AN ABSTRACT MACHINE FOR ATTRIBUTE-VALUE LOGICS

Bob Carpenter Yan Qu
Computational Linguistics Program, Carnegie Mellon University
carp+@cmu.edu yqu@cs.cmu.edu

Abstract

A direct abstract machine implementation of the core attribute-value logic operations is shown to decrease the number of operations and conserve the amount of storage required when compared to interpreters or indirect compilers. In this paper, we describe the fundamental data structures and compilation techniques that we have employed to develop a unification and constraint-resolution engine capable of performance rivaling that of directly compiled Prolog terms while greatly exceeding Prolog in flexibility, expressiveness and modularity.

In this paper, we will discuss the core architecture of our machine. We begin with a survey of the data structures supporting the small set of attribute-value logic instructions. These instructions manipulate feature structures by means of features, equality, and typing, and manipulate the program state by search and sequencing operations. We further show how these core operations can be integrated with a broad range of standard parsing techniques.

Feature structures improve upon Prolog terms by allowing data to be organized by feature rather than by position. This encourages modular program development through the use of sparse structural descriptions which can be logically conjoined into larger units and directly executed. Standard linguistic representations, even of relatively simple local syntactic and semantic structures, typically run to hundreds of substructures. The type discipline we impose organizes information in an object-oriented manner by the multiple inheritance of classes and their associated features and type value constraints. In practice, this allows the construction of large-scale grammars in a relatively short period of time.

At run-time, eager copying and structure-sharing is replaced with lazy, incremental, and localized branch and write operations. In order to allow for applications with parallel search, incremental backtracking can be localized to disjunctive choice points within the description of a single structure, thus supporting the kind of conditional mutual consistency checks used in modern grammatical theories such as HPSG, GB, and LFG. Further attention is paid to the byte-coding of instructions and their efficient indexing and subsequent retrieval, all of which is keyed on type information.

1 Motivation

Modern attribute-value constraint-based grammars share their primary operational structure with logic programs. In the past decade, Prolog compilers, such as Warren's Abstract Machine (Aït-Kaci 1990), have supplanted interpreters as the execution method of choice for logic programs. This is in large part due to a 50-fold speed up in execution times and a reduction by an order of magnitude in terms of space required. In addition to efficiency, compilation also brings the opportunity for static error detection.

The vast majority of the time and space used by traditional unification-based grammar interpreters is spent on copying and unifying feature structures. For example, in a bottom-up chart parser, the standard process would be first to build a feature structure for a lexical entry, then to build the feature structures for the relevant rules, and then to unify the matching structures. The principal drawback to this approach is that complete feature structures have to be constructed, even though unification may result in failure. In the case of failure, this can amount to a substantial amount of wasted time and space. By adopting an incremental compiled approach, a description is compiled into a set of abstract machine instructions. At run-time a description is evaluated incrementally, one instruction at a time. In this way, conflicts can be detected as early as possible, before any irrelevant structure has been introduced. In practice, this often means that the inconsistency of a rule with a category can often be detected very

soon into the evaluation of its left corner (the leftmost daughter) or head. The reason that descriptions are compiled rather than entire representations is for the same reason as terms are compiled into low-level abstract machine instructions in the WAM, namely that it (1) allows unification to be replaced by the much faster abstract machine operations, and (2) it simplifies backtracking in the face of local non-determinism, which is the primary bottleneck for most unification-based parsers.

Compilation also allows static typing mechanisms to be exploited to reduce the amount of space used by a structure and to detect conceptual errors in user type declarations. For instance, as pointed out by Carpenter and Penn (1995), declaring the features appropriate for a given type of structure allows record-like structures to be used rather than lists of attribute-value pairs, resulting in a three-fold decrease in memory used per structure, as well as great savings in managing allocation and deallocation. In addition to the savings in space, the positions of the features are also known at compile time which eliminates the need to compare features and construct structure on the fly, another area which occupies a significant amount of time in interpreters. With type unification determined statically, type information can be incorporated by means of table look up, which often avoids the need for more complex structural unification. Errors such as undefined features, inappropriate types, inconsistent pairings of features, and so on can be easily detected at compile time.

