implementation. Any remaining penalty should be more than offset by the advantage of being able to automatically start reexecution at the point of change in a program.

Chapter Four

Unification-Based Type Inference

In many modern programming languages, a program is considered well-typed if and only if types can be consistently assigned to all expressions in the program. If every expression, including such intermediate-level expressions as i+j−3, is explicitly typed by the programmer, the compiler or interpreter need only check that the type conversion rules of the language are followed to verify the typing of the program. If some expressions are untyped, the compiler or interpreter must infer the types of those expressions from context and the types of component expressions. In languages such as Pascal and C, the programmer associates a specified type with each identifier and the compiler then uses these types to infer the types of intermediate-level expressions. The compiler produces a type error if it cannot consistently type such an expression. Under other typing schemes, however, a compiler will associate a type with every expression of a type-correct program without the aid of identifier declarations. Even in these cases, declarations remain useful both as a reference for the human programmer and as a statement to the compiler of the programmer's intended use of an identifier.

Whatever the typing scheme, type inconsistencies may occur between the declaration and uses, or simply between different uses, of an identifier. The usual way of dealing with such inconsistencies is to assume that the declaration, if any, gives the proper typing of the identifier. If there is no declaration, the first use in the program determines the type. Unfortunately, while simple to implement, this approach can be misleading as to the real source of the type errors. Often, an identifier may be used many times in a program in a way consistent with the programmer's idea of its type. The identifier may have been misdeclared as a different type, the programmer may misremember its type, or the declaration may be left over from an earlier sketch or version of the program. In any case, if the usage is consistent, the minimal change to bring the program to type consistency involves changing the declaration. The common simple approach to type inconsistencies would flag each "incorrect" use as a type error, but not flag the declaration or any uses consistent with the declaration. A better way to determine the "correctness" of each occurrence of an inconsistently typed identifier would be to figure out how many times the identifier was indicated to be of each inconsistent type and then to flag each occurrence with an intensity depending on the relative popularity of the type it supported.

As an example of the difference in philosophy, consider a program in a language without declarations, where each occurrence of the variable x is in a context permitting only a boolean or an integer value. If there are a number of inconsistent uses, scattered through the program, appearing in the order

use x as boolean;		*
use x as integer;	***	**
use x as boolean;		*
use x as boolean;		*
use x as boolean;		*
use x as integer;	***	**
use x as boolean;		*
use x as boolean;		*
use x as boolean;		*

the common approach of believing the first use would flag only the integer uses (and perhaps also the first boolean use as an indication of the "correct" type). These flags are shown in the second column. A voting approach capable of distinguishing majority and minority opinions, however, would flag all the uses with the intensities shown in the third column. In a similar program where the first two uses were interchanged to give the order

use x as integer;		**
use x as boolean;	***	*
use x as boolean;	***	*
use x as boolean;	***	*
use x as boolean;	***	*
use x as integer;		**
use x as boolean;	***	*
use x as boolean;	***	*
use x as boolean;	***	*

the voting approach still flags each use with the same severity as before, but the flags from the more common approach have changed drastically. If the program is at all large, with the uses of x scattered throughout it, the programmer is unlikely to be immediately aware of the context of all the uses if they are not flagged. However, the knowledge of the overall pattern of uses is exactly the information needed to decide whether some contexts are in error and should be changed (when all the uses of x are conceptually the same variable) or whether all the contexts are "right" but some uses of x refer to what is conceptually a different variable, in which case those uses should be replaced by uses of a new variable.

In real languages and programs, of course, there are a number of complications that do not appear in this simple example. Context may not be able to constrain x to be an integer but only to be the same type as y. Similarly, a function f may be known only to take three arguments of as-yet unknown types. The presence of such complications requires care in defining the number of times a type constraint is asserted by a program.

4.1. A Subset of ML

A type theory capable of counting the support for type options has been developed based on the polymorphic type theory exemplified in the language ML ([Mil78], [Car83]). ML is a statically scoped functional language resembling a typed lambda calculus that provides type operators for constructing functions, cartesian products, and lists. Having several operators enriches the available types, but the product and list operators add no essential complexity beyond that of function abstraction to the type theory. Therefore, explanations of type systems, including those in this thesis, often use only function abstraction.

A monomorphic language like Pascal needs to declare separate functions with types integer list → integer and boolean list → integer to be able to find the lengths of both integer lists and boolean lists. In ML, a single function length can be applied to a list of elements of any given type. Thus, length has the polymorphic type α list → integer, where α is a type variable that can unify with any type expression, including integer or boolean. In general, a type expression in ML will consist either of a type variable or of a type operator applied to other type expressions. Type constants like integer and boolean are treated as the operators integer() and boolean(), which take no arguments. (Type expressions may be considered to be implicitly quantified at the outermost level, so that length would actually have type $\forall \alpha.\alpha$ list → integer.)

ML's type theory, while much stronger than that of monomorphic languages, is still decidable at compile time. That is, during compilation, ML can infer the most general type for each expression in a program and check that all syntactic combinations of expressions are type correct. For any expression that has been explicitly typed by the programmer, ML also checks the inferred type against the specified type. Any program that passes type-checking during compilation is then guaranteed not to generate type errors at run-time.

The subset of ML that provides the basis for the language accepted by the MOE editor ([JoW86]) was described in [Car85] and is reproduced in Figure 4.1. Most of the syntax of this subset is a straightforward representation of function abstraction and application, conditional expressions, and the usual glue common to programming language grammars. This subset has the polymorphism of the full language, but it does not have type overloading, which would allow each occurrence of an identifier to have any of an enumerated set of types. For example, in many programming languages the operator + is overloaded so that it can take either integer or real operands, but not types such as boolean or string. The semantic interest of this subset language lies in the Term ::= **let** Declaration **in** Term and

Term ::=
    Identifier |
    if Term **then** Term **else** Term |
    fun Identifier . Term |
    Term Term |
    let Declaration **in** Term |
    (Term **)**
Declaration ::=
    Identifier = Term |
    Declaration **and** Declaration |
    rec Declaration |
    (Declaration **)**

Figure 4.1. An ML Subset.

Declaration ::= **rec** Declaration productions. The **rec** keyword changes the type environment scoping slightly to allow an identifier to be bound to a term with a recursive type. The special semantics of the **let** clause allow an identifier to be used polymorphically.

In some sense both

(**fun** f. pair (f 3) (f **true**)) (**fun** x.x)

and

let f = **fun** x.x **in** pair (f 3) (f **true**)

want to create an ordered pair from the results of applying the identity function to 3 and **true** and thus will β-reduce to

pair ((**fun** x.x) (3)) ((**fun** x.x) (**true**))

and then to

pair (3) (**true**)

when treated as pure λ-terms. However, in the first case, f, as a parameter, is constrained to be used as a single type over all of its occurrences. Therefore, although an identity function can be sensibly applied to both integers and booleans, there is an error on the type of f since it must take an argument of a type which is both integer and boolean. In the second case, f is bound to the polymorphic type $\alpha \rightarrow \alpha$ by the declaration, and each occurrence of f within the body of the **let** clause is only constrained to be consistent with the declaration (*i.e.*, two occurrences in the body do not have to be consistent with each other). This is done by defining type variables introduced by **let** declarations to be *generic*, as in [Mil78]. Each occurrence of f within the **let** body starts out with a type in which each generic type variable has been replaced by a new type variable unique to that occurrence. In this case, the two occurrences of f might start with types $\beta \rightarrow \beta$ and $\gamma \rightarrow \gamma$, which would then become boolean→boolean and integer→integer on the way to generating a correct typing for the entire term.

4.2. Generating Type Equations

In the usual setting for automatic type inference, parsing a syntactic fragment provides a number of type constraints that are expressed as a set of type equations. The standard unification algorithm, discussed in the next section, then proceeds to further constrain variables found in these equations by adding those constraints that can be generated by substituting for variables and by equating arguments of functions according to a set of standard rules.

In the ML subset, an identifier is associated with a different type variable each time it is redeclared in the new scope provided by a **fun** or a **let** clause. A parser would maintain a type environment as discussed in the previous section to allow it to retrieve the type variable associated with each identifier occurrence. The productions that directly impose type constraints are shown in Figure 4.2, along with their constraints. As a simple example,

```
Term ::=
    if Term1 then Term2 else Term3 |
        type(Term1) = boolean()
        type(Term2) = type(Term3)
    Term1 Term2
        type(Term1) = α→β    (where α and β are new type variables)
        type(Term2) = α
Declaration ::=
    Identifier = Term
        type(Identifier) = type(Term)
```

Figure 4.2. Type Constraints in an ML Subset.

the term **fun** x. **if** x **then** x **else** 3 generates the conflicting type constraints type(x) = boolean() and type(x) = integer().

4.3. Standard Unification

Several current systems, such as the ML interpreter, extract type equations imposed by a program. These systems use variants of the standard unification algorithm first proposed by Robinson ([Rob65]) to process their equations and determine the full constraints on the type variables. Conceptually, unification solves the system of constraints found in the input equations and yields a concise listing of the full constraints with one equation per constrained variable. One of the many equivalent algorithms to perform standard unification, found in [MaM82], can be stated in the following manner:

> Given a set of type equations, apply the following rules until no rule is applicable. The remaining equations give the final constraints on type variables. (By notational convention, x is a type variable, f is a type constructor, t is a functional term, and u is either a type variable or a functional term.)

(1) Given an equation $x = x$, delete the equation from the set.

(2) Given $t = x$, replace the equation by $x = t$.

(3) Given $f(u_1,...,u_n) = f'(u_1',...,u_m')$, where $f = f'$ and $n = m$, replace the equation by the n equations $u_i = u_i'$.

(4) Given $x = u$, where x does not occur in u but x does occur in some other equation, replace all occurrences of x in other equations by u.

This standard algorithm has several desirable properties. It is guaranteed to find a conflict if one exists and to find the least constrained ("most general") type for each type variable if there are no conflicts. It is guaranteed, in the absence of conflicts, to produce the same final constraint (up to renaming of unconstrained variables) for each variable regardless of the order of application of the rewrite rules. Also, it can be implemented fairly efficiently. Most unification algorithms have an exponential worst-case behavior, since the size of a term can grow exponentially. The classic example of this exponential growth is the set $\{x=f(x_1,x_1), x_1=f(x_2,x_2), ..., x_{n-1}=f(x_n,x_n)\}$, where the solution for x contains $O(2^n)$ f's and x_n's. The algorithm discussed in [PaW78] and [dCh86], however, runs in linear time by using pointers to subterms and thus never creating multiple copies of one subterm; the final answer remains implicit in the data structures.

4.4. Recursive Unification

A drawback of the standard unification definition for some applications is its use of the "occurs check" in rule (4). This check exists because, in many applications, the mere appearance of a recursive type signals an error. In addition, the check helps to guarantee termination by preventing an infinite number of substitutions for the same variable. (With nonrecursive types, no occurrences of the variable x would remain in other equations once the equation $x=t$ was used for substitution, so that equation could not be used for substitution again.) However, newer functional languages allow self-application of functions and other means of producing objects that are only correctly described by recursive types ([MPS84]). Among the terms which cannot be correctly typed in a nonrecursive system is any syntactic representation of the paradoxical combinator, Y, which finds the least fixed point of functionals. Languages restricted to nonrecursive types often hardwire some special fixpoint operator into the language (*e.g.* the functional and applicative languages in [Bro86]), but it seems more intuitively pleasing to allow languages the ability to define such operators on their own. Since we would like to be able to use our ideas in environments for these stronger languages, we must use an extended definition of unification that admits the legality of such types.

Many people ([Col82], [CKv83], [MPS84], [MaR84]) have extended the definition of unification to allow recursive types. This relaxation is done by removing the occurs check and explicitly or implicitly creating representations for infinite types as they are found. Colmerauer uses the concept of representing terms as rational trees (sometimes known as regular trees by analogy with regular expressions), which are trees with a finite number of different subtrees. All trees representing finite types are therefore rational, as are some, but not all, infinite trees. Figure 4.3 shows several representations of an infinite tree with only two distinct subtrees.

The primary formulation used in this thesis is a modification of that presented in [JoW86]. That set of rewrite rules, reproduced below to introduce the concepts, involves removal of the occurs check in rule (4) of the standard set and modification of the other rules. The notation $f_i(@i)$ is used to represent the infinite regular tree f(f(f(...))). The occurrence of the label "@i"

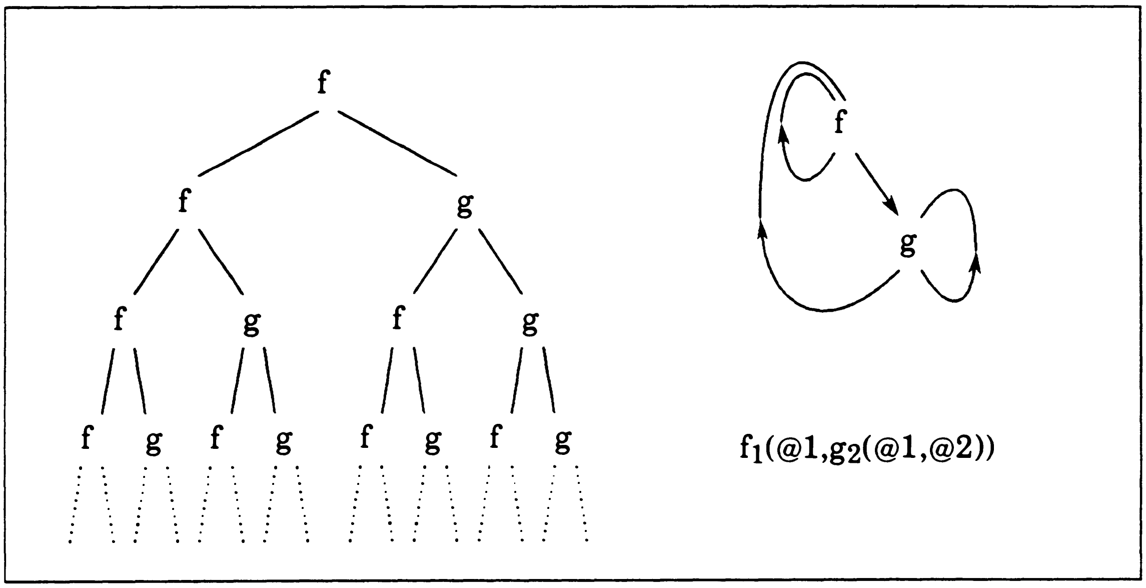


Figure 4.3. Regular or Rational Trees.

inside an expression stands for a copy of the expression whose root symbol is subscripted by the label i.

This set of rewrite rules is nonstandard in that it does not remove or replace equations. A given rewrite rule that would normally involve replacement of an equation by other equations simply causes the other equations to be added. To insure that a given equation is not used more than once by the same rule, a clause is added to the guard of each rewriting rule, requiring that an application of a rule to a candidate equation result in equations that are not already present. The approach makes the termination proof of the algorithm easy, and anticipates further modifications to the standard set of rewrite rules that will appear later.

After the algorithm has terminated, every nonvacuous variable will appear exactly once as the left-hand side of some equation, so constraints on variables are easy to identify even in the presence of extraneous equations that the standard unification procedure would have eliminated.

The rules are given below.

(1) If $t=x$ is present in the set of equations and $x=t$ is not present, add $x=t$.

(2) If in $f(u_1,\ldots,u_n)=f'(u_1',\ldots,u_n')$ some $u_i=u_i'$ is not already in the set of equations, add from $\{u_1=u_1',\ldots,u_n=u_n'\}$ those $u_i=u_i'$ that are not already in the set of equations. If f (or f') is subscripted, replace any occurrences of the subscript in any u_i (or u_i') by a copy of the entire expression.

(3) If there is an equation $x=u$, u is not x, and x appears in some other equation:

i) If x occurs in u then, since u is not x, u must be a function. Subscript the root symbol of u with a new label and replace occurrences of x in u by that label.

ii) Replace occurrences of x in other equations by the possibly modified u.

Using this new term for replacement avoids the potential termination problem, as there will no longer be an occurrence of x in any other equation after $x = f_i(@i)$ is used for substitution. The equate-arguments rule requires that a pair of regular trees be tested for equality to other pairs of regular trees. An algorithm to perform this test can be found in [Knu73].

Cardelli presents a sample ML implementation of type inference using environments ([Car85]). That algorithm can be modified to implement recursive unification by adding mark bits for the traversal of the now-recursive data structures. The other error-tolerant unification schemes found in the following chapters will also permit recursive sets of unification equations.

Chapter Five

Error-Tolerant Type Inference

For sets of type equations without conflicts, standard unification and its extension to recursive unification produce well-defined answers. However, standard unification is not nearly so well-behaved when presented with a conflicting set of type equations.

One form of poor behavior is reflected in the lack of relationship between the set of input equations and the resulting error indications. While a conflict will be found if one exists, there may be no way to tell which type variables caused the conflict. As an example of this, consider the two sets of type equations $\{x=f(z), x=g(z), y=f(z)\}$ and $\{x=f(z), y=g(z), y=f(z)\}$. By using the first equation with the substitution rule from the standard unification set of the previous chapter, the first set can be rewritten to give $\{x=f(z), f(z)=g(z), y=f(z)\}$. By using the third equation with that rule, the second set also gives $\{x=f(z), f(z)=g(z), y=f(z)\}$. Although these rewritten sets are identical, the "blame" for the conflict in the first case should be given to x while in the second case it should be given to y. The standard definition of unification does not retain the information we need to pinpoint the cause of the conflict.

Another facet of this drawback, also due to the inability to remember earlier versions of an equation, is that the constraints produced for variables from a conflicting set of equations will depend on the order in which rules were applied. Consider the set $\{x=f(z), x=g(z)\}$. We must choose one of the two equations to use with substitution. These two choices give us the final sets $\{x=f(z), f(z)=g(z)\}$ and $\{g(z)=f(z), x=g(z)\}$. Both results tell us that the original set had conflicts, but the results have different constraints on x. Since we would like to extract useful constraints from a unification algorithm even when there are conflicts in the input set, neither of these choices can be considered a satisfactory final answer. As the original set asserts x to be both f(z) and g(z), the final constraint on x should indicate this by some means such as $x = f(z) \mid g(z)$, which shows the conflicting alternative options f(z) and g(z).

