A Structure-Sharing Representation for Unification-Based Grammar Formalisms

Fernando C. N. Pereira
Artificial Intelligence Center, SRI International
and
Center for the Study of Language and Information
Stanford University

Abstract

This paper describes a structure-sharing method for the representation of complex phrase types in a parser for PATR-II, a unification-based grammar formalism.

In parsers for unification-based grammar formalisms, complex phrase types are derived by incremental refinement of the phrase types defined in grammar rules and lexical entries. In a naïve implementation, a new phrase type is built by copying older ones and then combining the copies according to the constraints stated in a grammar rule. The structure-sharing method was designed to eliminate most such copying; indeed, practical tests suggest that the use of this technique reduces parsing time by as much as 60%.

The present work is inspired by the structure-sharing method for theorem proving introduced by Boyer and Moore and on the variant of it that is used in some Prolog implementations.

1 Overview

In this paper I describe a method, *structure sharing*, for the representation of complex phrase types in a parser for PATR-II, a unification-based grammar formalism.

In parsers for unification-based grammar formalisms, complex phrase types are derived by incremental refinement of the phrase types defined in grammar rules and lexical entries. In a naïve implementation, a new phrase type is built by copying older ones and then combining the copies according to the constraints stated in a grammar rule. The structure-sharing method eliminates most such copying by representing updates to objects (phrase types) separately from the objects themselves.

The present work is inspired by the structure-sharing method for theorem proving introduced by Boyer and Moore [1] and on the variant of it that is used in some Prolog implementations [9].

This research, made possible in part by a gift from the Systems Development Foundation, was also supported by the Defense Advanced Research Projects Agency under Contracts N00039-80-C-0575 and N00039-84-C-0524 with the Naval Electronic Systems Command. The views and conclusions contained in this document are those of the author and should not be interpreted as representative of the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, or the United States government.

Thanks are due to Stuart Shieber, Lauri Karttunen, and Ray Perrault for their comments on earlier presentations of this material.

2 Grammars with Unification

The data representation discussed in this paper is applicable, with but minor changes, to a variety of grammar formalisms based on unification, such as definite-clause grammars [6], functional-unification grammar [4], lexical-functional grammar [2] and PATR-II [8]. For the sake of concreteness, however, our discussion will be in terms of the PATR-II formalism.

The basic idea of unification-based grammar formalisms is very simple. As with context-free grammars, grammar rules state how phrase types combine to yield other phrase types. But whereas a context-free grammar allows only a finite number of predefined atomic phrase types or *nonterminals*, a unification-based grammar will in general define implicitly an infinity of phrase types.

A phrase type is defined by a set of constraints. A grammar rule is a set of constraints between the type X_0 of a phrase and the types $X_1, \ldots, X_n$ of its constituents. The rule may be applied to the analysis of a string s_0 as the concatenation of constituents $s_1, \ldots, s_n$ if and only if the types of the s_i are compatible with the types X_i and the constraints in the rule.

Unification is the operation that determines whether two types are compatible by building the most general type compatible with both.

If the constraints are equations between attributes of phrase types, as is the case in PATR-II, two phrase types can be unified whenever they do not assign distinct values to the same attribute. The unification is then just the conjunction (set union) of the corresponding sets of constraints [5].

Here is a sample rule, in a simplified version of the PATR-

II notation:

$$X_0 \rightarrow X_1\, X_2 : \begin{array}{lcl} \langle X_0\ cat \rangle & = & S \\ \langle X_1\ cat \rangle & = & NP \\ \langle X_2\ cat \rangle & = & VP \\ \langle X_1\ agr \rangle & = & \langle X_2\ agr \rangle \\ \langle X_0\ trans \rangle & = & \langle X_2\ trans \rangle \\ \langle X_0\ trans\ arg_1 \rangle & = & \langle X_1\ trans \rangle \end{array} \qquad (1)$$

This rule may be read as stating that a phrase of type X_0 can be the concatenation of a phrase of type X_1 and a phrase of type X_2, provided that the attribute equations of the rule are satisfied if the phrases are substituted for their types. The equations state that phrases of types X_0, X_1, and X_2 have categories S, NP, and VP, respectively, that types X_1 and X_2 have the same agreement value, that types X_0 and X_2 have the same translation, and that the first argument of X_0's translation is the translation of X_1.