2 Abstract Machine Architecture

There is more than a passing similarity between the structure and execution of logic programs and that of "unification-based" grammars; (Carpenter 1992) shows that the resolution model of Prolog execution can be generalized to grammars based on typed attribute-value logics.

Our architecture is based on a typed, breadth-first extension of Warren's Abstract Machine for compiling Prolog (Aït-Kaci 1991). The WAM defines a mapping between logical rules expressed as Horn clauses, and the execution of a sequential machine machine (von Roy 1990). The development of the WAM has brought Prolog into the mainstream as an efficient programming language for industrial strength tasks. Similar gains were made by Lisp and ML when adequate compilers were developed for those functional languages.

A first pass an abstract machine architecture for grammars based on typed attribute-value logic was used for the ALE system (Carpenter and Penn 1994, 1995). ALE compiled typed attribute-value logic grammars down to Prolog clauses, which were then compiled using the WAM. The problem with Prolog as a target language is the lack of true random access, destructive memory operations, and pointers, all of which are necessary to generate efficient compiled code. On the other hand, Prolog is a handy intermediary because of its built-in handling of search and simple pattern matching. The architecture we describe here is the result of cutting out the middle man in the compilation of ALE, while retaining the efficiencies arising from the WAM.

Following ALE, categories are expressed using attribute-value logic descriptions. These descriptions contain feature information, type information, and equality information, logically structured by conjunction (sequencing) and disjunction (non-deterministic choice). In this paper, we show how these descriptions are incrementally executed. A description is decomposed to a number of operations which are then applied one by one to a feature structure; at each stage of execution, either the information is successfully added to the structure, or a failure flag is raised and backtracking ensues.

Various parsing mechanisms, such as the chart parser built into ALE or a left-corner parser can then be run off the basic description resolution mechanism. Any choice or copy points in the parser can simply be interleaved with the choice points for the descriptions in the rules or lexical entries.

Aït-Kaci and Di Cosmo (1993) developed a compiler for sorted terms that are similar to ours, but without any type declarations for feature appropriateness. Their representation of structures is based on lists of feature/value pairs, which consumes around three times as much memory space as our encoding, and prevents the precompilation of unifications (see Carpenter