A second drawback of the standard definition is its failure to preserve any information regarding strengths of assertions. Given $\{ x = f(y),\ x = f(y),\ x = f(y) \}$, it will generate a sequence of sets similar to

$\{ x = f(y), f(y) = f(y), f(y) = f(y) \}$

$\{ x = f(y), y = y, f(y) = f(y) \}$

$\{ x = f(y), f(y) = f(y) \}$

$\{ x = f(y), y = y \}$

$\{ x = f(y) \}$.

The initial set asserted $x = f(y)$ three times, yet this information has been lost. For a set of equations without conflicts, the only kind for which the standard definition is expected to provide constraints, such information is not important. However, the ability to retain assertion strength information would make it possible to use the relative strengths to indicate the most probable error sites in the program. (The most probable error sites would be those supporting the weakest of the conflicting assertions.)

The set of equations $\{ x = f(y),\ x = f(y),\ x = f(y),\ x = g(z) \}$ should imply that x is predominantly the same type as f(y), while $\{ x = f(y),\ x = g(z),\ x = g(z),\ x = g(z) \}$ should imply that x is predominantly the same type as g(z). By this standard, an unweighted constraint of the form $f(y) \mid g(z)$ for x would be considered an incomplete solution.

5.1. Desired Properties

When the type variable x is involved in conflicts as in the previous section, it is useful to write its final constraint in the form f(y)[m] | g(z)[n]. This notation, where m and n are the numbers of times that x has been indicated to be the same type as f(y) and g(z), respectively, retains much more information about the implications of the original equations and allows easy determination of the predominant constraint on x. In a constraint of this form, f(y)[m] and g(z)[n] are called weighted options while the type of x is the disjunction of these two options.

In general, every part of a term that is a function or a recursive variable (*i.e.*, not an ordinary variable) will get a weight. Ordinary variables do not get weights because they do not provide any further information about the term; recursive variables get weights because they stand in the place of functional subterms. This weight is the number of independent votes for a term of that particular functional form, or when the term is thought of as a tree, the number of votes for that branch of the tree. The example in Figure 5.1 displays the weighting concepts graphically. A * serves as a placeholder for a subtree or subterm whose identity is unimportant for the moment; it can be thought of as a pattern matching any set of subtrees. As this example suggests, the weight of a form such as f(*,h(f(*,*))) cannot exceed the weight of higher forms such as f(*,h(*)).

A *branch* is defined as a path from the root of the tree, together with the portions of the overall tree structure visible from that path. Being able to see the local tree structure allows branches to know how many arguments each of their functions take, which in turn allows functions with the same name but different numbers of arguments (whether by design or by programmer error) to coexist peacefully. Treating such duplicated function

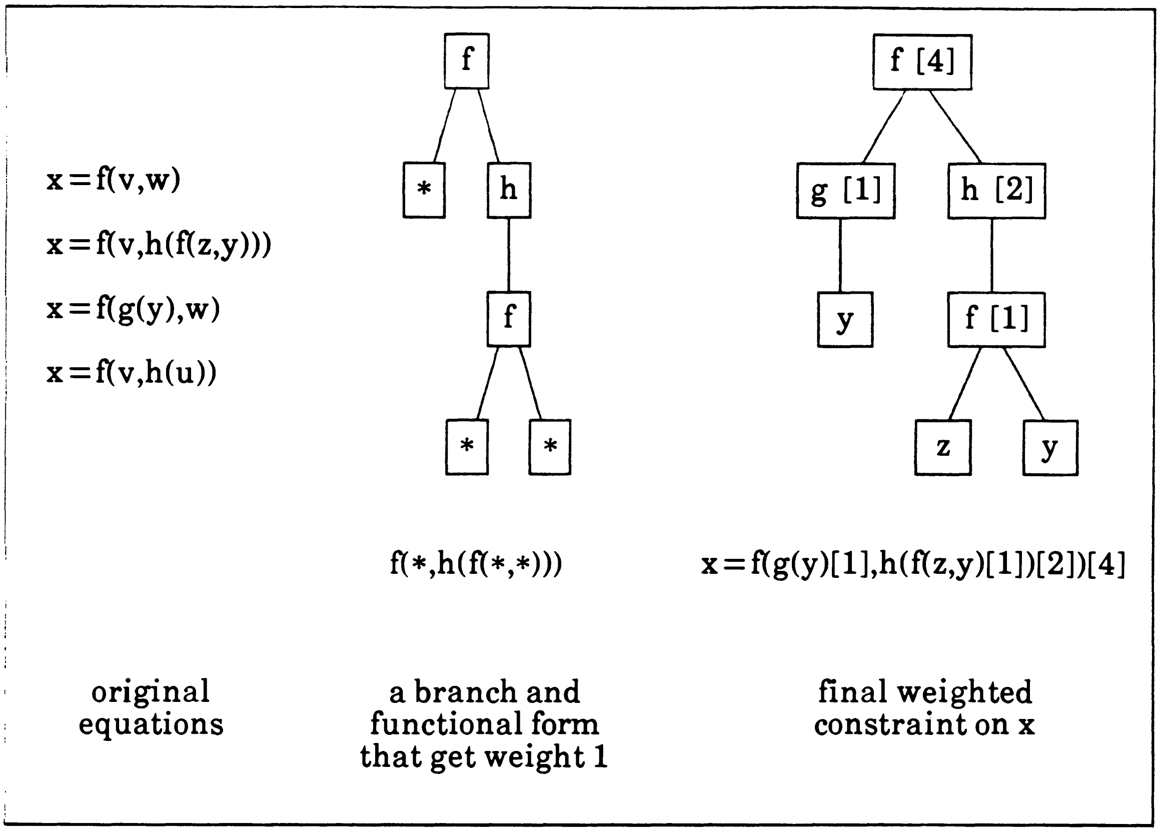


Figure 5.1. Weighting by Branches.

names as the same function is quite possible, but it does not seem particularly useful. Treating them as different, as would be suggested by the standard unification rules and by the standard definition of functions with fixed numbers of arguments in predicate calculus (*e.g.* [End72]), generates conflicts such as f(x) | f(y,x); treating the duplicated function names as the same, as might be suggested by the denotational semantics practice of treating functions in multiple stages of Currying as "identical" due to their isomorphism, would in that case equate x and y. This could be a valid constraint if a function is being used as $\alpha \rightarrow (\beta \rightarrow \gamma)$ with either one argument of type α or two arguments of types α and β being supplied, but the function could as easily be being used as both $(\alpha \times \beta) \rightarrow \gamma$ and $\alpha \rightarrow (\beta \rightarrow \gamma)$. In this latter case, equating the first argument positions equates $\alpha \times \beta$ with α, thus implying a recursive type where none should exist. This uncertainty of interpretation, along with the idea that the programmer is expected to make mistakes such as leaving out or inserting extra arguments, leads to the idea that a type function should accept only a fixed number of arguments, with any variation in that number indicating an error.

The first property that any new style of constraints must satisfy is compatibility with the standard unification algorithm in cases without conflicts. Since $\{x=f(y),\ x=f(w),\ w=g(z)\}$ is a consistent set of equations, any algorithm tolerating type conflicts must give the same final constraint for x (modulo weighting) that the standard algorithm does, namely $f(g(z))$. Thus the modified algorithm must merge consistent constraints to avoid returning a constraint of $f(y)[1] \mid f(g(z)[1])[1]$. As x is twice indicated to be an application of the function f, the most reasonable new-form constraint would be $f(g(z)[1])[2]$.

However, the original equations may contain multiple constraints which are pairwise consistent but not totally consistent, as in the case of $\{x=f(y,y),\ x=f(z,int()),\ x=f(bool(),w)\}$. In such a situation, whichever two constraints are merged first can dominate the third constraint in the final result, if consistent constraints are merged as they are found. In other words, although consistent alternatives must be merged to obtain the same answer as the standard algorithm in the absence of conflicts, merging alternatives can make the outcome dependent upon the order in which rewrite rules are applied in the presence of conflicts.

Therefore, to satisfy the second desired property, that a given set of equations should have a single solution independent of the order of original equations or rule applications, the generation of implied constraints must be separated from the merging of those constraints.

5.2. Modifying a Standard Algorithm to Tolerate Conflicts

A number of alternate definitions of unification have been examined to see which ones can overcome the drawbacks in standard unification. The first attempt, implemented in Pascal under 4.2 BSD UNIX†, involves a simple modification to the unification algorithm presented by Cardelli in [Car85]. Cardelli's algorithm infers potentially polymorphic types for the identifiers of a program written in the ML subset of the previous chapter as the program was typed in by a programmer. The algorithm was expanded to deal with recursive and conflicting types, and to provide approximate weights for various conflicting options. This is done by merging constraints on the same variable as they are found and incrementing the weights for consistent portions of the constraint. As an example of this, consider the set { x = f(y,w), x = f(g(z),w), x = g(y), x = f(y,h(z)) }. If the constraints are encountered in order from left to right, the first step would be to merge f(y,w)[1] and f(g(z)[1],w)[1] to get f(g(z)[1],w)[2]. This constraint would then be merged with g(y)[1] to give f(g(z)[1],w)[2] | g(y)[1] and then finally merged with f(y,h(z)[1])[1] to yield f(g(z)[1],h(z)[1])[3] | g(y)[1].

The costs of allowing for and dealing with recursive types are the usual ones associated with any recursive data structure, namely more complex traversal and replication. The cost of allowing for conflicting types is an extra level of indirection in referencing functional types. The cost of actually dealing with conflicting types when they do occur is O(mn) where m and n are the numbers of conflicting types for two variables involved in a unification. As both m and n are expected to be very small, indeed normally 1, the efficiency of the implementation does not suffer greatly from allowing far

† As anyone who's read this far doubtless knows, UNIX is a registered trademark of AT&T Bell Laboratories.

more type information to be retained. Thus we see that a conflict-tolerant unification algorithm meeting the first criterion can be implemented with acceptable efficiency. Unfortunately, this algorithm does not meet the second criterion, that of providing unique answers, in cases with pairwise but not total consistency. Using the earlier example of { x=f(y,y), x=f(z,int()), x=f(bool(),w) }, merging the first two constraints first would give f(int()[1], int()[1])[2] and f(bool()[1],w)[1] which then form f(int()[1] | bool()[1], int()[1])[3]. Merging the first and third constraints together first would give f(bool()[1], bool()[1])[2] and f(z,int()[1])[1] which then form f(bool()[1], bool()[1] | int()[1])[3], which shows the conflict in the other argument position.

Discrepancies of that sort seem to be minor and uncommon, as the implementation has given reasonable results, including variable option weightings, for a variety of sample programs. Predefined functions, such as plus and equals, can be handled by simply placing their initial definitions in the initial environment. As these functions are not reserved words of the language but merely predefined identifiers, the definitions can be changed either by introducing a new scope or by causing type conflicts which introduce new options for the identifiers.

5.3. A Maximum-Flow Algorithm for Unique Answers

The second attempt, a graph-based maximum flow algorithm ([JoW86]), satisfied both criteria by separating the derivation of implied constraints from the production of constraints in their final form. To derive all constraints implied by the original set of equations, there must be a means to coordinate pairwise-consistent constraints without explicitly merging them. This coordination is supplied by applying a flow detection algorithm to a graphical representation of constraints. Each node in the graph represents a term or subterm in the original set of equations; each (undirected) edge in the initial graph has a weight which is the number of original equations whose left and right sides correspond to the endpoints of the edge. The amount of flow between two functional terms in this graph indicates the number of times these terms could be equated by using transitive rewrite rules, such as replacing equations $t_1 = t_2$ and $t_1 = t_3$ by $t_2 = t_3$, on the original equations. The next three subsections provide a full description of the specific algorithm which processes this graph. The last two subsections show the relationship between such an algorithm and various modifications to the standard set of unification rules and set the stage for further modifications in the next chapter.

5.3.1. Deriving Implied Constraints

To provide the effects of equating arguments of equated terms, additional edges are added to the graph by examining the maximum flow between various pairs of terms. If two terms with the same functional symbol have a nonzero flow between them, their corresponding arguments are equated with a weight equal to the maximum flow between the terms. (If the arguments had already been equated, this new weight is added to the previous weight.) All pairs of functional terms are repeatedly considered, with additional weight being added to edges between corresponding arguments if additional flow has appeared between the terms, until no changes are made to the graph on some pass over the pairs. The flow between two nodes in the final graph is designed to indicate the maximum number of simultaneously existing equations relating those nodes that could be derived from the original set of equations by some ordering of rewrite rules for transitivity and for equating arguments.

A simple example of this derivation of implied constraints is found in Figure 5.2. Here the graph edges derived directly from the set of equations

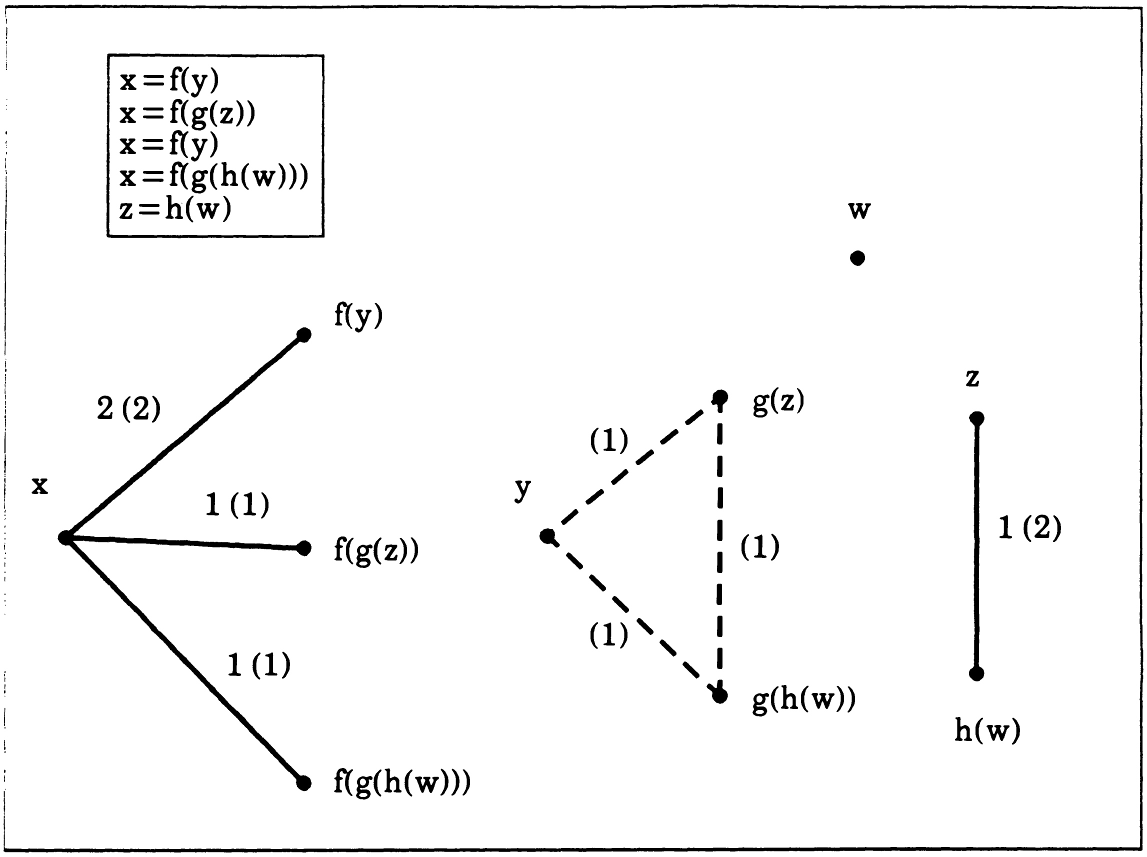


Figure 5.2. A First-Phase Maximum Flow Graph.

are solid lines with their original weights marked; graph edges added by this phase are dashed. Each pair of f terms is connected by one unit of flow and so generates a dashed line connecting the arguments. The g(z)-g(h(w)) edge then strengthens the z-h(w) edge. All the edges have their final weight shown in parentheses.

As this phase of the algorithm makes no provision for creating new functional terms by substitution into existing ones, it alone cannot generate all equations that could be derived by the standard rules. However, a second phase can examine the final graph and determine all constraints on each type variable in the original set of equations along with votes for the dominance of each constraint.

5.3.2. Producing Final Constraints

The first step in determining the final constraint on some variable x is to accumulate the set of functional terms connected, possibly indirectly, to x in the final graph. If the set is empty, x is unconstrained, but its constraint is considered to be the earliest variable (in some variable ordering) equated to x. Although this earliest variable must also be unconstrained, using it as a constraint allows all mutually equated but unconstrained variables to appear as the same variable in final constraints. If the set of functional terms is not empty, it is subjected to a cleaning process in which mutually consistent constraints are merged into a single term.

The set is broken into subsets with one subset for each outermost functional symbol among its terms. As the option corresponding to each subset in the final constraint on x is to be a single term with the appropriate weighting, the subset must be consolidated if there is more than one term. For example, the constraint resulting from the subset {f(g(z)), f(y), f(h(z))} will be of the form f(u)[w_1] where u will have at least the options g(z) and h(z). (As discussed later in this subsection, u may have more options if y is equated to other functional terms.) The weight w_1 is calculated by introducing a temporary node into the final graph and connecting this node to all terms of the subset with edges of infinite weight. The maximum flow between x and this temporary node is assigned to w_1 and indicates the number of times that x could independently be inferred to be an application of the function f.

The same weighting principle is applied throughout the algorithm so that the weight for each option and suboption in the final constraint for x reflects the number of times the type of x could independently be inferred to contain that option or suboption. The maxflow method used above to determine how many times x is an application of f will also be used to determine how many times x is independently asserted to be of any particular functional form. The terms connected to x which share the common functional form are connected to a temporary node with edges of infinite weight, and the maximum flow between x and the temporary node is then the number of times x is directly asserted to be of that form.