Formally, the expressions of the form $\langle l_1 \cdots l_m \rangle$ used in attribute equations are *paths* and each l_i is a *label*.

When all the phrase types in a rule are given constant *cat* (category) values by the rule, we can use an abbreviated notation in which the phrase type variables X_i are replaced by their category values and the category-setting equations are omitted. For example, rule (1) may be written as

$$S \rightarrow NP\ VP : \begin{array}{lcl} \langle NP\ agr \rangle & = & \langle VP\ agr \rangle \\ \langle S\ trans \rangle & = & \langle VP\ trans \rangle \\ \langle S\ trans\ arg_1 \rangle & = & \langle NP\ trans \rangle \end{array} \qquad (2)$$

In existing PATR-II implementations, phrase types are not actually represented by their sets of defining equations. Instead, they are represented by symbolic solutions of the equations in the form of directed acyclic graphs (*dags*) with arcs labeled by the attributes used in the equations. Dag nodes represent the values of attributes and an arc labeled by l goes from node m to node n if and only if, according to the equations, the value represented by m has n as the value of its l attribute [5].

A dag node (and by extension a dag) is said to be *atomic* if it represents a constant value; *complex* if it has some outgoing arcs; and a *leaf* if is is neither atomic or complex, that is, if it represents an as yet completely undetermined value. The *domain* dom(d) of a complex dag d is the set of labels on arcs leaving the top node of d. Given a dag d and a label $l \in$ dom(d) we denote by d/l the subdag of d at the end of the arc labeled l from the top node of d. By extension, for any path p whose labels are in the domains of the appropriate subdags, d/p represents the subdag of d at the end of path p from the root of d.

For uniformity, lexical entries and grammar rules are also represented by appropriate dags. For example, the dag for rule (1) is shown in Figure 1.

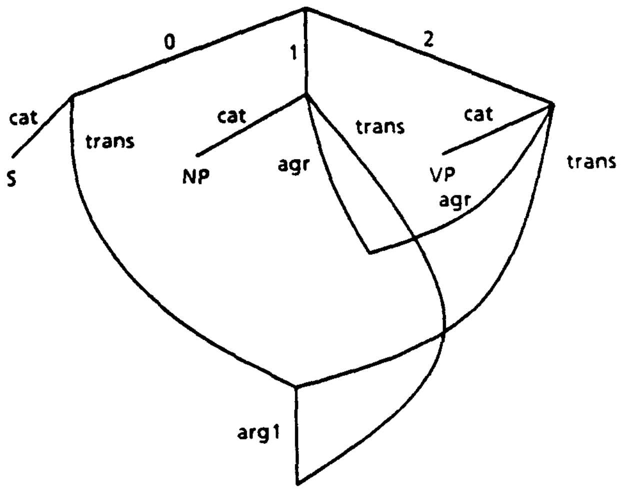

Figure 1: Dag Representation of a Rule

3 The Problem

In a chart parser [3] all the intermediate stages of derivations are encoded in *edges*, representing either incomplete (*active*) or complete (*passive*) phrases. For PATR-II, each edge contains a dag instance that represents the phrase type of that edge. The problem we address here is how to encode multiple dag instances efficiently.

In a chart parser for context-free grammars, the solution is trivial: instances can be represented by the unique internal names (that is, addresses) of their objects because the information contained in an instance is exactly the same as that in the original object.

In a parser for PATR-II or any other unification-based formalism, however, distinct instances of an object will in general specify different values for attributes left unspecified in the original object. Clearly, the attribute values specified for one instance are independent of those for another instance of the same object.