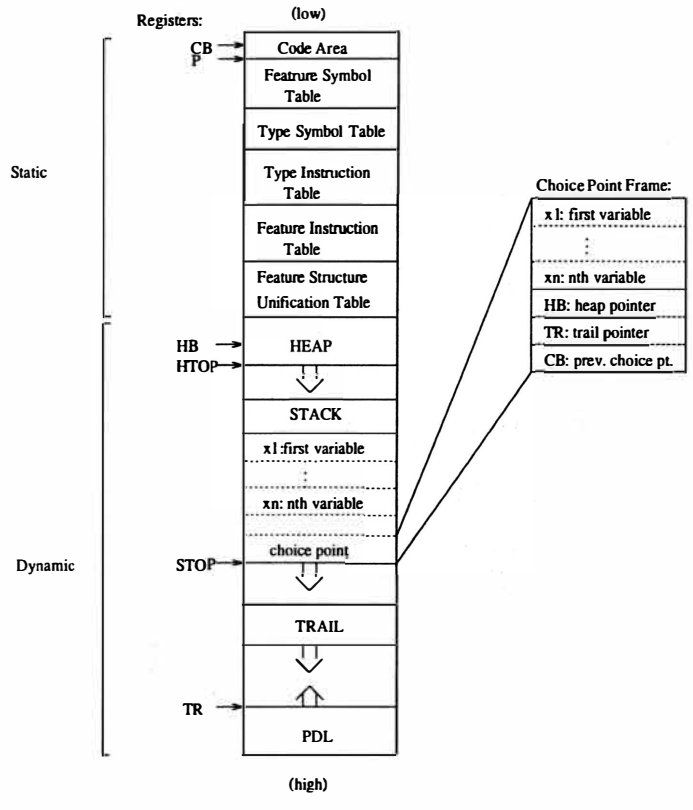


Figure 1: **Abstract Machine Memory Layout and Registers**

and Penn 1995). Wintner and Francez (1994) present a representation of feature structures based in large part on our approach (Qu 1994; Carpenter and Penn 1994). But their grammars are given in terms of Aït-Kaci's ψ-terms, which are suboptimal in terms of their control over choice points and "anonymous" variables; they are currently working to integrate disjunction into their architecture in a manner similar to ours (Wintner p.c.).

In the rest of this section, we describe the memory layout, instruction set, and execution mechanism of our system.

2.1 Memory areas

In this section, we describe the allocation of memory in the machine. Memory is divided into two major logical areas (Figure 1): static and dynamic. The static memory holds the program and its associated type declarations. This includes the symbol tables, type unification and type inference information, as well as the instructions associated with lexical entries and phrase structure schemata.

The dynamic memory holds the program execution state. Its primary areas include the **heap**, which contains the representations of feature structures, the **stack**, which keeps track of the local bindings of arguments, variables and structures (a stack provides a natural (though not the most economical) method for handling the allocation and deallocation of space for nested feature values; the values of variables reside at the bottom of the stack), the **trail**, which keeps track of information that needs to be restored on backtracking (pairs of addresses and their previous values), the **choice point stack**, which holds information about variable and stack bindings, an indicator of where to resume control, how far to unwind the trail, as well as information on recovery point in the heap and the latest choice point, and finally, the **push down list**, another stack structure which is used to store sequences of feature structures remaining to be unified for the unification algorithm.

```
% Type Signature

bot sub [t,s].
 t sub [t1,t2]
   intro [h:bot].
  t1 sub [t3]
     intro [f:bot].
  t2 sub [t3]
     intro [g:bot].
    t3 sub []
       intro [j:bot].
 s sub [s1,s2].
  s1 sub [].
  s2 sub [].
```

```
% Lexicon

word1 ---> t1.
word2 ---> f:t.
word3 ---> t1, f:(t2, g:t2), h:t3.
word4 ---> t1, f:(X, g:t2), h:X.
word5 ---> t1; t2.

% Grammar

rule1 rule
(s)
==>
cat> (t1, f:X),
cat> (t2, g:X).
```

Figure 2: **An Example Grammar in ALE Format**

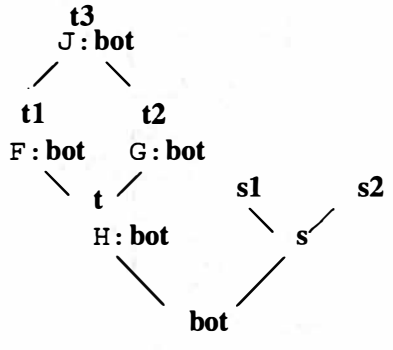


Figure 3: **An Example Type Hierarchy**

2.2 Representation of feature structures

In this section, we define an internal representation for feature structures in the abstract machine. The global block storage for representing feature structures is an addressable heap. The heap is an array of data cells, and is managed as a stack.

We distinguish three kinds of terms for representing feature structures: a *variable*, a *structure*, and a *pointer*. As in the WAM, we use explicit *tags* as part of the format of some heap cells to discriminate between these three sorts of heap data. The tags for the three types of data are STR, VAR, and PTR.