As an example of the application of this principle, assume that the variable v is connected to the terms f(g(g(w))), f(g(h(w))), and f(h(y)) as suggested in Figure 5.3. The constraint on v implied by these terms is of the form f(g(g(w)|h(w))[w_3] | h(y)[w_4])[w_2]. The weight w_2, representing the number of times v is asserted to be an f(*), is determined by setting w_2 to the flow from v through the set { f(g(g(w)), f(g(h(w))), f(h(y)) }. In the figure, this would be the flow between v and the temporary "for f(*)" node. Similarly, w_3, representing the number of times v is asserted to be an f(g(*)), is to be the flow from v through terms of that form to "for f(g(*))", and w_4, representing the number of times v is asserted to be an f(h(*)), is to be the flow from v to "for f(h(*))". (The

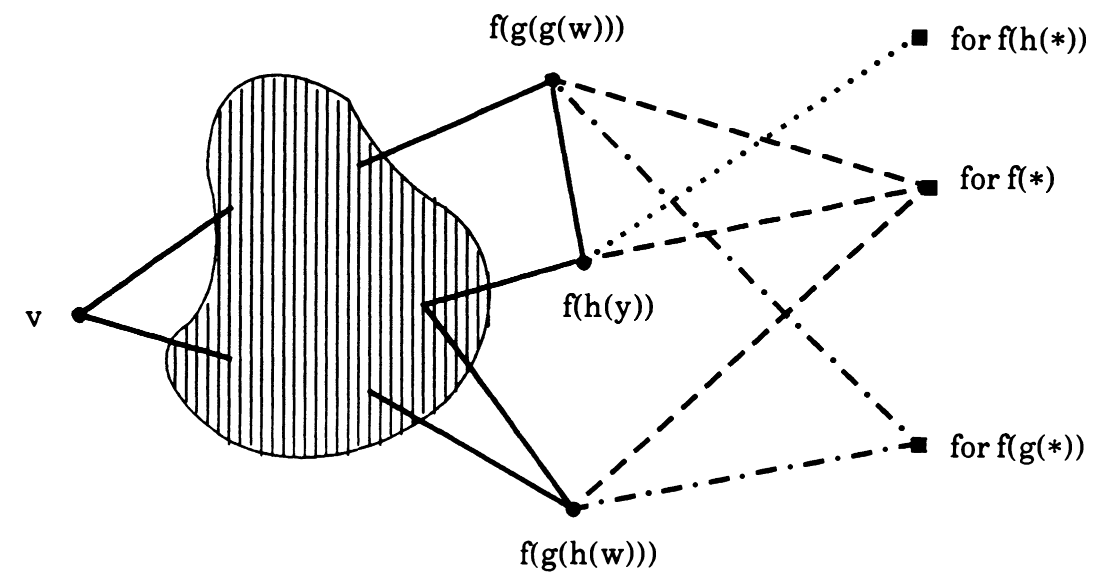


Figure 5.3. A Second-Phase Maximum Flow Graph.

weights associated with the inner terms, f(g(g(*))) and f(g(h(*))) are not shown in this example.)

In addition to direct constraints caused by functional terms connected to a variable, there can also be indirect constraints derived from constraints on variables in those connected terms. In the earlier example, the final constraint for y must include g(z)[w_5] | h(z)[w_6] for some weights w_5 and w_6 as there is nonzero flow through x from f(g(z)) to f(y) and from f(h(z) to f(y)). If y were also equated to other terms, there may be additional options in u, leading to additional options for x. If the only constraints on y are those derived from equating arguments of terms connected to x, y's constraints cannot be used to constrain x because they themselves were derived from constraints on x. If y and g(z) had also been equated for some reason having nothing to do with x, a portion of the g(z) option for y is independent of the terms connected to x and this portion may be used to strengthen the f(g(z)) option for x. If y had been equated to a term q(z) for some reason, the q(z) option for y will generate a f(q(z)) option for x.

However, if x is equated to f(y) with weight 1 and y is equated to q(z) with weight 17, the f(q(z)) option for x should be restricted to weight 1 because there is only one x = f(y) equation in which q(z) can be substituted for y. In general, the contribution of an option for y toward an option for x, where y is a variable found in some subterm of a term connected to x, may be found by first removing the portion of the y option's weight attributable to equating y to corresponding subterms of other terms connected to x and then further limiting the y option's contribution to the number of times x is inferred to be of a type with y occurring in that particular subterm position.

In the algorithm given in Figure 5.4, this approach is used to inductively derive the final constraint on a variable x. The terms connected to x to determine FinalConstraint(x) for a variable x:

1. let S = { functional terms (indirectly) connected to x in final graph }
   if S is empty, return y[1] where y is the earliest variable that is connected to x
2. split S into subsets S.f with one subset for each outermost function symbol in S
3. initialize Options to be empty
   for each subset S.f do
   add FinalOption(S.f,f(*),x) to Options
4. return the disjunction of the elements of Options

Figure 5.4. Algorithm for Producing Final Constraints.

are broken into subsets on their outermost function symbols, and a single final option is found for each subset. This final option is an appropriately-weighted term which may have options in its subterms. Final options from different subsets are known to be incompatible, as each option inherits the root symbol from its subset and each subset is defined to have a different root symbol. Thus the final constraint for the variable is simply the disjunction of the subset options as no further merging can take place. As the algorithm stands, functions with the same name but different numbers of arguments are treated as the same function, although it would be a simple matter to insert a test to force them to be treated differently.

To determine the final option for a subset via the algorithm in Figure 5.5, consider the functional form common to the terms of the subset. Looking at the terms as trees, this common form precisely meets the earlier definition of a branch. When FinalOption is called directly from FinalConstraint, the common functional form is a common root symbol with only placeholders as arguments. When FinalOption calls itself for the first time, the common functional form is a root symbol with one of the argument placeholders replaced by a function symbol with placeholder arguments. One of these to determine FinalOption(S,Φ,x) for a set of terms S connected to the variable x where all elements of S have the common functional form Φ:

1. let q be the deepest function specified in Φ
   (*i.e.*, S is to be broken into subsets on the arguments of q)
2. determine weight(S) by finding the maxflow from x through members of S to a temporary node
3. for each argument slot i of q do

```
split S into subsets S.u with one subset for each function symbol or
    variable u in slot i
initialize Ti, the set of fully expanded versions of terms in slot i, to
    be empty
for each subset S.f, f a functional term, do
    for each subset S.v, v a variable, do
        determine maxflow(v,S.f), the maximum flow from v through
            terms in slot i in S.f to a temporary node
    add FinalOption(S.f,Φ[i/f],x), a weighted term with no variables
        which can be further constrained, to Ti
        (Φ[i/f] is the functional form produced by substituting f(*) into
        slot i of Φ)
for each subset S.v, v a variable, do
    determine weight(S.v) by finding the maxflow from x through
        members of S.v to a temporary node
    find FinalConstraint(v)
    for each option o[w] in FinalConstraint(v) do
        limit weight(o) to min(w-maxflow(v,S.f),weight(S.v)) where S.f
            is the (possibly empty) set with function corresponding to o
        descend o and make sure that no weights exceed weight(o)
        if there is an element u of Ti with the same outer function as o
            descend o and add the weights of u to the appropriate
                suboptions of o
            remove u from Ti and add o
        else
            add o to Ti
set Oi, the final options for slot i, to be the disjunct of the terms in Ti
```

4. return $q(O_1,\ldots,O_n)$[weight(S)]

Figure 5.5. Algorithm for Producing Final Options.

placeholders may in turn be replaced by a function symbol with placeholder arguments, but the common functional form is always of finite depth with only one function symbol at each level of nesting.

The final option for the subset is the deepest function symbol in the common functional form, applied to the result of independent processing for each argument slot, weighted by the strength of connection of terms sharing the common functional form to the variable whose constraint is being determined. The caller of FinalOption will then bundle this option with any other contradictory options at the current level.

The result of argument slot processing in FinalOption is a weighted disjunction of the various incompatible options that could occur in that slot. The processing for an argument slot of the deepest function begins, as in FinalConstraint, by breaking the subset into smaller subsets based on the function symbols and variables found in that argument slot. For each function symbol, FinalOption is called recursively with an updated common functional form and set of terms to determine the weight and suboptions of the option derivable from terms with that function symbol. This option is incompatible with the option from any other function symbol, so it is added to the set of incompatible options. At the same time, the contribution of this function symbol to the constraints of any variables in this slot is calculated.

For each variable in the slot, consider the options in its final constraint. The same reasoning given in the earlier example for the constraint on y will generalize to show that the variable constraint must include information gathered from other terms in the slot, so a variable option already contains all suboptions of the option derived from any terms in the slot with the same function symbol. Thus after removing the contributions of slot terms and limiting the variable option's weight to the strength of connection between the original variable and terms containing the slot variable, one is left with the contributions unique to that variable. At this point, one may merely add the weights of suboptions derived from slot terms to the weights of the corresponding suboptions derived from the slot variable (which amounts to merging the term and variable contributions), remove the term options from the incompatible set, and add the newly-weighted variable options to that set.

5.3.3. Extension to Recursive Types

The above discussion of type inference in the presence of conflicts has assumed that no recursive type constraints were implied by the original set of equations. However, modifying the algorithm to deal with recursive types is a simple process.

Having recursive constraints implied by the original set of equations requires no modifications whatsoever to the first phase of the algorithm, as shown by:

Theorem 5.3.3.1. The first phase of this conflict-tolerant algorithm will terminate even in the presence of recursive types.

Proof: As there are a finite number of subterms in the original equations, there are a finite number of possible links. For nontermination to occur, some link (t_1,t_2) must be strengthened an infinite number of times. After the initial pass over pairs of functional subterms, a link (t_1,t_2) will be strengthened only if t_1 and t_2 are corresponding arguments of two functional subterms whose connection strength has been increased. As there are a finite number of pairs of subterms that can strengthen the (t_1,t_2) link, some pair $\langle f_1,f_2 \rangle$ must have its connection strength increased an infinite number of times. Since the connection strength between f_1 and f_2 cannot exceed the sum of the weights of edges leading from f_1, some link (f_1,t_3) must be strengthened an infinite number of times. By the same argument, this infinite strengthening is possible only if f_1 is an argument of some functional subterm f_3, which must in turn be an argument of some functional subterm f_4. As all subterms from the original set of equations are of finite size, this regression cannot be continued indefinitely as demanded by the premise that some link is strengthened infinitely often. □

Dealing with recursive constraints in the second phase of the algorithm is merely a matter of keeping track of the variables on which work is being done. If a variable is encountered again during the derivation of its constraint, the algorithm does not try to reanalyze the same constraint but instead marks that variable occurrence as recursive. If types are being maintained as a data structure with pointers to various records, a recursive occurrence will be replaced by a pointer to the final constraint on the variable, when determined. The same effect is achieved when types are maintained as character strings by replacing a recursive occurrence of x by a string such as "@x" and marking the final constraint with a subscript as in $(f(@x)[2]|g(@x)[3])_x$. Any recursive occurrence is considered incompatible with other options (although it must subsume them) because there seemed to be no clean way to combine the weights from other options with those from the options of an unrolled form of the recursion marker.

5.3.4. Comparison to Standard Rules

The algorithm described in the previous sections determines the same type for a variable as do the standard rewrite rules in those cases for which the standard rules apply. (These are the cases with no type conflicts and no implied recursion.)

Consider the following set of transitive rewrite rules which does not distinguish between the left and right sides of equations. (By expanding rule

(iii) into four rules and adding a rule to reverse $t=x$ to $x=t$, these rules could be transformed to an equivalent set which distinguishes between the sides of equations.)

(i) Given $x=u$ where x does not occur in u, replace occurrences of x in other equations with u.

(ii) Given $f(u_1,...,u_n)=f(v_1,...,v_n)$, replace the equation by n equations, $u_1=v_1,...,u_n=v_n$.

(iii) Given $u_1=u_2$ and $u_2=u_3$, add the equation $u_1=u_3$.

(Note that this set of rules adds new equations through transitivity instead of replacing two old equations with a new equation as was suggested to motivate the subsection on deriving implied constraints.)

While the rules as written above give the spirit of the proposed unification process, two modifications are made to the conditions for applying these rules to ensure termination. First, rule (ii) is modified to leave the given equation in the set but to mark it so that it cannot be used again for argument equation. Second, a set of ancestor equations is associated with each equation. An equation is considered to be an ancestor of itself, and a new equation generated by rule (iii) will get the union of the ancestors of its parents. (Original equations and equations generated by rule (ii) have no explicit ancestors in this scheme.) Before a proposed application of rule (iii), a check must be made to ensure that no equation that is identical to the one proposed by the rule application and that has an ancestor in common with the proposed equation already exists in the equation set. This restriction, which basically prohibits any equation from being used in more than one similar transitive chain, prevents any pair of equations from generating multiple new equations through rule (iii) and also prevents cycles where equations regenerate their ancestors through rule (iii). As these modifications do not restrict the set of equations which can be generated by the transitive rule set, but merely ensure termination, we proceed with the analysis of possible equations derived by the transitive rule set.

Theorem 5.3.4.1. Unification using this set of transitive rewrite rules is equivalent to unification using the standard set of rules:

(I) Given $x=u$ where x does not occur in u, replace occurrences of x in other equations with u.

(II) Given $f(u_1,...,u_n)=f(v_1,...,v_n)$, replace the equation by n equations, $u_1=v_1,...,u_n=v_n$.

(III) Given $x=x$, discard the equation.

(IV) Given $t=x$, replace the equation by $x=t$.

The sets of rules are equivalent in the sense that both will produce the same constraints on variables as long as there are no conflicts inherent in the original set of equations.

Proof:

Rules (i) and (ii) are identical to rules (I) and (II).

Rule (IV) puts an equation into a form acceptable to rule (I) by interchanging the left- and right-hand sides. Rule (i) can already handle an equation with left- and right-hand sides reversed as its algorithm does not distinguish the left side from the right. Rules (II) and (III) treat their left- and right-hand sides identically so there is no need to distinguish between the sides.

Therefore rules (i) and (ii) can generate all equations generated by rules (I), (II), and (IV). Rule (III) does not generate any new equations.

Thus the transitive set of rules can generate any equation (and thereby any constraint) generated by the standard set, and we need only show that the standard set can generate any variable constraint that the transitive set can.

If the transitive set is to generate anything extra, it must be by way of rule (iii).

Let us consider the cases in which rule (iii) is applied.

(1) $u_1 = x$, $x = u_2$

The equation $u_1 = u_2$ is also generated by rule (I).

(2) $x_1 = f(u_1,...,u_n)$, $f(u_1,...,u_n) = x_2$

The types of x_1 and x_2 are forced to be identical under the standard set of rules because rule (I) will make the same substitutions into both occurrences of $f(u_1,...,u_n)$.

(3) $x = f_1(u_1,...,u_n)$, $f_1(u_1,...,u_n) = f_2(v_1,...,v_n)$

If f_1 is not the same function as f_2, there is a conflict in the system of equations. If f_1 is the same as f_2, rule (II) will equate their arguments. Any resulting restrictions of variables in the terms $u_1,...,u_n$ will be substituted back into the equation $x = f_1(u_1,...,u_n)$. Thus x will be equated to the unifier of $f_1(u_1,...,u_n)$ and $f_2(v_1,...,v_n)$. This is the same result as would be produced from $x = f_1(u_1,...,u_n)$ and $x = f_2(v_1,...,v_n)$ after applying rule (iii).

(4) $f_1(u_1,...,u_n) = f_2(v_1,...,v_n)$, $f_2(v_1,...,v_n) = f_3(w_1,...,w_n)$

Rule (iii) would produce $f_1(u_1,...,u_n) = f_3(w_1,...,w_n)$ which would then become $u_1 = w_1,...,u_n = w_n$. Rule (II) would produce $u_i = v_i$ and $v_i = w_i$ for each argument position. By induction on the nesting depth of these subterm equations, nothing extra is gained by adding the equations $u_i = w_i$. □

Theorem 5.3.4.2. A unification algorithm using the set of transitive rules which performs all possible applications of transitivity and equating arguments before doing any substitution is equivalent to an algorithm which interleaves applications of all three rules.

Proof: The only way in which this equivalence could fail to hold is for a substitution to allow the production of new equations. For these equations to be new, they cannot have been produced by transitivity and equating arguments alone, nor by applications of substitutions after the other rules are finished.

Substituting $y=g(...)$ into $u=y$ gives the same result as transitivity, so a problem could only arise when substituting y into a functional term. Consider the substitution of $y=g(...)$ into the equation $u=f(...y...)$, with y being the i^{th} argument, yielding $u=f(...g(...)...)$. If u is a functional term $f(u_1,...,u_n)$, a new equation $u_i=g(...)$ can be generated by equating arguments. However, this equation had already been produced by using transitivity on $y=g(...)$ and $u_i=y$ which was obtained by equating arguments prior to the substitution.

If there is an additional equation $u=v$, an additional equation $v=f(...g(...)...)$ can be generated from $u=v$ and $u=f(...g(...)...)$ by transitivity after the substitution for y. However, this equation could also be produced by substitution directly into the equation $v=f(...y...)$ generated from $u=v$ and $u=f(...y...)$. Thus there is no need for additional applications of transitivity and equating arguments to be interspersed with substitutions. □

Taken together, these theorems show that the main ideas of the maximum-flow algorithm, transitive unification and moving substitution into a second phase, are sound.

5.3.5. Comparison to Design Criteria

The earlier algorithm is subject to pairwise consistency discrepancies due to early merging, but such merging allows for a much simpler and faster one-phase algorithm. This second algorithm avoids the pairwise consistency problems by separating constraint derivation into two phases but incurs a considerable cost in complexity and loss of intuition about its workings.

Although the maximum-flow algorithm produces unique, reproducible, weighted constraints from a set of input equations, these weights are not necessarily the intuitively "correct" ones. This second algorithm produces every constraint implied by the original set, and it produces intuitively acceptable weights for nonrecursive cases. However, when presented with multiple forms of "the same" recursive equation, the weighting seems wrong. For example, given $\{ x = f(x), x = f(f(x)) \}$, this algorithm concludes (by adding in the result of equating the terms' arguments) that x is an application of f with weight 3. There are two arguments against this being correct. First, the occurrences of x inside the terms are perhaps best thought of as implicit recursion markers instead of actual occurrences. Second, even if they are treated as normal occurrences, the $x = f(x)$ equation created by equating the arguments cannot be considered to be independent of the parent equations. Without recursive occurrences, parents and children cannot try to constrain the same variable, so the problem does not arise in the nonrecursive case.