One obvious solution is to build new instances by copying the original object and then updating the copy with the new attribute values. This was the solution adopted in the first PATR-II parser [8]. The high cost of this solution both in time spent copying and in space required for the copies themselves constitutes the principal justification for employing the method described here.

4 Structure Sharing

Structure sharing is based on the observation that an initial object, together with a list of update records, contains the same information as the object that results from applying the updates to the initial object. In this way, we can trade the cost of actually applying the updates (with possible copying to avoid the destruction of the source object) against the cost of having to compute the effects of updates when examining the derived object. This reasoning applies in particular to dag instances that are the result of adding attribute values to other instances.

As in the variant of Boyer and Moore's method [1] used in Prolog [9], I shall represent a dag instance by a *molecule* (see Figure 2) consisting of

1. [A pointer to] the initial dag, the instance's *skeleton*
2. [A pointer to] a table of updates of the skeleton, the instance's *environment.*

Environments may contain two kinds of updates: *reroutings* that replace a dag node with another dag; *arc bindings* that add to a node a new outgoing arc pointing to a dag. Figure 3 shows the unification of the dags

$$\begin{aligned} I_1 &= [a : x, b : y] \\ I_2 &= [c : [d : e]] \end{aligned}$$

After unification, the top node of I_2 is rerouted to I_1 and the top node of I_1 gets an arc binding with label c and a value that is the subdag $[d : e]$ of I_2. As we shall see later, any update of a dag represented by a molecule is either an update of the molecule's skeleton or an update of a dag (to which the same reasoning applies) appearing in the molecule's enviroment. Therefore, the updates in a molecule's environment are always shown in figures tagged by a boxed number identifying the affected node in the molecule's skeleton.

The choice of which dag is rerouted and which one gets arc bindings is arbitrary.

For reasons discussed later, the cost of looking up instance node updates in Boyer and Moore's environment representation is $O(|d|)$, where $|d|$ is the length of the derivation (a sequence of resolutions) of the instance. In the present representation, however, this cost is only $O(\log |d|)$. This better performance is achieved by particularizing the environment representation and by splitting the representational scheme into two components: a *memory organization* and a *dag representation.*

A dag representation is a way of mapping the *mathematical entity* dag onto a memory. A memory organization is a way of putting together a memory that has certain properties with respect to lookup, updating and copying. One can think of the memory organization as the hardware and the dag representation as the data structure.

5 Memory organization

In practice, random-access memory can be accessed and updated in constant time. However, updates destroy old values, which is obviously unacceptable when dealing with alternative updates of the same data structure. If we want to keep the old version, we need to copy it first into a separate part of memory and change the copy instead. For the normal kind of memory, copying time is proportional to the size of the object copied.

The present scheme uses another type of memory organization — *virtual-copy arrays* — which requires $O(\log n)$ time to access or update an array with highest used index of n, but in which the old contents are not destroyed by updating. Virtual-copy arrays were developed by David H. D. Warren [10] as an implementation of extensible arrays for Prolog.

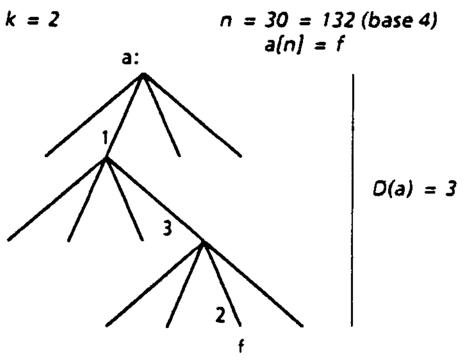

Figure 4: Virtual-Copy Array

Virtual-copy arrays provide a fully general memory structure: anything that can be stored in random-access memory can be stored in virtual-copy arrays, although pointers in machine memory correspond to indexes in a virtual-copy array. An updating operation takes a virtual-copy array, an index, and a new value and returns a new virtual-copy array with the new value stored at the given index. An access operation takes an array and an index, and returns the value at that index.