Recall that a feature structure Fs consists of two components: a type and a list of n feature values, $v_1, \ldots, v_n$, each of which is itself a feature structure. Such a structure is represented on the heap by means of $n+1$ contiguous cells. The type value is represented as a cell tagged by STR, as denoted as **<STR, τ>**, where τ is the value for the feature structure Fs. The n other cells contain references to the n appropriate features (in alphabetical order in the current compiler). The n heap cells representing the values will typically contain pointers to other structures, of the form **<PTR, p>**, where p is the index of another heap cell. Alternatively, if the value is a structure with no features, it can fit into one cell, and will thus not need to employ a level of indirection through the pointers. Similarly, we use a variable structure, where **<VAR, τ>** represents a structure of type τ whose feature values are unknown. By using variable structures of this kind, we are able to postpone some of the eager type inference that is automatically performed in ALE; if nothing is known about a structure's feature's values, we can leave them as variables, thus conserving space and time in the short run (these areas may be later overwritten with actual values and consume space on backtracking). For example, suppose we have a type hierarchy specification as in Figure 2. The diagram in Figure 3 displays the type signature in Figure 2 graphically. Figure 4 shows a heap representation for a feature structure satisfying the description `t1, f:(t2, g:t2), h:t3` starts at heap address 11. Note that the value for feature F is represented at heap address 12 as a PTR to a STR cell at address 14 because the

11	STR	t_1
12	PTR	14
13	VAR	t_3
14	STR	t_2
15	VAR	t_2
16	VAR	*bot*

Figure 4: **Example Heap Representation**

HTOP	Top of the heap
STOP	Top of the stack
P	Program counter
CB	Continuation code counter
HB	Heap pointer
TR	Trail pointer
$x_1, x_2, \ldots, x_n$	Registers for argument passing and temporary storage

Figure 5: **Execution State**

feature value for feature G of type **t2** is explicitly specified. Unspecified information on the type appropriateness of feature H of type **t2** is automatically recovered at address 16 with the default type **bot**, which is taken to be the most general type in the hierarchy. The value for feature H of type **t1** is represented as a VAR cell at address 13 rather than a STR cell, as no explicit feature information is specified for the structure of type **t3**.

2.3 Execution state

A description of the information stored to represent a given execution state of the machine is given in Figure 5.

2.4 The Instruction set

Figure 6 contains the instruction set for the abstract machine with a brief description of what each instruction does. The registers and arguments following some instructions are left out in the table. For details of these instructions, see (Qu 1994).

3 Compiling

In this section, we discuss how the typed feature structure logic can be compiled using the abstract machine. The syntactic representation form we adopt here follows the ALE format as described in (Carpenter and Penn 1994). We will first describe how type definitions are compiled, then we will explain how the descriptions are compiled, and lastly, we will discuss how the grammar rules are compiled.

3.1 Compiling type definitions

The first component of a grammar is a type definition. Type definitions include types for feature structures and declarations of appropriate features for each type. Enforcing an inheritance-based type discipline on feature structures yields the following characteristics for type and feature structure interaction:

- *Constraint Inheritance*: Type constraints on more general types are inherited by their more specific subtypes.

Instructions for adding a type	
BIND	Bind a new feature cell to an old feature cell
FRESH	Create a new feature cell
Instructions for getting a feature from a type	
GET	Give the position of the feature for a feature structure
ADD	Create a new feature structure with the type as value
Instructions for procedural control	
ALLOCATE	Allocate a new environment on the stack
DEALLOCATE	Deallocate an environment on the stack
JUMP	Jump to labeled clause and continue execution
Instructions for unification of feature structures	
SKIP1	Skip one heap cell for feature structure one
SKIP2	Skip one heap cell for feature structure two
UNIFY	Unify two heap cells
COPY1	Copy the heap cell of structure one
COPY2	Copy the heap cell of structure two
Choice instructions	
TRY	Allocate a new choice point
RETRY	Unwind heap and try next alternative (branch)
LASTTRY	Unwind heap and try last alternative (tail recursive)
Unifying instructions	
ADDNEW	Add a new type onto the heap
PUSH	Push the feature value to the stack
POP	Pop a value off stack
UNIFY_VAR	Update the binding of variables

Figure 6: **Abstract Machine Instructions**

- *Feature Appropriateness*: Each type must specify which features it can be defined for, and which types of value the features must take. The feature appropriateness is inherited along the type hierarchy in two ways.
  - Feature restrictions are inherited. That is, if a feature is appropriate for a type, then it is appropriate for all of the subtypes of the type.
  - Type restrictions on feature values are inherited. That is, if a type is the appropriate value for a feature, then all of its subtypes are appropriate values for the feature.