Given the relative lack of intuition behind the weights of the maximum-flow algorithm and its disagreement with intuition in some recursive cases, it is time to step back and carefully define weights in a way that better accords with intuition. And yet, the maximum-flow algorithm need not be discarded. Since it does derive all possible constraints and give them unique weights, it can be useful in determining everything contributing to a conflict in those cases where an error-tolerant, order-independent algorithm is necessary but recursive types are not allowed or the exact weightings are not critical. If the application demands only a solution with all possible constraints (and perhaps the reasons behind any conflicting constraints) but does not care about the weighting information, the algorithm can be simplified considerably (compare Figures 5.5 and 5.6) by merely checking the graph

```
to determine FinalOption(S,Φ,x) for a set of terms S connected to the
variable x where all elements of S have the common functional form Φ:
1. let q be the deepest function specified in Φ
       (i.e., S is to be broken into subsets on the arguments of q)
3. for each argument slot i of q do
       split S into subsets S.u with one subset for each function symbol or
           variable u in slot i
       initialize Ti, the set of fully expanded versions of terms in slot i, to
           be empty
       for each subset S.f, f a functional term, do
           add FinalOption(S.f,Φ[i/f],x), a term with no variables which
               can be further constrained, to Ti
               (Φ[i/f] is the functional form produced by substituting f(*) into
               slot i of Φ)
       for each subset S.v, v a variable, do
           find FinalConstraint(v)
           for each option o in FinalConstraint(v) do
               if there is an element u of Ti with the same outer function as o
                   remove u from Ti and add o
               else
                   add o to Ti
       set Oi, the final options for slot i, to be the disjunct of the terms in Ti
4. return q(O1,...,On)
```

Figure 5.6. Revised Algorithm for Producing Final Options.

for connectivity instead of calculating the maximum flow between nodes. Even this simplification provides more information than many algorithms in current use, which complain whenever something is not of the "correct" type without giving any indication of whether different "incorrect" types were encountered. The algorithms developed in the next two chapters give answers agreeing with intuition in both nonrecursive and recursive cases, but such algorithms cannot be easily modified to perform the simpler task of extracting only unweighted constraints.

Chapter Six

Error-Tolerant Counting Type Inference

Since there is occasional disagreement between the counts produced by unaided intuition and the counts defined by an algorithm developed from the sole intuitive idea of maximum flow, there must be another intuitive principle at work. The basis for the maximum flow idea was that no type constraint should be allowed to influence another constraint more strongly than either constraint was asserted independently. As another formulation of this basis, the intrinsic strength of an input equation does not depend upon any other input equations -- a single $y=g(z)$ equation asserts y is a $g(*)$ with strength 1, no matter how many $x=f(y)$ equations there are, so the single $y=g(z)$ can only help assert that x is a $f(g(*))$ with strength 1. A way to express this idea in the rule-based unification framework is to modify the substitution rule to allow only a single substitution by each equation, instead of allowing a single $y=g(z)$ to help create indefinitely many $x=f(g(z))$ equations. One derivation sequence using single substitution may not derive all the constraints from a consistent set of equations that would show up when using the standard rules, but each standard constraint will appear at some point in some single-substitution derivation sequence. Proper coordination of single-substitution derivations, as presented in succeeding sections, will provide us with all possible constraints, whether consistent or not, along with their appropriate weights.

6.1. Single Substitution Unification

Single substitution unification can be expressed by the following four rules, three of which come directly from the standard unification set:

(1) Given an equation $x=x$, delete the equation from the set.

(2) Given $t=x$, replace the equation by $x=t$.

(3) Given $f(u_1,...,u_n)=f'(u_1',...,u_m')$, where $f=f'$ and $n=m$, replace the equation by the n equations $u_i=u_i'$.

(4) Given $x=u$, where x does not occur in u but x does occur in some other equation, replace one occurrence of x in another equation by u and remove the $x=u$.

Note that the transitivity so natural to the graphical approach is not reflected in this set of rules. Adding a transitive rule to this set does not change the character of any of the development that follows, but it would seem to decrease the usefulness of the unified solutions. Given both x=int() and x=bool(), transitivity would propagate this conflict to every other variable equated to int() or bool(). This propagation would seriously hamper attempts to pinpoint the source of type errors.

Theorem 6.1.1. Repeated application of these single-substitution unification rules will always terminate.

Proof: For a finite set S of equations, let s(S) be the number of occurrences of function and variable symbols in S, and let r(S) be the number of equations of the form $t=x$ in S. Both s(S) and r(S) are well-defined nonnegative integers for any such set S. Now let $d(S)=\langle s(S), r(S)\rangle$ and use a lexicographic ordering on these pairs. Each of the rules decreases the value of this function d in a way that excludes the possibility of an infinite sequence of rule applications. For each case, let S be the set of equations before the rule application and T be the set afterwards.

Using rule (1), $d(T)=\langle s(S)-2,r(S)\rangle$. Using rule (2), $d(T)=\langle s(S),r(S)-1\rangle$. Using rule (3), $d(T)=\langle s(S)-2,r(T)\rangle$ where r(T) may be greater than r(S), but the amount of increase is limited to the number of arguments of the function in the deleted equation. Using rule (4), $d(T) = \langle s(S)-2, r(T) \rangle$ where $r(T)$ may be one more than $r(S)$ if the x substituted for was the left-hand side of an equation in S. Since the value of s never increases and the value of r increases by a bounded amount at the time of a decrease in s, there cannot be an infinite sequence of rule applications. □

Since the parent equations are removed during the application of each rule, there is no temptation to reuse an equation inappropriately. Sets like $\{x=f(y), x=f(y), x=f(y), y=g(z)\}$ and $\{x=f(y), y=g(z), y=g(z), y=g(z)\}$ can each only produce one $x=f(g(z))$ equation, while $\{x=f(y), x=f(y), y=g(z), y=g(z)\}$ can produce two. For all three sets, the answer agrees with intuition (and incidentally with the maximum flow answer). In general, the weight of a constraint option should be the number of independent assertions supporting that option, which under this algorithm is the number of simultaneous equations making that assertion.

However, removing equations when they are used for substitution again makes the generated equations sensitive to the order in which rules are applied, as shown in Figure 6.1. Each path from the original set of equations is a possible *derivation sequence*. Each node on that path is a *stage* of the derivation sequence. To reduce the clutter in the figure, stages with the same sets of equations, even if from different derivation sequences, have been identified with each other. Depending on the derivation sequence, zero to two $x=f(g(z))$ equations are produced. Even more strikingly, for this set of input equations, the final set of equations will always be empty. Clearly, this formulation of the single-substitution unification rules is not the final answer. To weigh branches properly (as defined in Section 5.1) we will have to coordinate the various derivation sequences.

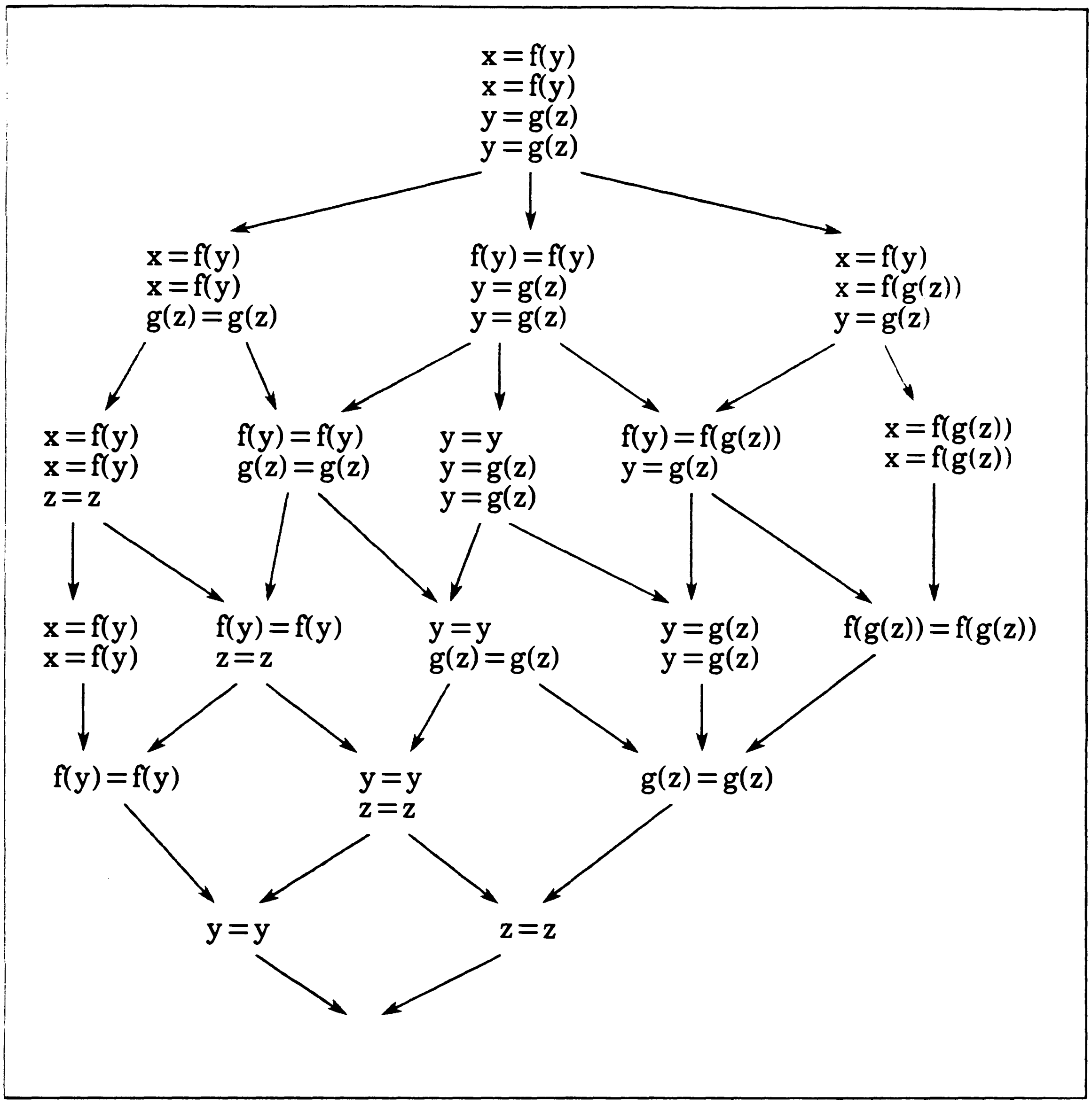


Figure 6.1. Derivation Sequences Using Single Substitution.

6.2. The Union of Derivation Sequences

In the above definition of the weight of a constraint option, "the number of simultaneous equations making that assertion" was implicitly the maximum number of simultaneous equations that could appear in any stage of any derivation sequence. However, as can be seen from a glance at Figure 6.1, the large number of possible derivation sequences makes computing weights directly from this definition (by enumerating all the possible stages) prohibitively expensive. What we need is a way to compute the union of the equations appearing in all possible derivation sequences, with ways to tell which equations appear at the same time (*i.e.*, in the same stage) in some derivation sequence.

For the moment, this concept of union is necessarily fuzzy as we have not yet defined a clear notion of when equations are "the same". Intuitively, the two $x = f(y)$ equations in the stage at the left of the second row in Figure 6.1 are the same two as in the initial set, but in collapsing these four equations to two in the union, we must be careful not to further collapse them to a single $x = f(y)$ equation if we are to have any hope of counting equations properly.

Since set union is a commutative and associative operation, we can expect that the union over all derivation sequences from an original set of equations should be independent of the order in which equations were added to the union. Of course, before we can prove this, we must formally define what we mean by "the union over all derivation sequences".

Since the concept of "union" implies that any equation added to the set must stay in the set, we can no longer remove equations to prevent them from being used too often. Instead, we can keep track of which equations and which rule created each additional equation. (Doing this requires that every equation be uniquely identified to distinguish it from other, identical equations; equations will later get unique numbers for this purpose.) By backtracking along these parent pointers, we can find a *derivation tree* consisting of every equation used in the process of constructing a particular equation. Technically, the structure resulting from these parent pointers would be a derivation DAG, but we are more interested in tracing ancestry than in the actual DAG nature. (Similarly, genealogical family trees are often really family DAGs after a few generations, but as long as the rejoinings are sufficiently remote, people are much more interested in the tree structure than in the existence of the joins.) Figure 6.2 shows the derivation of y=w from two original equations. Reversing the direction of these derivation arrows yields the parent pointers, and thus the derivation tree, for each equation, as shown in Figure 6.3. In both figures, dashed lines indicate that an equation was created by equating arguments, and thus that the equation depends on only one argument position of its parent.

Derivation trees, when combined with the unique numbering of original equations, give us a clean definition of when textually identical equations in different stages of derivation sequences are "the same". As the base case, two occurrences of original equations are the same equation iff they have the same number. Then inductively, two derived equations are the same iff they have the same parents and were derived by the same rule

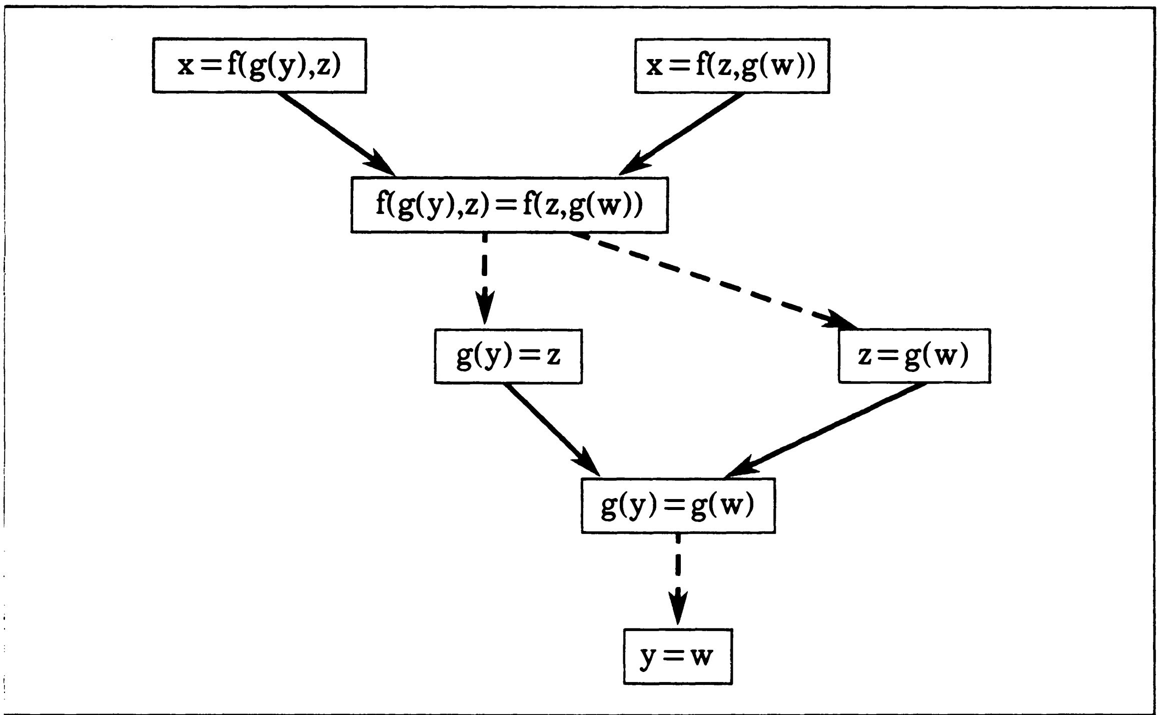


Figure 6.2. Deriving an Equation.

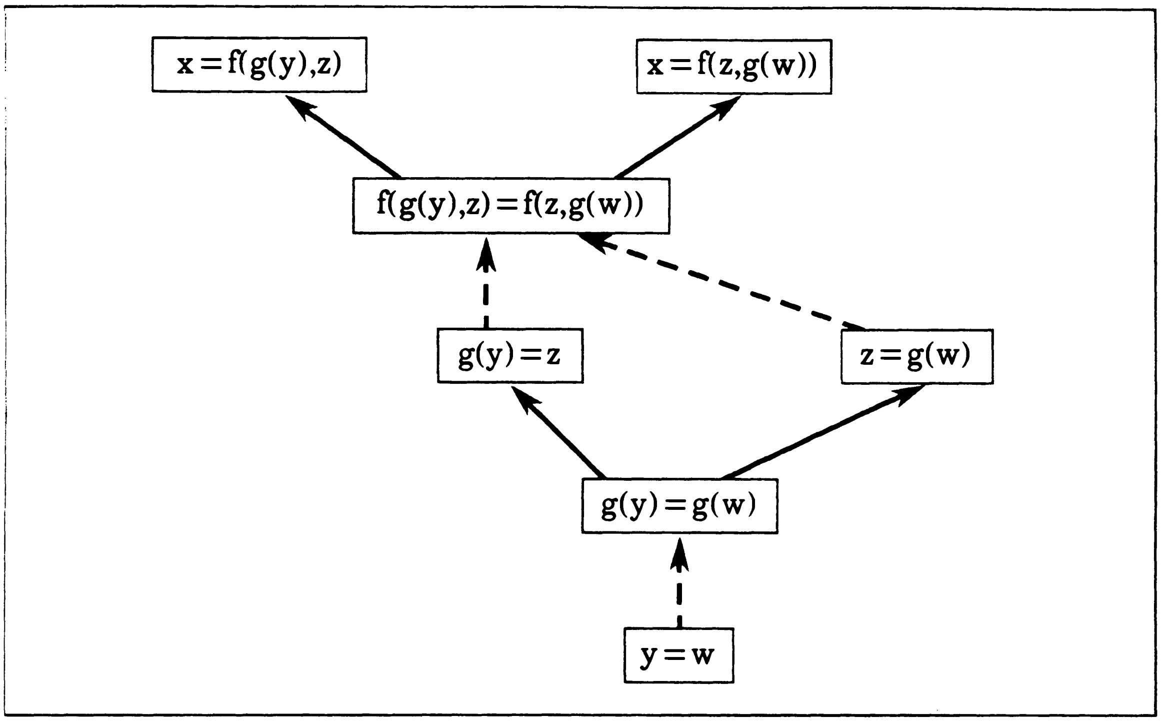


Figure 6.3. The Derivation Tree from Figure 6.2.

applied to the same sites in the parents. From here on, an "equation" should be taken to be a uniquely identified equation from this definition.