Basically, virtual-copy arrays are 2^k-ary trees for some fixed $k > 0$. Define the *depth* $d(n)$ of a tree node n to be 0 for the root and $d(p) + 1$ if p is the parent of n. Each virtual-copy array a has also a positive *depth* $D(a) \geq \max\{d(n) : n \text{ is a node of } a\}$. A tree node at depth $D(a)$ (necessarily a leaf) can be either an array element or the special marker $\perp$ for unassigned elements. All leaf nodes at depths lower than $D(a)$ are also $\perp$, indicating that no elements have yet been stored in the subarray below the node. With this arrangement, the array can store at most $2^{kD(a)}$ elements, numbered 0 through $2^{kD(a)} - 1$, but unused subarrays need not be allocated.

By numbering the 2^k daughters of a nonleaf node from 0 to $2^k - 1$, a path from a's root to an array element (a leaf at depth $D(a)$) can be represented by a sequence $n_0 \cdots n_{D(a)-1}$ in which n_d is the number of the branch taken at depth d. This sequence is just the base 2^k representation of the index n of the array element, with n_0 the most significant digit and $n_{D(a)}$ the least significant (Figure 4).

When a virtual-copy array a is updated, one of two things may happen. If the index for the updated element exceeds the maximum for the current depth (as in the $a[8] := g$ update in Figure 5), a new root node is created for the updated array and the old array becomes the leftmost daughter of the new root. Other nodes are also created, as appropriate, to reach the position of the new element. If, on the other hand, the index for the update is within the range for the current