For example, consider the type inheritance relations in Figure 2, which are represented as a graph in Figure 3. The type t introduces one feature H and appropriate value constraints. The subtype t1 of t inherits the feature H and introduces a new feature F.

Compiling signatures involves the compilation of the type hierarchy and appropriateness conditions. The five global static storage areas are associated with this compilation: the symbol tables for types and features, the table for feature structure unification instructions, the table for type instructions, and the table for feature-value instructions. These instructions are all byte coded.

The feature structure unification table is indexed on pairs of types, and indicates first whether the types are consistent, and if so, what the value type is and how the features match up for further unification and type inference. The type instructions are also indexed on pairs of types, and indicate how to add a type's information to another structure and perform the relevant type inferences. The feature instructions are indexed by a feature and a type and indicate how to take the feature's value at the type, including any necessary type coercions. All instructions might also indicate failure.

3.2 Compiling descriptions

The compilation of descriptions follows the five clauses in the recursive definition of descriptions for types, features, variables, conjunction, and disjunction. Figure 7 shows the BNF grammar

```
⟨ desc ⟩ ::= ⟨ type ⟩
           | ⟨ variable ⟩
           | (⟨ feature ⟩: ⟨ desc ⟩)
           | (⟨ desc ⟩, ⟨ desc ⟩) ;; conjunction
           | (⟨ desc ⟩; ⟨ desc ⟩) ;; disjunction
```

Figure 7: **BNF Grammar for Descriptions in ALE**

14	STR	t_2
15	VAR	t_2
16	VAR	*bot*
17	STR	t_3
18	VAR	*bot*
19	PTR	15
20	PTR	16
21	VAR	*bot*

Figure 8: **Adding a Type to Heap**

for a description. A description is always evaluated by adding its information to the structure indicated by the top of the stack, which will point into the heap.

Compiling a type

The abstract machine instruction for adding a type to a feature structure is ADDNEW. This instruction adds the information associated with the type to a structure located at a specific heap address *heapaddr*. After dereferencing the structure by following PTR values, the result may be either a VAR cell or a STR cell. If the result is a VAR cell, we can just look up the result of unification in a table and update the variable structure's type (trailing if it is new), and backtrack if unification fails. Otherwise, assuming the type being added is τ and the resulting structure cell represents a feature structure of type τ', we need to consider four possibilities. If τ and τ' can't unify, we simply backtrack. If the new type τ is subsumed by the type τ' of the structure in the heap, nothing needs to be done and the command merely succeeds. Otherwise, we need to create a new structure on the heap for the result of the unification if τ' subsumes τ or if τ and τ' unify to a new type.

For example, suppose type **t1** is to be added to heap address 14 in Figure 8. From the type hierarchy in Figure 3, we know that **t1** and **t2** unify to a new type **t3**. Thus a new structure has to be created at heap address 17 and the old structure must be redirected to it by means of a PTR cell replacing the old value at 14 (which must be recorded on the trail if it might be needed for backtracking). Then we look up the precompiled instructions for adding a type τ' to a structure of type τ, which might involve creating new features with their most general values (which can be represented as VAR cells). The sequence of instructions for adding type **t1** to a structure of type **t2** is: FRESH (creating a new feature F for type **t3**), BIND (binding to previous value for feature H), BIND (binding to previous value for feature G), and FRESH (creating a new feature J for type **t3**); recall that the features are in alphabetic order. In general, the FRESH commands will indicate the appropriate type of the variable structure to be created.