We say that derivation trees *have a dependency conflict* (or simply that they *conflict*) if they share a subtree, unless each tree equated the arguments of the root of the (maximal) shared subtree and then used a child equation from a different argument slot. As a shorthand notation, two equations are said to conflict if their derivation trees do. Therefore, in the sample derivation shown in Figure 6.3, the equations $g(y)=z$ and $z=g(w)$ do not conflict since they use different arguments of the shared subtree root $f(g(y),z)=f(z,g(w))$, but the equations $x=f(g(y),z)$ and $g(y)=g(w)$ do conflict since $x=f(g(y),z)$ appears in both trees. The following important theorem shows why this definition of conflicts is both natural and useful:

Theorem 6.2.1. Two equations appear at the same time (*i.e.*, in the same stage) in some derivation sequence iff their derivation trees do not conflict.

Proof: The single-substitution unification rules remove parent equations in the process of creating new equations. Thus, the equations found in a derivation tree (except for the equation at the root of the tree) are those which would necessarily have been removed in any derivation sequence containing the tree. (A derivation sequence contains a tree if the sequence contains the derivations indicated by the nodes of the tree.) Also note that equating arguments is the only rule which can create more than one child equation from a parent.

Assume two equations appear at the same time in some derivation sequence. If their derivation trees do not share any subtrees, the equations do not conflict, so assume the trees share a subtree rooted by the equation E. E cannot be the root of both derivation trees, or we would have only one equation instead of two. E cannot be the root of one derivation tree, because an equation cannot exist at the same time as any of its descendants (*e.g.* the root of the other derivation tree). Only one rule in the derivation sequence can use E, as the first rule application will remove it, making it unavailable for any subsequent applications. The descendants of E used in the two trees must be different, or E would not be the root of a maximal shared subtree. Thus E was used by the argument equation rule, and different argument positions were used in the different trees. Therefore the derivation trees do not conflict.

Conversely, assume two derivation trees do not conflict. The nodes of the first derivation tree can easily be written linearly and applied to the initial set of equations to give a minimal derivation sequence containing that tree. Trimming any shared subtrees from the second tree and appending a linear form of the remainder of that tree to the first sequence gives a derivation sequence which ends with both root equations appearing at the same time.

The only way in which this concatenated sequence could fail to be a valid derivation sequence would be if the second part tried to use an equation which no longer existed. To eliminate this possibility, we need only show that all the leaves of the trimmed tree exist after the end of the first part of the sequence. Leaves which were leaves of the full second tree are original equations which are not mentioned by the first tree and so are not used in the first sequence. Leaves which were not leaves of the full second tree are children of shared equations. Since the trees did not conflict, the shared equations must have produced equated arguments with the trees using different positions. Therefore the first sequence has, in the process of getting the argument position it needs, already produced the other argument positions for the second part to use, and the constructed derivation sequence is valid. □

Using induction to extend these ideas to multiple trees, we find:

Theorem 6.2.2. A set of equations appears together in some derivation sequence iff none of the derivation trees for those equations conflict.

This theorem tells us how to identify equations that appear together at a stage of a derivation sequence, so to define the union of the derivation sequences algorithmically, we need only make sure that every equation that can occur in a derivation sequence is actually generated and that this process of generating equations eventually terminates.

6.3. A Set of Rules for Error-Tolerant Countable Unification

To ensure termination in this rule-based setting, we record which equations have reacted under each rule and specify that the same rule may not be applied again to the same set of equations. We also record the creators of each new equation, so that we are able to restrict rule applications to equations that can exist at the same time. Putting these ideas into the formal rule-based setting leads to the following set of rules for error-tolerant counting unification:

(1) Given an equation $x = x$, mark the equation as deleted.

(2) Given $t = x$ which has not been reversed, add $x = t$ with creators Reversal and the parent equation; record that the parent has undergone Reversal.

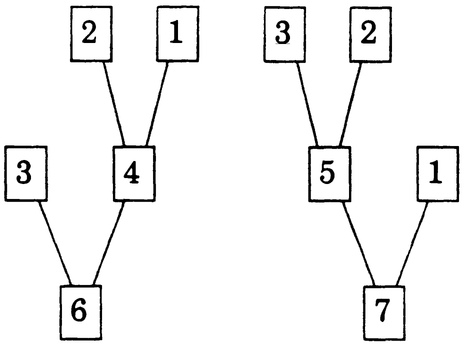
(3) Given $f(u_1,\ldots,u_n) = f(u_1',\ldots,u_n')$ which has not had its arguments equated, add $u_1 = u_1'$, ..., $u_n = u_n'$ with creators EquateArgs and the parent equation; record that the parent has undergone EquateArgs.

(4) Given $x = u$ where x does not occur in u, and $v = w$ where x occurs in v or w, where the parents do not conflict and $x = u$ has not been substituted into $v = w$, for each occurrence of x in $v = w$, add a new equation with that single occurrence replaced by u with creators Subst and the parent equations; record that $x = u$ has been substituted into $v = w$.

Before proceeding with an analysis of this set of rules, consider the very simple example shown in Figure 6.4. The first list of equations is the desired union over all derivation sequences from the first three equations. The subsequent equations, all of which are created by substitution in this case, are marked with their parents. The second equation list, giving the final weights for each branch equated to each variable, is the result of examining the first list to count the independent (*i.e.* nonconflicting) occurrences of each branch. As would be expected from the input set, this final answer gives everything a weight of one. (Equations 6 and 7 conflict because their derivation trees, shown at the right of the figure, share leaves.)

The algorithms for processing a union of derivation sequences are given in Figure 6.5. As in the graph algorithm, terms are recursively divided for counting based on their structure. In this case, however, the complex

Equation list:
1: x = f(y)
2: y = g(z)
3: z = h(w)
4: x = f(g(z)) [[2, 1]]
5: y = g(h(w)) [[3, 2]]
6: x = f(g(h(w))) [[3, 4]]
7: x = f(g(h(w))) [[5, 1]]



Equation list:
8: x = f(g(h(w)[1])[1])[1]
9: y = g(h(w)[1])[1]
10: z = h(w)[1]

Figure 6.4. A Union of Derivation Sequences.

problem of determining the independent strength of an assertion vanishes. Given the set of equations asserting a branch, the weight of the branch is simply the maximum number of those equations that could exist at the same time. This weight is determined by recursively marking the derivation trees rooted by the branch equations and seeing how many such trees can coexist. Such a weighting algorithm, intended to be called as Count(S,0,maxcount) with maxcount initially 0 to find the maximum number of independent equations in a list S, can be found in Figure 6.6.

Now that we have an algorithm that purports to define the union of derivation sequences and another that can (by Theorem 6.2.2) properly count the branches in such a union, we must show that the purported union is correct.

Theorem 6.3.1. The union of derivation sequences as defined by repeated applications of the error-tolerant countable unification rules is the union of

```
to find the constraints on the variable x from a union U of derivation
sequences,
1.extract the set S of x = u equations from U
2.split S into subsets S.f with one subset for each outermost function
  symbol (and each different number of arguments) in the right-hand sides
3.if there are no S.f's,
      let Options be the earliest variable in the right-hand sides of S
  else
      initialize Options to be empty
      for each S.f,
          add BranchCount(f(*),S.f) to the disjuncts in Options
4.return Options

to determine the weighted BranchCount(Φ,S) for a set of equations S with
the same left-hand-side variable and with right-hand sides consistent with
the functional form Φ,
1.let q be the deepest function specified in Φ
2.for each argument slot i of q do
      split S into subsets S.f with one subset for each function symbol in
      slot i of a right-hand side
      if there are no S.f's,
          let O_i be the earliest variable in slot i
      else
          initialize O_i to be empty
          for each S.f
              add BranchCount(Φ[i/f],S.f) to the disjuncts in O_i
              (Φ[i/f] is the functional form produced by substituting f(*) into
              slot i of Φ)
3.return q(O_1,...,O_n)[weight(S)]
```

Figure 6.5. Processing the Union of Derivation Sequences.

finding the maximum number of independent branches via Count(eqnlist,incount,maxcount), where incount is the number of derivation trees marked before Count is called and maxcount is the (updatable) maximum number that have been marked at one time:

1. thiseqn ← first equation in eqnlist
2. while thiseqn exists do
    mark thiseqn's derivation tree using its unique number
    if the tree does not conflict with the other marked trees,
        if maxcount = incount, then increment maxcount
        Count(next equation in eqnlist, incount + 1, maxcount)
    clear marks of thiseqn's number from its derivation tree
    thiseqn ← next equation in eqnlist

Figure 6.6. Counting Independent Equations.

all stages of all derivation sequences using the single-substitution unification rules.

Proof: Comparing Theorem 6.2.1 with the conditions needed to apply each error-tolerant countable unification rule, we know that every equation inserted into the union will occur in some stage of some derivation sequence, as its nonconflicting parents occur together in some stage of some sequence. Conversely, by induction on the number of steps in a derivation sequence, every equation occurring in some stage of some derivation sequence will be inserted into the union, as its parents occur together in the preceding stage of that sequence and so do not conflict. □

Theorem 6.3.2. The union of the derivation sequences is always finite, so the algorithm to find this union will always terminate.

Proof: Theorem 6.1.1 shows that the length of any given derivation sequence is finite. Since the original set of equations is assumed to be finite and each single-substitution unification rule adds only a finite number of equations, the number of equations at each stage of a derivation sequence is finite. With a finite number of equations, there are only a finite number of possible applications of single-substitution rules available at each point. This number of possible applications limits the splitting into distinct derivation sequences. Finite splitting at each of a finite number of steps means there are only a finite number of distinct derivation sequences. Since each derivation sequence contains only a finite number of equations, the union of the derivation sequences is also finite.

Since the conditions for applying each of the error-tolerant countable unification rules prohibit multiple applications to the same equations, an equation cannot be inserted into the union infinitely often, so the algorithm always terminates. □

6.4. Implementing Error-Tolerant Countable Unification

While the rule-based setting is convenient for proofs about the set of possible equations, it is not particularly convenient for implementation on a sequential computer. The rules have elaborate guards which ensure their correctness and termination for any possible ordering of derivation steps -- that is, for nondeterministic selection of the rules. The nondeterministic selection process always creates the same set of equations for the union of the derivation sequences; only the order in which equations enter that set varies. A deterministic algorithm for rule selection can use far simpler and less expensive guards by inserting equations in a definite, repeatable order based on the ordering of the input equations. Since the counting procedure is sensitive only to the elements in the union, not the order in which they were inserted, selecting rules deterministically does not affect the analyses of the preceding section.

The particular deterministic algorithm used by the implementation represents a set of equations as a linked list. The starting list consists of the input equations, each of which is marked as not having undergone argument equation, as never having been used for substitution, and as having no parents. Each equation added to the tail of the list by some rule is then marked as not having undergone argument equation and as never having been used for substitution in addition to being marked with its creating rule and parents.

This algorithm, as sketched in Figure 6.7, then repeatedly traverses the list from head to tail. For each equation, any possible rule applications which could not be done on earlier passes through the list are performed. The argument equation and substitution markers are combined into a pointer, lasteqn, representing the last equation which was tested against a given equation. (When an equation is inserted into the list, it has been tested against no other equations, so the pointer is null. After an equation has been

```
while equations are being added do
        thiseqn ← the head of the list
        while thiseqn exists do
                if thiseqn's lasteqn is null then
                        try to equate the arguments of thiseqn
                        thateqn ← the head of the list
                else
                        thateqn ← the successor of thiseqn's lasteqn
                mark thiseqn's derivation tree with its equation number
                while thateqn exists do
                        if thateqn's derivation tree does not conflict,
                            try to substitute thiseqn into thateqn
                        thateqn ← the successor of thateqn
                clear the marks from thiseqn's derivation tree
                thiseqn's lasteqn ← the last existing equation
                thiseqn ← the successor of thiseqn
```

Figure 6.7. Implementing Countable Unification.

checked for equating its arguments and tested against other existing equations, the pointer will be no longer be null and picking up the testing from the subsequent equation allows us to avoid repeating previous tests.) Since each equation is tested against each other equation exactly once, this algorithm's running time is quadratic in the total number of equations produced.

Only two of the four rules are being actively used in the algorithm sketch. The rest of the implementation is not sensitive to which side of an equation a term is on, so the reversal rule is no longer necessary. The removal of $x = x$ equations has been extended to the removal of any identity equation ($u = u$), as such identities do not serve to constrain anything. Input identities are removed before the main loop is started; identities which might be created later are detected before they can be inserted into the list.

These algorithms were earlier applied by hand to create the example of a union of derivation sequences in Figure 6.4. Other examples, this time the actual output of the implementation, are shown in Figures 6.8 and 6.9. Figure 6.9 also shows how little complexity the actual occurrence of conflicts adds. For reasons which will become clear over the next few sections, the implementation keeps track of the substitution history of terms by retaining vestiges of the variables for which substitutions have been made.

6.5. Restricting the Set of Generated Equations

Since in many cases the same information can be obtained by any of a number of different derivations, we would, for efficiency's sake, like to be able to eliminate some derivations from the union of derivation sequences processed by the separate counting phase. Before eliminating any derivations, however, we must be certain that the remaining derivations still have the same number of independent assertions for each branch as did the full union.

Equation list:
1: x = f(y)
2: y = g(z)
3: x = f(w)
4: f\x(y) = f(w) [[1, 3]]
5: f(g\y(z)) = x [[2, 1]]
6: f\x(w) = f(y) [[3, 1]]
7: f\x(w) = f(g\y(z)) [[3, 5]]
8: y = w [[4]]
9: w = y [[6]]
10: w = g(z) [[8, 2]]
11: w = g(z) [[9, 2]]
12: g\y(z) = w [[2, 8]]
13: g\y(z) = w [[2, 9]]

Equation list:
14: x = f(g(z)[1])[2]
15: y = g(z)[1]
16: w = g(z)[1]

Figure 6.8. Another Example of Counting Unification.

Equation list:
1: x = f(y)
2: x = g(z)
3: x = f(y)
4: f\x(y) = g(z) [[1, 2]]
6: g\x(z) = f(y) [[2, 1]]
7: g\x(z) = f(y) [[2, 3]]
9: f\x(y) = g(z) [[3, 2]]

Equation list:
10: x = g(z)[1] | f(y)[2]

Figure 6.9. An Example of Counting Unification with Conflicts.

(A branch is still the augmented path from the root of a type tree, as in Section 5.1.)

One restriction, already mentioned in the preceding section, involves immediately removing any identity equations from the set. Another easily understood restriction involves removing any equation which is an exact duplicate of one of its ancestors. Looking at the derivation in Figure 6.10, it is clear that the second $x = f(y)$ equation gives no information not found in its ancestor. In the general case, we have two identical equations, one of whose derivation trees is a subtree of the other. The identical equations themselves conflict, as will the identical results of any rule applied to both equations. Since these two equations cannot generate more independent votes than a single equation, one of them should be unnecessary. The descendant equation conflicts with everything its ancestor does, as well as with some additional equations. Therefore the ancestor is the equation that should be kept, and any equation which duplicates one of its ancestors may be discarded immediately. (Any derivation tree for E containing an equation D which is a duplicate of one of its ancestors, D′, can be transformed into another valid derivation tree for E by replacing the subtree rooted by D with the smaller subtree rooted by D′.)

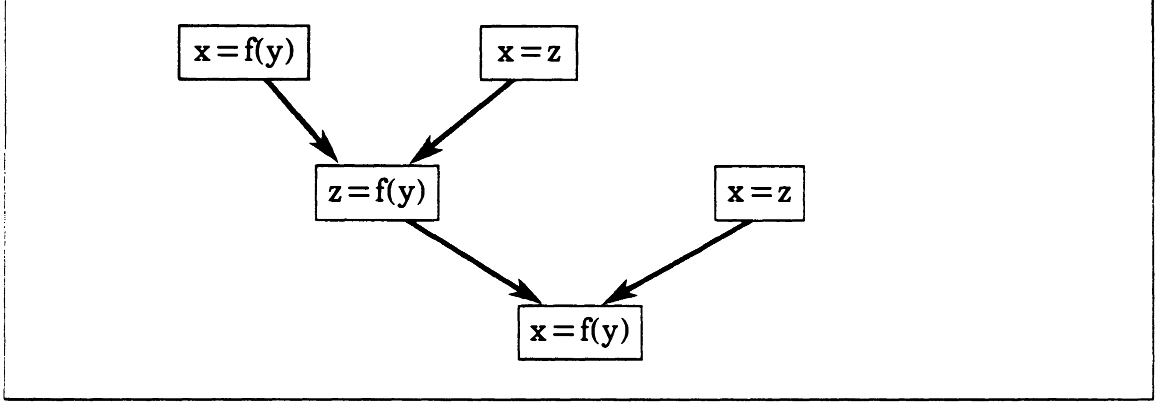


Figure 6.10. Duplicating an Ancestor Equation.