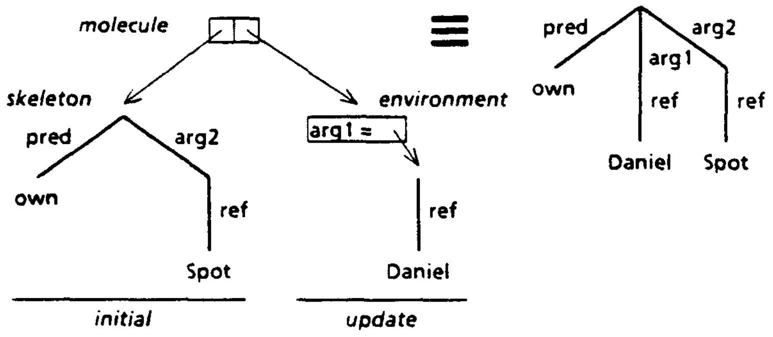

Figure 2: Molecule

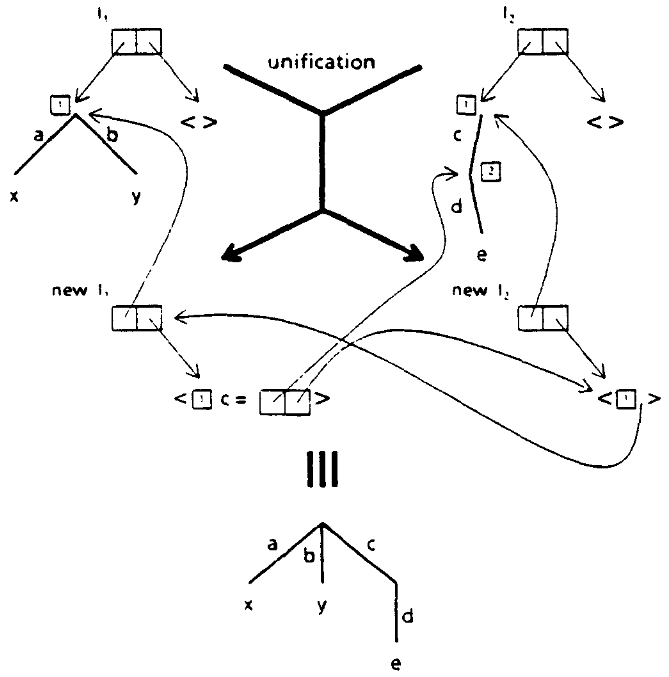

Figure 3: Unification of Two Molecules

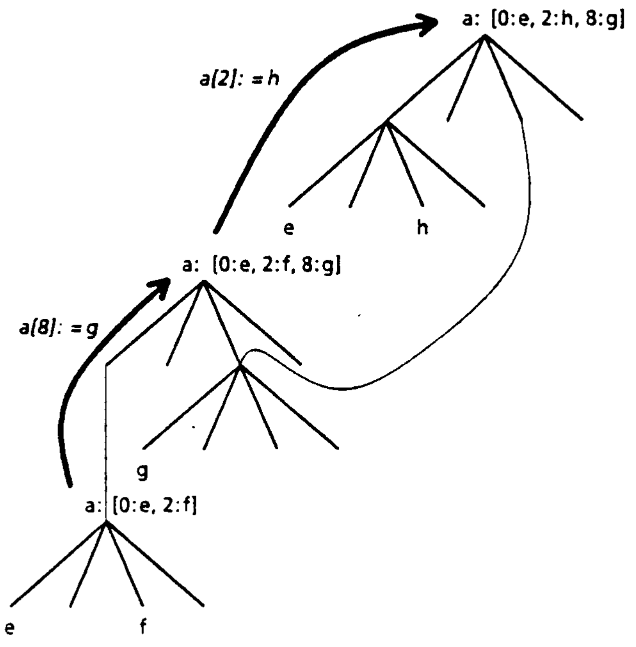

Figure 5: Updating Virtual-Copy Arrays

depth, the path from the root to the element being updated is copied and the old element is replaced in the new tree by the new element (as in the $a[2] := h$ update in Figure 5). This description assumes that the element being updated has already been set. If not, the branch to the element may terminate prematurely in a $\perp$ leaf, in which case new nodes are created to the required depth and attached to the appropriate position at the end of the new path from the root.

6 Dag representation

Any dag representation can be implemented with virtual-copy memory instead of random-access memory. If that were done for the original PATR-II copying implementation, a certain measure of structure sharing would be achieved.

The present scheme, however, goes well beyond that by using the method of structure sharing introduced in Section 4. As we saw there, an instance object is represented by a molecule, a pair consisting of a skeleton dag (from a rule or lexical entry) and an update environment. We shall now examine the structure of environments.

In a chart parser for PATR-II, dag instances in the chart fall into two classes.

Base instances are those associated with edges that are created directly from lexical entries or rules.

Derived instances occur in edges that result from the combination of a *left* and a *right parent edge* containing the *left* and *right parent instances* of the derived instance. The *left ancestors* of an instance (edge) are its left parent and that parent's ancestors, and similarly for *right ancestors*. I will assume, for ease of exposition, that a derived instance is always a subdag of the unification of its right parent with a subdag of its left parent. This is the case for most common parsing algorithms, although more general schemes are possible [7].

If the original Boyer-Moore scheme were used directly, the environment for a derived instance would consist of pointers to *left* and *right* parent instances, as well as a list of the updates needed to build the current instance from its parents. As noted before, this method requires a worst-case $O(|d|)$ search to find the updates that result in the current instance.

The present scheme relies on the fact that in the great majority of cases no instance is both the left and the right ancestor of another instance. I shall assume for the moment that this is always the case. In Section 9 this restriction will be removed.

It is a simple observation about unification that an update of a node of an instance I is either an update of I's skeleton or of the value (a subdag of another instance) of another update of I. If we iterate this reasoning, it becomes clear that every update is ultimately an update of the skeleton of a base instance ancestor of I. Since we assumed above that no instance could occur more than once in I's derivation, we can therefore conclude that I's environment consists only of updates of nodes in the skeletons of its base instance ancestors. By numbering the base instances of a derivation consecutively, we can then represent an environment by an array of *frames*, each containing all the updates of the skeleton of a given base instance.