Compiling feature/value pairs

To add a description of the form of `<feat>:<desc>` to a structure at a given heap address, we need to first find the value of the feature and then apply the description to the resulting value. If the structure is a variable cell, it must first have its appropriate features added with variable

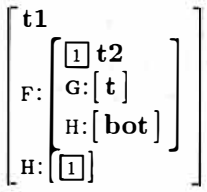


Figure 9: **Structure Sharing: Feature Structure Notation.**

```
      addnew    t1
1:    push      f
1:    unify_var 1
      addnew    t2
2:    push      g
      addnew    t
2:    pop
1:    pop
3:    push      h
2:    unify_var 1
3:    pop
```

(a)

22	STR	t_1
23	PTR	25
24	PTR	23
25	STR	$t2$
26	VAR	t
27	VAR	*bot*

(b)

Figure 10: **Structure Sharing: (a) Machine Instructions and (b) HEAP Representations After Dereferencing.**

values of the appropriate type. Next, if the feature is absent from the structure, the structure is coerced to one that is appropriate for the type by adding the most general type appropriate for the feature, which may result in failure and backtracking. Finally, we PUSH the value of the feature that is now guaranteed to exist onto the top of the stack and recursively add the description. After all of the information in a description is added to a feature value, we can POP the value from the stack. Note that if we do not need to reuse the structure currently at the top of the stack, we can overwrite the current stack position rather than pushing a new value.

Compiling variables

A variable occurring in two locations in a description has the same interpretation as in Prolog — the structures must be identical. For example, the variable `X` in the description `(t1, f:(X, g:t2), h:X)` indicates that feature `h` and feature `f` share the same value (see Figure 9, which represents the most general structure that satisfies this description).

In the abstract machine, space is allocated at the bottom of the stack for all of the local variables in a description. When the first instance of a variable is encountered, the variable's position on the stack is bound to the current structure on the top of the stack (a more clever compiler for variables, such as that found in the WAM, could avoid some of this duplication of representations on the stack). Any later attempts to add the same variable will simply unify the current structure with the value of the variable. A variable's lifespan is the entire phrase structure rule in which it occurs, which allows sharing across daughter and mother categories. Note that live variables (those whose last occurrence has not been encountered) must be maintained as part of the information in a choice point. The value of a variable need not be stored beyond its last occurrence in a clause.

The instruction UNIFY_VAR deals with a variable; note that it takes an argument that indicates the position of the variable in the stack to allow random access. The "unification" of a variable either sets the value of the variable on first encounter, or unifies the current structure with the previously established value of the variable. As can be seen with descriptions such as `(f:X ; g:Y), h:X`, the first use of a variable cannot be predicted statically; if the first branch

of the disjunction is taken, then `X` is set to the value of `f`, and then the value of `h` is unified with the value set by `f`, but if the second branch of the disjunction is taken, the value of `X` must be set by `h`. Note that a reasonable compiler can reuse the space allocated for a variable after its last instance is encountered. For instance, in the description `f:X, g:X, h:Y, j:Y`, the variables `X` and `Y` can use the same space, as long as it is reset to null after the last occurrence of `X`.

The instructions for the description `t1, f:(X, g:t2), h:X` are given in Figure 10(a). The resulting heap representation after their execution is shown in Figure 10(b). 1:UNIFY_VAR(1) sets the binding of `X`, and 2:UNIFY_VAR(1) unifies the current structure with the structure previously bound to `X`.

Compiling unification

It is significant to note that shared variables are the only instructions that call the unification operation. All other structural manipulation is carried out by the other primitive instructions. Instructions for unifying two feature structures at two different heap addresses are stored in a table, which can be represented as a (sparse) two dimensional array. There are a number of cases depending on whether the structures are variables or not, and whether they fail to unify, unify to a new type, or unify to one of their existing types. If one of the structures is a variable, it is simply updated as a pointer pointing to the other structure, and its type is added to the other structure. If one of the structures has a type subsuming the other, the more general structure is made to point to the more specific structure, and the features that need to be unified are read out of the table entry for the two types. These are indicated in terms of UNIFY and SKIP instructions, indicating whether the features in the more specific structure should be unified with the next feature in the more general structure or skipped. Finally, if the two structures unify to a new type of structure, a new structure must be allocated of the appropriate type and both previous structures are made to point to it. Then the new values are constructed as either pointers to a value in one of the existing structures or by creating a fresh variable cell. If both structures have a feature, these values must be pushed onto the unification stack so that their values are eventually unified. The instruction used is COPY(n) if the value is simply pointing to the value in the first or second structure (indicated by n), UNIFY if the value is determined by unifying the next value in both structures, or FRESH(τ) if a new variable structure of type τ should be created. As usual, any updates are trailed, and any failures cause backtracking.