A somewhat more subtle restriction can be placed on the sites where substitution is allowed to occur. Specifically, it is sufficient to substitute along only one branch of each term. Consider the situation in Figure 6.11, where the central term is one side of a particular equation. There is another equation y = h(x), not conflicting with the equation holding the central term, which gives a possible replacement for the y's in the central term. The central term already has one substitution site, where h(f(z,y)) was substituted for w. Substituting on only one branch means that the y in h(f(z,y)) may be substituted for, but the one in g(y) may not. The correctness of this restriction is based on each branch's being counted separately. Substituting into h(f(z,y)) gives a new, longer branch which could not be created without that substitution. Substituting into g(y) also gives a new, longer branch, but that same branch could be created by substituting into the earlier version, f(g(y),w), of the central term. The equations with the terms f(g(h(x)),h(f(z,y)))

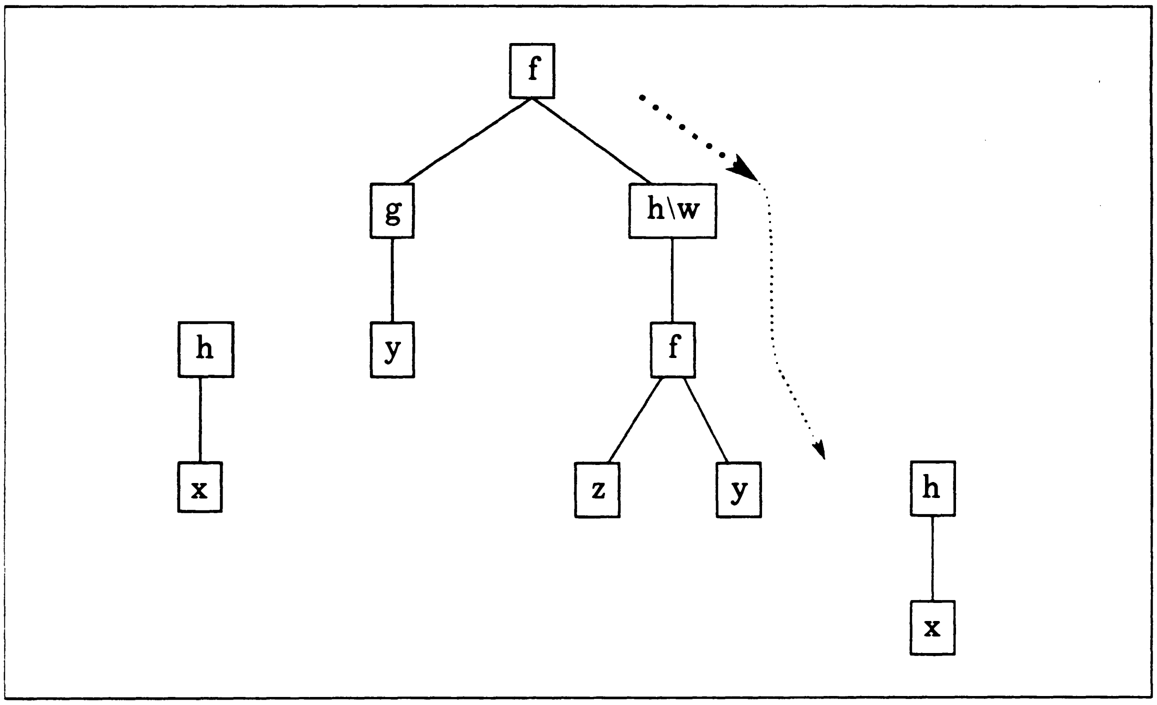


Figure 6.11. Substituting on Only One Branch.

and f(g(h(x)),w) will not be independent of each other, so there is no need to keep both equations as support for the f(g(h(*)),*) branch. More formally:

Theorem 6.5.1. Every derivation tree supporting a particular branch can be pruned to give a similar derivation tree supporting that branch with no substitutions on other branches of the term.

Proof: Assume we have an equation $x=t$ with derivation tree T. Let b be the branch of t in which we are interested. Let s be a node in the derivation tree where a substitution in t but not on b was made by substituting equation s_1 into equation s_2. (See Figure 6.12 for a diagram of the situation.) Further, select s so that no other node meeting these conditions is on the path from s to the root of T. Replacing the subtree of the derivation tree which is rooted by s with the tree rooted by s_2 and backing out the substitution in the equations between s_2 and the root of the derivation tree yields a new derivation tree T′ for the equation $x=t'$, where t′ is t with one off-b substitution removed.

T′ can be shown to be a valid derivation tree by induction on the number of equations between s_2 and the root of the tree. Each step in the induction con-

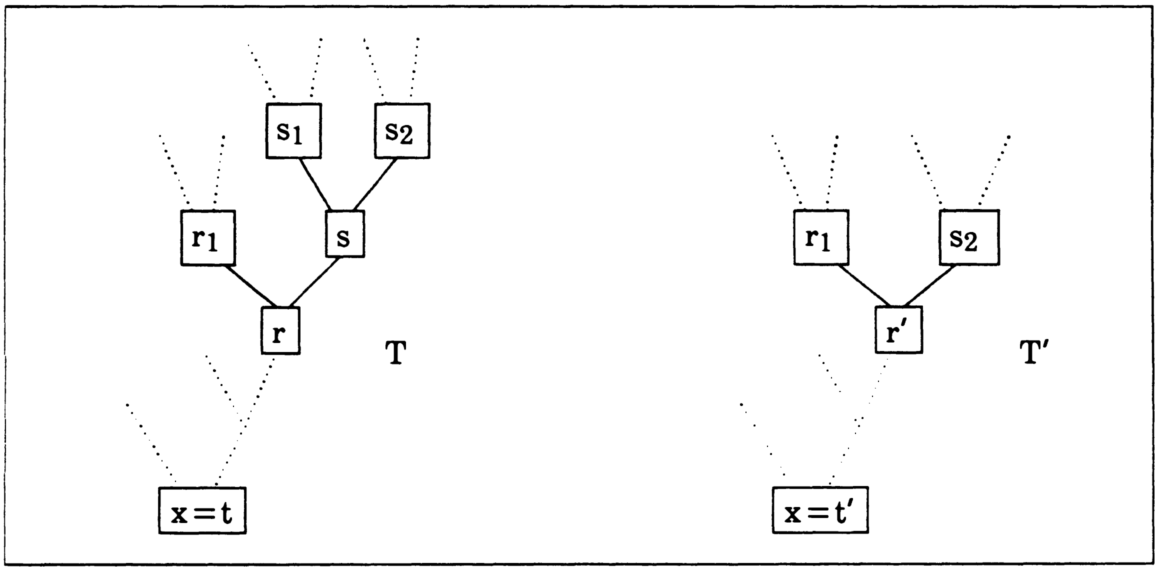


Figure 6.12. Pruning Substitutions from a Derivation Tree.

sists of proving that a similar child equation without the extra substitution can be created once a parent equation without that extra substitution exists. Suppose the equation r, the child of s in T, was created by substitution as in Figure 6.12. The derivation trees rooted by s_2 and r_1 are known to be valid and nonconflicting. Due to the selection of s, r_1 does not try to substitute into the term previously added by s_1, so r_1 still has the same place available for substitution in s_2, and this substitution creates the similar equation r′ without the extra off-b substitution that was in r.

Alternatively, suppose r was created by equating the arguments of s. Since substituting for a side consisting of a variable cannot qualify as an off-b substitution, s_1 must have been substituted into a functional term of s_2 to give an equation (s) where both sides were functional terms; therefore both sides of s_2 must already have been functional terms. Thus s_2's arguments may be equated, creating the similar equation r′ without the extra off-b substitution that was in r. □

Another simple restriction involves selecting a normal order for substitution applications. Consider the situation in Figure 6.13, where the same three input equations produce identical equations by means of different derivation trees. The two x=f(g(h(w))) equations together give no more

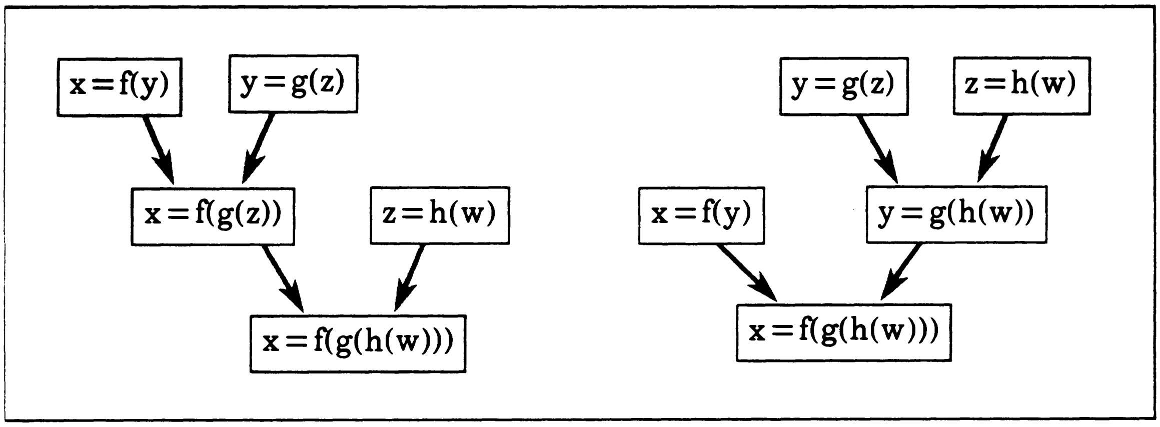


Figure 6.13. Normal Form for Substitutions.

constraints, support for constraints, or distinct opportunities for further rule application than either one alone. Therefore we add the restriction that we cannot substitute a term that already contains substitutions, with the understanding that there will always be a normal ordering of substitutions to generate each equation prevented by this restriction. Following this ordering, the first $x = f(g(h(w)))$ equation and the $y = g(h(w))$ equation will be formed, but the second $x = f(g(h(w)))$ equation will not be.

In addition to this normal ordering on substitutions alone, we can define a normal ordering on combinations of substitution and equating arguments. Figure 6.14 shows two derivations, both again using the same starting equations, of the equation $u_i = g(\ldots)$. Since the lower derivation shows an explicit $u_i = y$ constraint not found in the upper while the upper derivation has no additional explicit constraints, we add the restriction that we cannot substitute into an equation which is ready for equating arguments. Following this ordering, the two derivations represented by dotted arrows in Figure 6.14 will not occur.

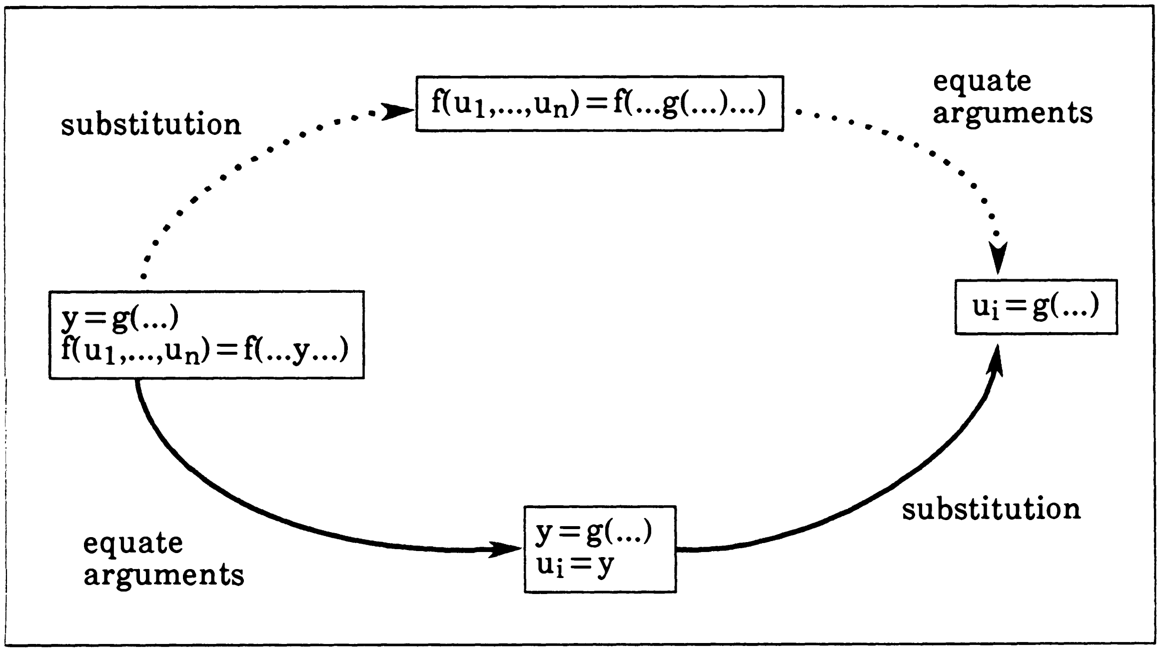


Figure 6.14. Delaying Substitutions.

While each of these restrictions looks like it would save only one or two equations, the cumulative effect is quite impressive. Merely following the single-substitution rules on the input set { f(y,y) = f(g(z),x), w = f(y,x), y = g(z), y = g(z), y = g(z) } resulted in 435 equations appearing in the union of derivation sequences (54 of them were promptly discarded as being identities, so the true number resulting from blind rule application is even higher). Following the restrictions discussed in this section resulted in only 40 equations appearing in the union, yet these 40 equations were guaranteed to give the same answer ({ w = f(g(z)[1],g(z)[1])[1] , y = g(z)[4] , x = g(z)[1] }) as the 435 earlier equations.

Chapter Seven

Counting Recursive Types

The previous chapter developed the ideas and formalism used to define unique, order-independent, weighted solutions to sets of conflicting type equations. Implicit in that development was the assumption that there would be no recursive type constraints in the input equations. The single-substitution rule,

> Given x = u, where x does not occur in u but x does occur in some other equation, replace one occurrence of x in another equation by u and remove the x = u,

refuses to consider a recursive type equation for substitution. In some applications, finding x occurring in u calls for an error indication and often an immediate halt of the unification process. However, for generality as well as the ability to type self-application of functions, we need to allow the substitution rule to use recursive terms.

7.1. Rule-Based Countable Unification with Recursive Types

Section 4.4 introduced the notation $f_1(@1)$ to represent the infinite regular tree f(f(f(...))). Using this notation, we can transform the implicitly recursive equation x = f(x) into an explicitly recursive form, $x = f_1(@1)$. Similarly, x = g(x,y) can be rewritten as $x = g_2(@2,y)$ and x = f(g(x),h(x)) can become $x = f_3(g(@3),h(@3))$. In Chapter Four, this sort of transformation was included in the substitution rule itself, which was written as

> If there is an equation x = u, u is not x, and x appears in some other equation:

i) If x occurs in u then, since u is not x, u must be a function. Subscript the root symbol of u with a new label and replace occurrences of x in u by that label.

ii) Replace occurrences of x in other equations by the possibly modified u.

The transformation was done once, when an equation was used with the substitution rule, and the explicitly recursive form then replaced all other occurrences of the appropriate variable. The single-substitution rule as modified to allow for creating the union of derivation sequences, in contrast, is applied to an equation $x=u$ once for each other nonconflicting equation containing x. Therefore it is more efficient to move the transformation to explicitly recursive form into a separate rule, whose result will be available for substitution.

Once we have such a rule to create explicitly recursive terms, the single-substitution and equate-arguments rules require minor adjustment. In the single-substitution rule, we merely replace "x does not occur in u" by "x is not u". In the equate-arguments rule, we will have to unroll recursive terms as it becomes necessary. As an example, equating the arguments of $f_1(g(@1))=f(g(z))$ should yield $g(f_1(g(@1)))=g(z)$ and eventually $f_1(g(@1))=z$. The @1 in such an equation must not lose its reference as arguments are equated. This "unrolling if necessary" function is denoted by ρ, so that in this example $\rho(g(@1))$ is $g(f_1(g(@1)))$ while $\rho(g(z))$ is simply $g(z)$.

Making these adjustments gives us the following final set of unification rules, which will give us the union of possible derivation sequences from any set of equations, whether or not they imply recursive types:

(1) Given an equation $x=x$, mark the equation as deleted.

(2) Given $t=x$ which has not been reversed, add $x=t$ with creators Reversal and the parent equation; record that the parent has undergone Reversal.

(3) Given $f(u_1,...,u_n)=f(u_1',...,u_n')$ which has not had its arguments equated, add $\rho(u_1)=\rho(u_1')$, ..., $\rho(u_n)=\rho(u_n')$ with creators EquateArgs or UnrollEquateArgs and the parent equation; record that the parent has undergone EquateArgs. ($\rho(u_i)=\rho(u_i')$ gets UnrollEquateArgs as a creator if at least one side of the equation had to be unrolled; otherwise it gets EquateArgs.)

(4) Given $x=u$ where x is not u, and $v=w$ where x occurs in v or w, where the parents do not conflict and $x=u$ has not been substituted into $v=w$, for each occurrence of x in $v=w$, add a new equation with that single occurrence replaced by u with creators Subst and the parent equations; record that $x=u$ has been substituted into $v=w$.

(5) Given $x=u$ where x is not u but x occurs in u and the equation has not been made recursive, create the explicitly recursive term $\hat{u}$ from u by subscripting the root function of u with a new label and replacing occurrences of x in u by that label; add $x=\hat{u}$ with creator MakeRecurs and the parent equation; record that the parent has undergone MakeRecurs.

Figure 7.1 helps to explain the distinction between an equation created by EquateArgs and one created by UnrollEquateArgs. As in the previous chapter, an equation created by EquateArgs depends on only one argument position of its parent equation. An equation created by UnrollEquateArgs, however, contains at least one copy of an entire side of the parent equation, so it must be said to depend upon the entire parent. Depending on the entire

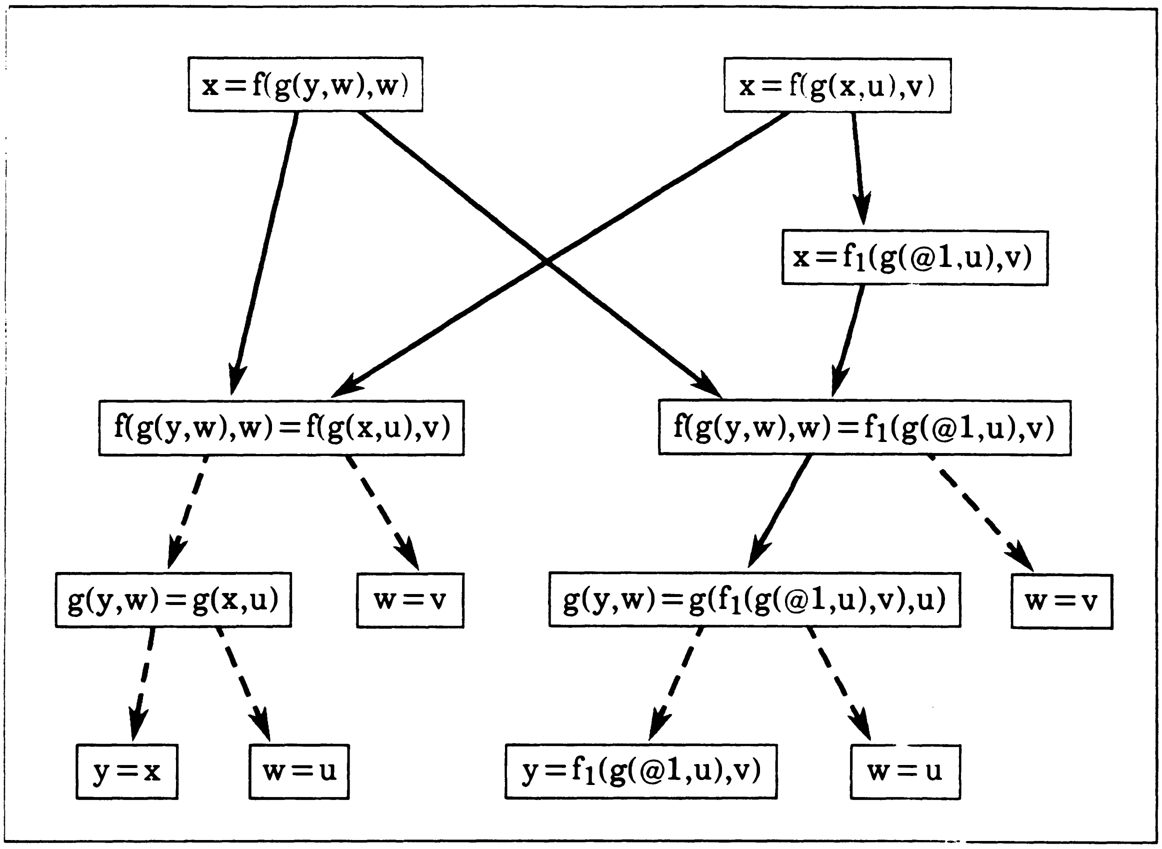


Figure 7.1. Comparing Implicit and Explicit Recursive Forms.

parent is indicated by a solid derivation line in the figure; depending on only one argument position is indicated by a dashed line.