Actually, the environment of an instance I will be a *branch environment* containing not only those updates directly relevant to I, but also all those that are relevant to the instances of I's particular branch through the parsing search space.

In the context of a given branch environment, it is then possible to represent a molecule by a pair consisting of a skeleton and the index of a frame in the environment. In particular, this representation can be used for all the values (dags) in updates.

More specifically, the frame of a base instance is an array of *update records* indexed by small integers representing the nodes of the instance's skeleton. An update record is either a list of arc bindings for distinct arc labels or a rerouting update. An arc binding is a pair consisting of a label and a molecule (the value of the arc binding). This represents an addition of an arc with that label and that value at the given node. A rerouting update is just a pointer to another molecule; it says that the subdag at that node in the updated dag is given by that molecule (rather than by whatever was in the initial skeleton).

To see how skeletons and bindings work together to represent a dag, consider the operation of finding the subdag $d/\langle l_1 \cdots l_m\rangle$ of dag d. For this purpose, we use a *current skeleton* s and a current frame f, given initially by the skeleton and frame of the molecule representing d. Now assume that the current skeleton s and current frame f correspond to the subdag $d' = d/\langle l_1 \cdots l_{i-1}\rangle$. To find $d/\langle l_1 \cdots l_i\rangle = d'/l_i$, we use the following method:

1. If the top node of s has been rerouted in f to a dag v, *dereference* d' by setting s and f from v and repeating this step; otherwise
2. If the top node of s has an arc labeled by l_i with value s', the subdag at l_i is given by the moledule (s', f); otherwise
3. If f contains an arc binding labeled l_i for the top node of s, the subdag at l_i is the value of the binding

If none of these steps can be applied, $\langle l_1 \cdots l_i\rangle$ is not a path from the root in d.

The details of the representation are illustrated by the example in Figure 6, which shows the passive edges for the chart analysis of the string ab according to the sample grammar

$$\begin{array}{llcl} S \rightarrow A\ B: & \langle S\ a\rangle & = & \langle A\rangle \\ & \langle S\ b\rangle & = & \langle B\rangle \\ & \langle S\ a\ x\rangle & = & \langle S\ b\ y\rangle \\ A \rightarrow a: & \langle A\ u\ v\rangle & = & a \\ B \rightarrow b: & \langle B\ u\ v\rangle & = & b \end{array} \tag{3}$$

For the sake of simplicity, only the subdags corresponding to the explicit equations in these rules are shown (ie., the *cat* dag arcs and the rule arcs 0, 1,... are omitted). In the figure, the three nonterminal edges (for phrase types S, A and B) are labeled by molecules representing the corresponding dags. The skeleton of each of the three molecules comes from the rule used to build the nonterminal. Each molecule points (via a frame index not shown in the figure) to a frame in the branch environment. The frames for the A and B edges contain arc bindings for the top nodes of the respective skeletons whereas the frame for the S edge reroute nodes 1 and 2 of the S rule skeleton to the A and B molecules respectively.

7 The Unification Algorithm

I shall now give the unification algorithm for two molecules (dags) in the same branch environment.

We can treat a complex dag d as a partial function from labels to dags that maps the label on each arc leaving the top node of the dag to the dag at the end of that arc. This allows us to define the following two operations between dags:

$$\begin{aligned} d_1 \setminus d_2 &= \{\langle l, d\rangle \in d_1 \mid l \notin \mathrm{dom}(d_2)\} \\ d_1 \lhd d_2 &= \{\langle l, d\rangle \in d_1 \mid l \in \mathrm{dom}(d_2)\} \end{aligned}$$

It is clear that $\mathrm{dom}(d_1 \lhd d_2) = \mathrm{dom}(d_2 \lhd d_1)$.

We also need the notion of dag dereferencing introduced in the last section. As a side effect of successive unifications, the top node of a dag may be rerouted to another dag whose top node will also end up being rerouted. Dereferencing is the process of following such chains of rerouting pointers to reach a dag that has not been rerouted.