Compiling conjunctions of descriptions

Conjunction amounts to simple sequencing of operations, which is the standard method by which the program pointer moves through the code area. The way in which information is packaged through conjunction and backtracking has a significant impact on run-time efficiency.

Different descriptions may be logically equivalent, but vary in performance, as do the examples in Figure 11(a), all of which generate the structure in Figure 11(b). The shortest form (1) is the most efficient while the longest description (3) is the least efficient. This can be seen from the size of the code in Figure 11(a) for each description. From the size of instructions, we can see that (2) and (3) involve more steps to represent the same feature structure. This ability to use concise descriptions is one significant way in which attribute-value logic descriptions provide finer control over term evaluation than Prolog terms; the order and amount of information added can be controlled by the programmer in a logically transparent fashion.

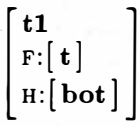
Compiling disjunctions of descriptions

When disjunction exists in descriptions, a failure of unification no longer yields irrevocable abortion of execution. Like the WAM, the execution of a description is left to right, and depth-first. When encountering a failure or a call for additional solutions, execution returns to the last considered choice point, restores information about variable binding, and continues execution.

Unlike the WAM, which only has one representation on the heap at any execution time, many parsing strategies rely on the dynamic programming technique of storing intermediate

(1) f:t	(2) t1, f:t	(3) t1, f:t, h:bot
push f addnew t pop	addnew t1 push f addnew t pop	addnew t1 push f addnew t pop push h addnew bot pop

(a)



(b)

Figure 11: **Different Descriptions, Same Feature Structure**

results. To accommodate this behavior, we have built in mechanisms for copying structures and reusing them — our memory architecture is not sensitive to the memory area into which a feature structure pointer is directed. In addition, the trail allows such "permanent" memory to be modified incrementally and copied later, thus avoiding one of the major time sinks of ALE (Carpenter and Penn 1995).

We now consider how disjunction executes. First, when a disjunct is evaluated, it may create side effects on the stack and heap by updating the variable binding on the stack and changing the feature structure values of the heap. These effects must be undone when considering an alternative. The data area TRAIL is used to keep track of all the heap cells that have been updated for the chosen disjunct and that need to be restored for another disjunct. TRAIL(a), which is the operation on heap address a, allocates two TRAIL cells to record the heap address and its current value.

Only conditional bindings, those affecting a variable existing before creation of the current choice point, need to be trailed. In our case, a conditional binding is one affecting the content of the heap cell before creation of the current choice point. To determine this, we use a global register HB to contain value of HTOP set at the time of latest choice point.

Like the WAM, we use a choice point frame to remember all the necessary states for restoring and continuing execution upon backtracking. Every choice point frame is initially pushed on the top of the choice point stack. The following information need to be stored in a choice point frame:

- *The next clause* (value of register CB): the next clause to try upon backtracking.
- *The current trail pointer* (value of register TR): the boundary where to *unwind* the trail upon alternative definition building.
- *The current top of heap* (value of register HB): the point before which the copying algorithm should consider. Heap cells after this point are created by the current choice point and need not to be copied.
- *Variable bindings* $x_1, \ldots, x_n$, where n is the number of variables in this description, including the implicit values created by feature values. If x_i is bound after the choice point, it is updated by the current heap address to which it is bound.