This figure also illustrates the different behavior of implicit and explicit recursion. The derivations on the left ignored the signs of implicit recursion in the top right equation and thus produced equations with no indication of recursion. Offspring of different argument positions, such as w=u and w=v, were always independent. However, these derivations could not meaningfully constrain y. The derivations on the right produced explicit recursion which allowed a nontrivial constraint on y to appear, but w=u and w=v are no longer independent. Since both kinds of behavior produce additional useful constraints, both implicit and explicit recursive types are tolerated in the first phase of the unification process, which consists of finding the union of derivation sequences. The second phase, however, will always produce an explicitly recursive type as the solution for any variable with a recursive nature.

7.2. Implementation of Countable Recursive Types

As in the nonrecursive case, the formal unification rules with their nondeterministic guards are not particularly convenient for sequential implementation. Again we can simplify the rule guards considerably by representing the set of equations as a list and accepting a deterministic order for rule application. In fact, assuming that the "equate the arguments" subroutine is extended to unroll arguments as necessary, we need only add one line, corresponding to the newly added rule which creates explicitly recursive terms, to the earlier sketch of the algorithm for nonrecursive countable unifi-

```
while equations are being added do
    thiseqn ← the head of the list
    while thiseqn exists do
        if thiseqn's lasteqn is null then
            try to create an explicitly recursive form of thiseqn
            try to equate the arguments of thiseqn
            thateqn ← the head of the list
        else
            thateqn ← the successor of thiseqn's lasteqn
        mark thiseqn's derivation tree with its equation number
        while thateqn exists do
            if thateqn's derivation tree does not conflict,
                try to substitute thiseqn into thateqn
            thateqn ← the successor of thateqn
        clear the marks from thiseqn's derivation tree
        thiseqn's lasteqn ← the last existing equation
        thiseqn ← the successor of thiseqn
```

Figure 7.2. Implementing Countable Recursive Unification.

cation. Since we want to try to make a recursive form of each equation exactly once, just as we wanted to try to equate the arguments of each equation exactly once, both rule applications can be attempted under the same condition, namely that the equation has never been examined before. The updated algorithm sketch appears in Figure 7.2.

The result of using this algorithm on an implicitly recursive set of input equations can be found in Figure 7.3. The implementation denotes a function subscript by prefixing the function symbol with the subscript in angle brackets and references a subscripted term by prefixing the subscript with 'v' (for 'variable'). Thus, <256> g is alternate notation for g_{256} and v256 is alternate notation for @256. All the other notation is the same as in the nonrecursive case. (As a quirk of the implementation, generated subscripts begin with 256 to avoid potential conflicts with variables in trial input lists.)

7.3. Further Restricting the Set of Generated Equations

Most of the means of restricting the set of generated equations that were discussed in the previous chapter remain equally valid in the recursive case. Removing identities, substituting on only one branch, and delaying

```
Equation list:
1: x = f(y)
2: y = g(x)
3: g(f\x(y)) = y [[ 1, 2]]
4: f(g\y(x)) = x [[ 2, 1]]
5: y = <256> g(f\x(v256)) [[ 3]]
6: x = <257> f(g\y(v257)) [[ 4]]

Equation list:
7: x = <257> f(g(v257[1])[1])[1]
8: y = <256> g(f(v256[1])[1])[1]
```

Figure 7.3. An Example of Counting Unification with Recursion.

substitution until after arguments have been equated all work exactly as before.

Avoiding the creation of duplicates of ancestor equations also works exactly as before, but its use becomes imperative in the recursive case. Instead of merely being inefficient, recreating ancestors is now a potential source of nontermination. The equation $f_1(g(@1)) = f(g_2(f(@2)))$, for example, yields $g(f_1(g(@1))) = g_2(f(@2))$ when its arguments are equated. Equating the arguments of this second equation recreates the first equation. Without a check for duplicated ancestors, this process would continue indefinitely.

The idea of not substituting a term that already contains substitutions remains valid, with the exception of terms that are explicitly recursive at their roots. Consider the situation in Figure 7.4. Unless we are allowed to substitute the f_3 term into the z equation, we will never be able to determine that z has a recursive character. This exception is required because we can only detect that a term is implicitly recursive when it is at the top level, *i.e.* when it is the u of an x=u equation; lower-level recursion can be found only by substituting explicitly recursive terms.

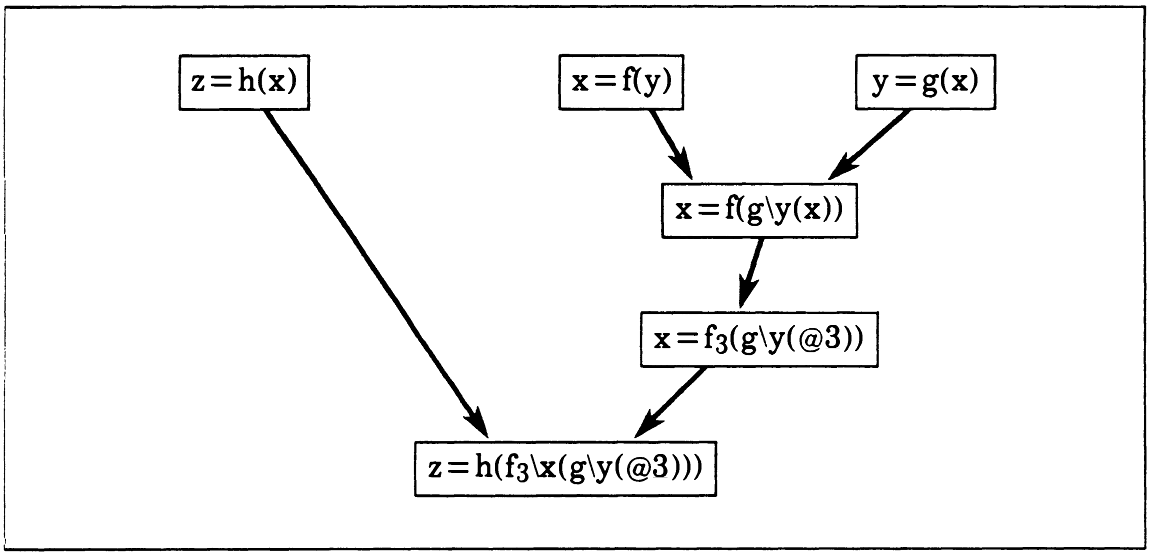


Figure 7.4. Substituting Recursive Forms.

A new restriction for the recursive case involves substituting for the same variable only once on a branch. The opportunity to substitute for a variable more than once on the same branch can arise only when that variable is recursive, since both situations require a variable to appear within a term which is equated to itself. Each occurrence of a variable must be equivalent to the same regular tree in a nonconflicting solution. Thus, subsequent occurrences of variables on a branch can be considered more as implicit recursion markers than as substitution loci. As an illustration, consider the situation in Figure 7.5. We have an equation with the large term, in which h(f(z,y)) has already been substituted for y, appearing as one side and also a second nonconflicting equation y=h(x) ready to use for substitution. We do not want to substitute h(x) for the y in h(f(z,y)) in the large term, because that would be a second substitution for y on the same branch. The side f(g(w),y) in the ancestral equation is, however, open for substitution to become either

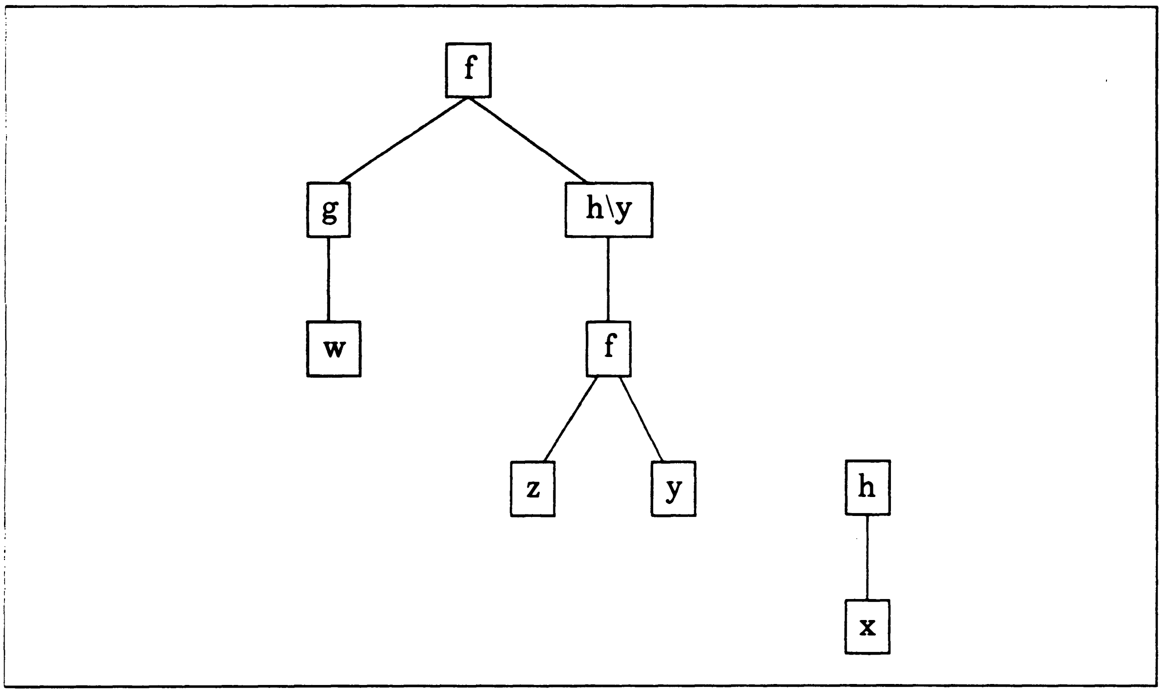


Figure 7.5. Substituting for the Same Variable Once per Branch.

$f(g(w),h(x))$ (using the $y=h(x)$ equation) or $f(g(w),h_4(f(z,@4)))$ (using the explicitly recursive form of the ancestral $y=h(f(z,y))$ equation).

As a matter of principle, we want all potential conflicts on y to be apparent at the highest possible level instead of being spread out over several levels. To achieve this, we fold all substitutions for a variable back to the highest level on which the variable occurred on that particular branch. The reasoning behind folding conflicts to the highest level is perhaps more readily apparent if the second equation in the example is changed from $y=h(x)$ to $y=g(x)$. Since there will now be a conflict on y due to the equations $y=h(f(z,y))$ and $y=g(x)$, this conflict should appear in the context of the larger term as $f(g(w), h_4(f(z,@4)) \mid g(x))$ instead of $f(g(w), h_4(f(z,@4 \mid g(x))) \mid g(x))$ as it would appear without this restriction. The process of constructing the second form, in fact, could be interpreted as first claiming that y was an $h(*)$, and, after accepting that claim, then claiming that y was a $g(*)$. By accepting the once-per-branch restriction, we can maintain the idea that each branch option represents one internally consistent alternative.

7.4. Presentation of the Final Answer

Once we accept recursive solutions with infinite regular trees as types, we no longer automatically have a single representation for our types. The regular tree represented by $f_1(@1)$ is "the same" as that represented by $f_2(f(@2))$ or $f_3(f(f(@3)))$. Each of these regular trees can be considered as a finite state machine with each function or variable occurrence becoming a state and the various argument indices providing the transitions between states. For each such finite state machine, there is a minimal equivalent finite state machine which can be found by algorithms such as those in [HoU79]. The minimal finite state machine can then be rewritten as a minimal equivalent regular tree. The minimal equivalent types for the three

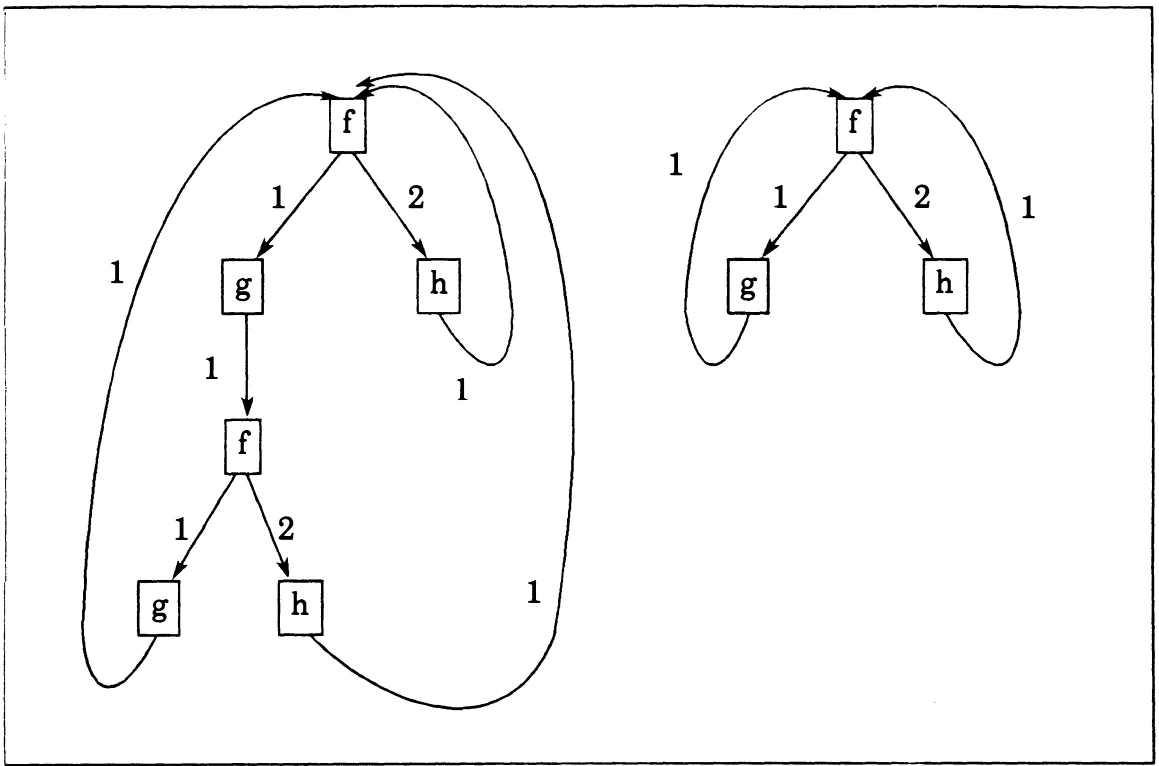


Figure 7.6. Recursive Types as Finite State Machines.

regular trees in question are all of the form $f_1(@1)$. As another example of this transformation, Figure 7.6 shows the immediate and minimal finite state machines for the term $f_4(g(f(g(@4),h(@4))),h(@4))$.

As long as the solution is consistent, we would like to get the solution in minimal form, since that is the shortest and easiest to comprehend. To allow this, we perform a preliminary pass over the set of generated equations before the main part of the second, or counting, phase of the unification process. For each explicitly recursive equation which is not in minimal form, this pass adds a minimal-form equivalent, with creators MinRecurs and the nonminimal-form equation. Figure 7.7 shows the behavior of the implementation when given two consistent recursive input equations. The two added minimal form equations are shown between the equation lists resulting from the two main phases. Since there is no conflict, only a minimal-form equation

```
Equation list:
 1: x = f(f(x))
 2: x = f(x)
 3: x = <256> f(f(v256)) [[ 1]]
 4: f\x(f(x)) = f(x) [[ 1, 2]]
 5: f(f\x(f(x))) = x [[ 1, 2]]
 6: x = <257> f(v257) [[ 2]]
 7: f\x(x) = f(f(x)) [[ 2, 1]]
 8: f(f(f\x(x))) = x [[ 2, 1]]
 9: f\x(x) = <258> f(f(v258)) [[ 2, 3]]
10: <259> f\x(f(v259)) = f(x) [[ 3, 2]]
11: f( <260> f\x(f(v260))) = x [[ 3, 2]]
12: <261> f\x(f(v261)) = <262> f(v262) [[ 3, 6]]
14: <263> f\x(v263) = f(f(x)) [[ 6, 1]]
15: f(f( <264> f\x(v264))) = x [[ 6, 1]]
16: <265> f\x(v265) = <266> f(f(v266)) [[ 6, 3]]
18: x = f( <267> f(f(v267))) [[ 9]]
19: f( <268> f\x(f(v268))) = x [[10]]
20: f( <269> f\x(f(v269))) = <270> f(v270) [[12]]
21: <271> f\x(v271) = f(x) [[14]]
22: <272> f\x(v272) = f( <273> f(f(v273))) [[16]]
26: f\x(f(x)) = <279> f(v279) [[ 1, 6]]
27: f(x) = <280> f(v280) [[26]]

29: x = <282> f(v282) [[3]]
30: x = <283> f(v283) [[18]]

Equation list:
31: x = <256> f(v256[2])[2]
```

Figure 7.7. Minimizing Recursive Forms.

indicating that x is an f(*) with weight 2 and, within that term, is recursive with weight 2 is returned as the final answer.