The unification of dags d_1 and d_2 in environment e consists of the following steps:

1. Dereference d_1 and d_2
2. If d_1 and d_2 are identical, the unification is immediately successful
3. If d_1 is a leaf, add to e a rerouting from the top node of d_1 to d_2; otherwise
4. If d_2 is a leaf, add to e a rerouting from the top node of d_2 to d_1; otherwise
5. If d_1 and d_2 are complex dags, for each arc $\langle l, d\rangle \in d_1 \lhd d_2$ unify the dag d with the dag d' of the corresponding arc $\langle l, d'\rangle \in d_2 \lhd d_1$. Each of those unifications may add new bindings to e. If this unification of subdags is successful, all the arcs in $d_1 \setminus d_2$ are are entered in e as arc bindings for the top node of d_2 and finally the top node of d_1 is rerouted to d_2.
6. If none of the conditions above applies, the unification fails.

To determine whether a dag node is a leaf or complex, both the skeleton and the frame of the corresponding molecule must be examined. For a dereferenced molecule. the set of arcs leaving a node is just the union of the skeleton arcs and the arc bindings for the node. For this to make sense, the skeleton arcs and arc bindings for any molecule node must be disjoint. The interested reader will have no difficulty in proving that this property is preserved by the unification algorithm and therefore all molecules built from skeletons and empty frames by unification will satisfy it.

8 Mapping dags onto virtual-copy memory

As we saw above, any dag or set of dags constructed by the parser is built from just two kinds of material: (1) frames; (2) pieces of the initial skeletons from rules and lexical entries. The initial skeletons can be represented trivially by host language data structures, as they never change. Frames, though, are always being updated. A new frame is born with the creation of an instance of a rule or lexical entry when the rule or entry is used in some parsing step (uses of the same rule or entry in other steps beget their own frames). A frame is updated when the instance it belongs to participates in a unification.

During parsing, there are in general several possible ways of continuing a derivation. These correspond to alternative ways of updating a branch environment. In abstract terms,

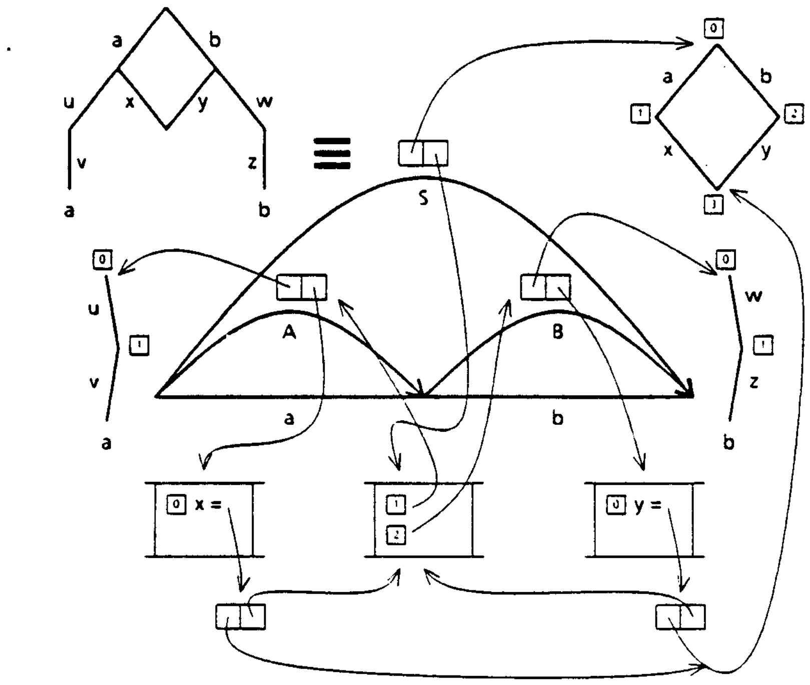

Figure 6: Structure-Sharing Chart

on coming to a choice point in the derivation with n possible continuations, $n-1$ copies of the environment are made, giving n environments — namely, one for each alternative. In fact, the use of virtual-copy arrays for environments and frames renders this copying unnecessary, so each continuation path performs its own updating of its version of the environment without interfering with the other paths. Thus, all unchanged portions of the environment are shared.