There are four instructions related to compiling disjunctions: TRY, RETRY, LASTTRY and JUMP. Descriptions of these instructions can be found in Figure 6. Consider how these instructions are used for compiling description `(t1;t2;t3), h:t`, the instructions for which are given in Figure 12. TRY allocates a choice point frame on the stack, then continues execution with the following instructions. JUMP directs the execution to the labeled instruction. RETRY resets all necessary information after backtracking, allocate a new choice point, then continues execution of the following instructions. LASTTRY is like TRY, but it indicates that variables need no longer be trailed because there are no more backtracking alternatives.

In evaluating disjunctions, two alternatives are possible. First, disjunctions can be handled by backtracking as in the WAM, using the structures for choice points we have described.

```
        try
        addnew   t1
        jump     l
        retry
        addnew   t2
        jump     l
        lasttry
        addnew   t3
l:      push     h
        addnew   t
        pop
```

Figure 12: **Compilation of disjunctions**

In this case, a choice point is recorded before the first alternative is considered. The second alternative is to eagerly copy, which allows chart-like memoization, thus avoiding the need for recording a choice point. The copying algorithm is not trivial, as a feature structure may be dispensed among different areas of heap spaces, and we need to constant looking up the trail to see whether a particular heap cell has been changed by the previous alternative or not. Details of how copying is done can be found in (Qu 1994).

4 Compiling a Parser

The architecture we have outlined for the processing of descriptions, with its features of incremental execution and efficient backtracking, make it an ideal candidate for integration with either a parser or a definite clause grammar system, or both, as found in ALE.

As described by Carpenter and Penn (1995), grammar rules can be compiled so as to have their own choice points interleaved with those of their embedded descriptions. In general, choice will be a matter of which grammar rule to apply, and which category to apply it to. In a simple chart parsing regime, the only matter that needs to be attended to is that of how the indexing will be done. For instance, in a bottom-up left-to-right chart-parser, after an edge is completed, all active edges immediately preceding it must be found and tested against it. To resume parsing an active edge, all that is needed is a pointer into the program space and a record of all of the variable instantiations. In testing the application of a grammar rule against a new completed edge, we must start by executing the first daughter description on the complete category. Thus choice points arising between different edges and between rules can be naturally accommodated in either a breadth-first (queue) or depth-first (stack) control strategy.

Memoizing parsers are not the only possibility for our description compilation scheme. Built into the model are the kind of management of non-determinism via a trail that lead to efficient implementations of backtracking parsers; for instance, LR or left-corner varieties. In the same way, definite clauses can be naturally integrated into our grammars and our control strategy, as demonstrated by Carpenter and Penn (1995).

5 Conclusion

We demonstrated how an abstract machine can be constructed for typed attribute-value logics along the lines of the Warren Abstract Machine for Prolog. By directly implementing the appropriate data structures, indexing mechanisms, and search engines, a parser can be built for feature structures that is as fast as the execution of a Prolog compiler. This greatly improves on the current state of the art, which is represented by either direct interpreted grammars or ones which are indirectly compiled by means of a detour through Prolog.

References

[1] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[2] H. Aït-Kaci and R. Di Cosmo. Compiling order-sorted feature term unification. Technical Report PRL 7, Digital Paris Research Laboratory, 1993.

[3] B. Carpenter. *The Logic of Typed Feature Structures.* Cambridge Tracts in Theoretical Computer Science 32. Cambridge University Press, New York, NY, 1992.

[4] B. Carpenter and G. Penn. Ale 2.0 users' guide - the attribute-logic engine. Technical report, Computational Lingusitcs, Carnegie Mellon University, 1994.

[5] B. Carpenter and G. Penn. Compiling typed attribute-value logic grammars. In H. Bunt and M. Tomita, editors, *Current Issues in Parsing Technologies*, volume 2. Kluwer, Philadelphia, 1995.

[6] Y. Qu. An abstract machine for typed feature structure theories. Master's thesis, Carnegie Mellon University, 1994.

[7] P. von Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, 1990.

[8] S. Wintner and N. Francez. Abstract machine for typed feature structures. In *Proceedings of the Conference on Natural Language Understanding and Logic Programming*, 1994.