In a situation with conflicts, however, the user may be interested in knowing at just what level recursion was implied. Given the equations x=f(f(f(x))) and x=f(f(g(y))), for example, the implementation returns the answer x= ⟨260⟩ f(v260[1] | f(f(v260[1])[1] | g(y)[1])[2])[2], which indicates both that there was an attempt to make x recursive which reduced to $x=f_{260}$(@260) and that the recursive attempt was three levels deep. Returning options for both the immediate and minimal forms of conflicting recursive types gives users the opportunity to analyze or ignore these forms as they see fit. If a given application knows that it will only be interested in immediate or minimal forms, it is a simple matter to not create the minimal forms at all or to use the minimal forms to replace instead of augment the immediate forms.

Chapter Eight

Conclusions

The preceding chapters have developed two major expansions to the common approaches to attribute grammars and unification algorithms. Gated attribute grammars provide a much more general way of dealing with circularity in attribute dependency graphs, and this generality allows programming environments that could previously incrementally analyze programs to incrementally execute them as well. Counting unification provides a rigorous way of dealing with compile-time type errors. By examining all the type information derivable from an incorrectly typed program, counting unification provides a much better overall picture of type usage in the program, and this overall picture then allows programming environments and compilers to give the user reasonable suggestions about the sources of the type errors.

8.1. Gated Attribute Grammars and Environments

Traditional approaches to attribute grammars ([Knu68], [RTD83]) have required that the attribute dependency graph for every parse tree derivable from the grammar be acyclic. To avoid unnecessary reevaluation, attribute instances from any parse tree are evaluated in accordance with the topological order imposed by the dependency graph. Some more recent approaches ([JoS86], [Far86]) allow restricted cycles, but these approaches require every evaluation function appearing in a cycle's strongly connected component to be monotonic, in addition to other conditions that guarantee termination of the attribute evaluation process. In these approaches, SCCs are scheduled for evaluation in accordance with the topological order imposed by a collapsed dependency graph where each SCC becomes a single node. Attribute instances within an SCC are evaluated as they are encountered by a depth-first search starting from points of change. These depth-first searches are repeated until the attributes of the SCC reach their guaranteed least fixed point.

Gated attribute grammars require only that every nontrivial SCC contain a gate attribute. Gated strongly connected components are again scheduled in accordance with the collapsed dependency graph, but the evaluation of attributes of a GSCC alternates between the gate attribute and the other attributes, which are scheduled in accordance with the internal topological order of the GSCC (as revealed by removing links leading to the gate attribute). Automatically added nonlocal dependencies link predecessors of GSCC attributes to the start attribute associated with the GSCC's gate attribute. These nonlocal links tell the start attribute when any of the GSCC's predecessors change value, and thus allow the start attribute to know whether a GSCC is just beginning evaluation or whether it is iterating. If evaluation has just reached a GSCC, its gate attribute takes a value depending on predecessors outside the GSCC; if evaluation is iterating over the GSCC, the gate takes a value depending on attributes inside the GSCC. The nonlocal links to start attributes, combined with the order of evaluation within a GSCC, prevent a GSCC from incrementally reaching different fixed points from the same predecessor values, which would otherwise be a danger when nonmonotonic functions are in the GSCC.

Because the dependency functions in a GSCC are no longer restricted to ensure termination, a wider variety of information can be computed by the attribute grammar. For example, an attribute grammar can be used to specify an interpreter which incrementally updates the output of the program as the program itself is modified. Since the halting problem is undecidable for any reasonable programming language, the earlier approaches to attribute grammars do not permit such interpretation via attributes. Therefore, the stronger gated attribute grammars can provide a rigorous new declarative way of handling incremental evaluation. Gated attribute grammars in general make run-time semantics available in the same framework that has been used so successfully for incremental compile-time semantics.

8.2. Counting Unification and Environments

Standard unification algorithms either halt when they detect an inconsistency in their input or reject that portion of the input which they believe to be incorrect. Unfortunately, these algorithms do not make helpful choices about which portion of the input is incorrect. The most common approach will believe whichever constraint is encountered first. If all the other constraints participating in a conflict agree with each other, however, it is that first constraint which should be suspected of being incorrect. Under the common approach, there is no way of even knowing whether all the "incorrect" constraints imply the same type. The expanded unification algorithms discussed here provide a formal, rigorous, and order-independent definition of the number of times the input equations imply each of the conflicting options. By comparing the relative strengths of the options, an application using one of these algorithms can make a reasoned judgment as to which options are likely to be incorrect. The maximum-flow algorithms from Chapter Five can be simplified to quickly determine all the various conflicting types associated with a variable. The single-substitution algorithms from Chapters Six and Seven give answers that agree with intuition in recursive as well as nonrecursive cases and can be modified for different styles of computing environments.

As an example of an application using a single-substitution algorithm, consider an interactive programming environment for a simple ML-like language which includes terms like **fun** x . **if** x **then** 3 **else** x. Following the type constraints for such a language from Section 4.2, we notice that as the guard of the **if** expression, x must be a boolean but as the body of the **else** expression, x must be of the same type as 3. These constraints are shown in the first section of Figure 8.1. Since the programming environment wants to highlight pieces of syntax contributing to a type error, it needs some way to map the results of counting unification back to the parse tree. To do this mapping, a new type variable is introduced whenever the syntax imposes another type constraint. In this example, guard1 is the variable which coordinates the guard constraint and thenelse1 the one which coordinates the types of the **then** and **else** expressions. After the counting unification algorithm has determined that each of these variables is implied to be an integer once and a boolean once, the environment can fish out the final constraints on the introduced type variables and check them for conflicts. Since the guard constraint (as represented by guard1) participated in a conflict, the **if** syntax can be highlighted to show this. The intensity of the highlighting depends on

```
Collected from parse tree, the equation list is:
    typex = guard1
    bool() = guard1
    int() = thenelse1
    typex = thenelse1
After counting unification, the equation list is:
    guard1 = int()[1] | bool()[1]
    thenelse1 = bool()[1] | int()[1]
    typex = int()[1] | bool()[1]
```

Figure 8.1. Using Counting Unification on a Program.

the relative weight of the constraint option (bool() in this case) imposed at that point. Support of a clear minority opinion would be highlighted much more strongly than a majority opinion. In this case, the **if** and the **then-else** expressions would be highlighted with the same intensity to show the reasons for the conflict over x's type.

This usage of introduced variables to map back to the syntax imposing the constraints can be avoided if the original equations are tagged with the locations in the program (or parse tree) which imposed them. This tagging approach would create only two original equations, typex=bool() and typex=int(), from the sample expression. After counting unification had determined that the result was typex=int()[1] | bool()[1], the ancestry of each equation supporting one of these options could be traced and the location of each ancestral original equation highlighted with the appropriate intensity.

Relatively complex algorithms used in inherently incremental applications such as language-based editors ought to be incrementally computable themselves. The single-substitution counting unification algorithm is composed of a relatively slow first phase (equation generation) and a much faster second phase (equation counting). It is not obvious how to do incremental equation counting, but it is fortunately very easy to incrementally update the results of the first phase, which will give most of the benefits of making the entire algorithm incremental. Assume that between two invocations of the counting unification algorithm some input equations (D) have been deleted while others (A) have been added to the input set. Even without the benefit of any additional data structures, all generated equations with ancestors in D can be removed in a single pass over the previous first-phase results. (This discussion presupposes the implementation decisions outlined in Section 6.4.) All equations which can be generated from the remaining input equations remain in the resulting equation list. Once the equations in A are added to the end of this equation list, equation generation can start again, but, thanks to the leftover lasteqn pointers, all newly generated equations will have ancestors in A and thus could not have been in the previous first-phase results. Therefore we have very simply removed exactly those equations which needed to be removed from the previous first-phase results and added in exactly those equations which needed to be added, so the incremental update is, in this sense, optimal.

8.3. Further Work in Counting Unification

One interesting possibility for further investigation involves developing parallel versions of the single-substitution unification algorithms for use with large software systems. Examining the output of the first phase shows that even when the total number of generated equations reaches several hundred, any given equation will have no more than eight or ten ancestors. This observation, coupled with the fact that the order in which equations are generated is unimportant, suggests that a parallel implementation could achieve significant speedup compared to the sequential one. As usual, the problems would involve either sequencing updates to a shared linked list of equations (retaining the structure from the sequential implementation) or developing ways to distribute the equation data structure among the processors. A compromise between the complexity of rule guards seen in the formal unification rules and the inflexibility of application order seen in the sequential implementation may be appropriate for the parallel case.

In the second phase of the counting unification process, it would be easy to assign the counting of branches for different variables to different processors and slightly more complicated to assign the counting of different branches for the same variable to different processors. Both cases are conceptually simple because the counting of each branch is independent of the existence or weight of any other branches. However, in the second case, the different branches would need to be merged together (and have consistent recursive forms cleaned up), whereas in the first case each processor could simply report its result at any time.

Another possibility for further investigation involves adapting the counting unification approach to other kinds of languages. Traditional strongly-typed procedural languages such as Algol, Pascal, and Ada do not provide the same sort of clean type constraints as ML does. The primary reason, of course, is that the type systems of these languages are not based on the general polymorphism that underlies ML and the entire counting unification development. A secondary reason is that these languages were designed more to allow efficient type checking proceeding from declarations than to allow type inference in the absence of declarations. ML, on the other hand, is designed for type inference and performs type checking by comparing inferred types against the optional types in declarations. In spite of these problems, it should be possible to let the syntax for any given procedural language provide some type constraints which can be used in the polymorphic framework. An editor for such a language is then free to consider it an error if unification does not provide a sufficiently monomorphic solution to fit into its type system. *Ad hoc* polymorphism such as overloading is likely to require *ad hoc* pre- or post-processing of the type constraints. (It is tempting, for example, to regard overloaded functions that take either integer or real arguments as taking a general numeric type for the purposes of unification, and then to straighten out the exact flavors of numeric types at a later stage.)

Whether an application uses the exact counts derived from the theory in Chapters Six and Seven, relies on approximations as in Chapter Five, or requires extra help to use either of these frameworks, a solution based on the overall pattern of type constraints induced by a program will undoubtedly provide far better data to guide error correction strategies than can be produced by the usual algorithms found in compilers and programming language environments.

Bibliography

[ASU86] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA (1986). (Chapter 10).

[BaS86] Bahlke, R. and G. Snelting, "The PSG System: From Formal Language Definitions to Interactive Programming Environments," *ACM Transactions on Programming Languages and Systems* 8(4), pp. 547-576 (October 1986).

[BMS87] Bahlke, Rolf, Bernhard Moritz, and Gregor Snelting, "A Generator for Language-Specific Debugging Systems," *Proceedings of the ACM SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notices* 22(7), pp. 92-101 (June 1987).

[Bro86] Broy, Manfred, "An Assessment of Programming Styles: Assignment-oriented Languages versus Functional and Applicative Languages," *Fundamenta Informaticae* 9, pp. 169-204 (1986).

[Car83] Cardelli, Luca, "ML under Unix," *Polymorphism* 1(3) (December 1983).

[Car85] Cardelli, Luca, "Basic polymorphic typechecking," *Polymorphism* 2(1) (January 1985).

[CKv83] Colmerauer, Alain, Henry Kanoui, and Michel van Caneghem, "Prolog, theoretical principles and current trends," *Technology and Science of Informatics* 2(4), pp. 255-292 (1983).

[Col82] Colmerauer, Alain, "Prolog and Infinite Trees," in K. L. Clark and S.-A. Tärnlund, *eds.*, *Logic Programming*, Academic Press, London (1982). (pp.231-251).

[DaM82] Damas, Luis and Robin Milner, "Principal type-schemes for functional programs," *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, pp. 207-212 (January 1982).

[dCh86] de Champeaux, Dennis, "About the Paterson-Wegman Linear Unification Algorithm," *Journal of Computer and System Sciences* **32**, pp. 79-90 (1986).

[End72] Enderton, Herbert B., *A Mathematical Introduction to Logic*, Academic Press, New York (1972).

[Far86] Farrow, Rodney, "Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars," *Proceedings of the ACM SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Notices* 21(7), pp. 85-98 (July 1986).

[FJM83] Fischer, C.N., G.F. Johnson, J. Mauney, A. Pal, and D.L. Stock, "An Introduction to Editor Allan POE," *Proceedings of Softfair, A Conference on Software Development Tools, Techniques, and Alternatives* (July 1983).

[FJM84] Fischer, C.N., G.F. Johnson, J. Mauney, A. Pal, and D.L. Stock, "The POE Language-Based Editor Project," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices* 19(5), pp. 21-29 (May 1984).

[FMM79] Fischer, Charles N., Donn R. Milton, and Jon Mauney, "A Locally least-cost LL(1) error corrector," Technical Report #371, University of Wisconsin (August 1979).

[GHJ79] Graham, Susan L., Charles B. Haley, and William N. Joy, "Practical LR error recovery," *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices* 14(8), pp. 168-175 (1979).

[HeS83] Henhapl, W. and G. Snelting, *Context Relations -- a concept for incremental context analysis in program fragments*, Technical Report PU1R8/83, Technische Hochschule Darmstadt (August 1983).

[HoU79] Hopcroft, John E. and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading MA (1979). (Section 3.4).

[Hoo86] Hoover, Roger, "Dynamically Bypassing Copy Rule Chains in Attribute Grammars," *Proceedings of the Thirteenth ACM Symposium on Principles of Programming Languages*, pp. 14-25 (January 1986).

[Hoo87] Hoover, Roger, *Incremental Graph Evaluation*, Ph.D. thesis, Technical Report 87-836, Cornell University (May 1987).

[JeW78] Jensen, Kathleen and Niklaus Wirth, *Pascal User Manual and Report, Second Edition*, Springer-Verlag, New York (1978).

[JoF85] Johnson, Gregory F. and C. N. Fischer, "A Meta-Language and System for Nonlocal Incremental Attribute Evaluation in Language-Based Editors," *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pp. 141-151 (January 1985).

[JoF87] Johnson, Gregory F. and Charles N. Fischer, "Nonlocal Attribute Grammars and Incremental Attribute Evaluation," submitted to *ACM Transactions on Programming Languages and Systems*, December 1987.

[JoW86] Johnson, Gregory F. and Janet A. Walz, "A Maximum Flow Approach to Anomaly Isolation in Unification-Based Incremental

Type Inference," *Proceedings of the Thirteenth ACM Symposium on Principles of Programming Languages*, pp. 44-57 (January 1986).

[JoS86] Jones, Larry G. and Janos Simon, "Hierarchical VLSI Design Systems Based on Attribute Grammars," *Proceedings of the Thirteenth ACM Symposium on Principles of Programming Languages*, pp. 58-69 (January 1986).

[KaW87] Karinthi, Raghu R. and Mark Weiser, "Incremental Re-Execution of Programs," *Proceedings of the ACM SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques*, *SIGPLAN Notices* **22**(7), pp. 38-44 (June 1987).

[Kas80] Kastens, Uwe, "Ordered Attributed Grammars," *Acta Informatica* **13**(3), pp. 229-256 (1980).

[Knu68] Knuth, Donald E., "Semantics of Context-free Languages," *Mathematical Systems Theory* **2**(2), pp. 127-145 (June 1968). Correction **5**(1), pp. 95-96 (March 1971).

[Knu73] Knuth, Donald E., *The Art of Computer Programming: Volume 1, Fundamental Algorithms, Second Edition*, Addison-Wesley, Reading MA (1973). (Section 2.3.5, Exercise 11).

[MPS84] MacQueen, David, Gordon Plotkin, and Ravi Sethi, "An ideal model for recursive polymorphic types," *Proceedings of the Eleventh ACM Symposium on Principles of Programming Languages*, pp. 165-174 (January 1984).

[MaM82] Martelli, Alberto and Ugo Montanari, "An Efficient Unification Algorithm," *ACM Transactions on Programming Languages and Systems* 4(2), pp. 258-282 (April 1982).

[MaR84] Martelli, Alberto and Gianfranco Rossi, "Efficient Unification with Infinite Terms in Logic Programming," *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.202-209 (1984).

[Mee83] Meertens, Lambert, "Incremental Polymorphic Type Checking in B," *Proceedings of the Tenth ACM Symposium on Principles of Programming Languages*, pp. 265-275 (January 1983).

[Mil78] Milner, Robin, "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences* **17**(3), pp. 348-375 (December 1978).

[PaW78] Paterson, M. S. and M. N. Wegman, "Linear Unification," *Journal of Computer and System Sciences* **16**(2), pp. 158-167 (April 1978).

[ReT84] Reps, Thomas and Tim Teitelbaum, "The Synthesizer Generator," *Proceedings of the ACM Software Engineering Symposium on Practical Software Development Environments*, pp. 42-48 (April 1984).

[RMT86] Reps, Thomas, Carla Marceau, and Tim Teitelbaum, "Remote Attribute Updating for Language-Based Editors," *Proceedings of the Thirteenth ACM Symposium on Principles of Programming Languages*, pp. 1-13 (January 1986).

[RTD83] Reps, Thomas, Tim Teitelbaum, and Alan Demers, "Incremental Context-Dependent Analysis for Language-Based Editors," *ACM Transactions on Programming Languages and Systems* 5(3), pp. 449-477 (July 1983).

[Rob65] Robinson, J. A., "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the ACM* 12(1), pp. 23-41 (January 1965).

[Ske78] Skedzeleski, Stephen K., *Definition and Use of Attribute Reevaluation in Attributed Grammars*, Ph.D. thesis, University of Wisconsin (December 1978).

[Sne86] Snelting, Gregor, "Unification in Many-Sorted Algebras as a Device for Incremental Semantic Analysis," *Proceedings of the Thirteenth ACM Symposium on Principles of Programming Languages*, pp. 229-235 (January 1986).

[WaJ88] Walz, Janet and Gregory Johnson, "Incremental Evaluation for a General Class of Circular Attribute Grammars," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation* (June 1988).

[Wan86] Wand, Mitchell, "Finding the Source of Type Errors," *Proceedings of the Thirteenth ACM Symposium on Principles of Programming Languages*, pp. 38-43 (January 1986).