In fact, derivations as such are not explicit in a chart parser. Instead, the instance in each edge has its own branch environment, as described previously. Therefore, when two edges are combined, it is necessary to merge their environments. The cost of this merge operation is at most the same as the worst case cost for unification proper ($O(|d| \log |d|)$). However, in the very common case in which the ranges of frame indices of the two environments do not overlap, the merge cost is only $O(\log |d|)$.

To summarize, we have sharing at two levels: the Boyer-Moore style dag representation allows derived dag instances to share input data structures (skeletons), and the virtual-copy array environment representation allows different branches of the search space to share update records.

9 The Renaming Problem

In the foregoing discussion of the structure-sharing method, I assumed that the left and right ancestors of a derived instance were disjoint. In fact, it is easy to show that the condition holds whenever the grammar does not allow empty derived edges.

In contrast, it is possible to construct a grammar in which an empty derived edge with dag D is both a left and a right ancestor of another edge with dag E. Clearly, the two uses of D as an ancestor of E are mutually independent and the corresponding updates have to be segregated. In other words, we need two copies of the instance D. By analogy with theorem proving, I call this the *renaming* problem.

The current solution is to use *real* copying to turn the empty edge into a skeleton, which is then added to the chart. The new skeleton is then used in the normal fashion to produce multiple instances that are free of mutual interference.

10 Implementation

The representation described here has been used in a PATR-II parser implemented in Prolog. Two versions of the parser exist — one using an Earley-style algorithm related to Earley deduction [7], the other using a left-corner algorithm.

Preliminary tests of the left-corner algorithm with structure sharing on various grammars and input have shown parsing times as much as 60% faster (never less, in fact, than 40% faster) than those achieved by the same parsing algorithm with structure copying.

References

[1] R. S. Boyer and J S. Moore. The sharing of structure in theorem-proving programs. In *Machine Intelligence 7*, pages 101–116, John Wiley and Sons, New York, New York, 1972.

[2] J. Bresnan and R. Kaplan. Lexical-functional grammar: a formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–281, MIT Press, Cambridge, Massachusetts, 1982.

[3] M. Kay. *Algorithm Schemata and Data Structures in Syntactic Processing*. Technical Report, XEROX Palo Alto Research Center, Palo Alto, California, 1980. A version will appear in the proceedings of the Nobel Symposium on Text Processing, Gothenburg, 1980.

[4] M. Kay. Functional grammar. In *Proc. of the Fifth Annual Meeting of the Berkeley Linguistic Society*, pages 142–158, Berkeley Linguistic Society, Berkeley, California, February 17-19 1979.

[5] Fernando C. N. Pereira and Stuart M. Shieber. The semantics of grammar formalisms seen as computer languages. In *Proc. of Coling84*, pages 123–129, Association for Computational Linguistics, 1984.

[6] Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231-278, 1980.

[7] Fernando C. N. Pereira and David H. D. Warren. Parsing as deduction. In *Proc. of the 21st Annual Meeting of the Association for Computational Linguistics*, MIT, Cambridge, Massachusetts, June 15-17 1983.

[8] Stuart M. Shieber. The design of a computer language for linguistic information. In *Proc. of Coling84*, pages 362–366, Association for Computational Linguistics, 1984.

[9] David H. D. Warren. *Applied Logic - its use and implementation as programming tool*. PhD thesis, University of Edinburgh, Scotland, 1977. Reprinted as Technical Note 290, Artificial Intelligence Center, SRI, International, Menlo Park, California.

[10] David H. D. Warren. Logarithmic access arrays for Prolog. Unpublished program, 1983.