TR AI2000-284, AI Lab, Department of Computer Sciences, Univ. of Texas at Austin
`http://www.cs.utexas.edu/users/ai-lab/`

The Algernon Abstract Machine: Compiling and Executing Rule-Based Programs

Micheal Hewett

Department of Computer Sciences
University of Texas
Taylor Hall 2.124
Austin, TX 78712-1188

May, 2000

Abstract

This report describes the Algernon Abstract Machine, an execution environment for access-limited logic programs. Like the Warren Abstract Machine for Prolog, the Algernon Abstract Machine (AAM) consists of an assembly-level language and a set of registers on which the language instructions operate. The AAM has a well-defined interface to a knowledge base, enabling it to be used in conjunction with any frame-based knowledge base. We describe the instructions of the AAM language and the process of compiling queries, assertions, and forward and backward chaining rules into AAM code. The restrictions of access limitation are handled by the compiler, so the abstract machine itself can execute general-purpose forward and backward chaining rules. We also describe how the AAM can be used as an interface to a network-accessible knowledge base.

Building a system around an abstract machine has several advantages, including enhancing the portability of user-level code to different execution environments. This has been apparent in the success of Prolog (using the Warren Abstract Machine) and Java (using the Java Virtual Machine). Additional advantages include enhanced maintainability, modularity and performance monitoring.

Table of Contents

1 Introduction

Intelligent systems, from rule-based systems to neural nets, are computationally intensive. Those who implement Artificial Intelligence programs are continually searching for more efficient algorithms and mechanisms. Solutions have come from many directions: *efficient activation mechanisms* (in rule-based systems [Forgy, 1982; Doorenbos, 1994], blackboard systems [Hewett and Hewett, 1997]); *parallelism* (in neural nets [Rumelhart and McClelland, 1986], blackboard systems [Rice, *et al.*, 1989], rule-based systems [Miranker and Lofaso, 1991]); and even *specialized hardware* [Hillis, 1985]. But efficiency is often gained at the expense of flexibility of representation and portability.

Perhaps the most successful balance of efficiency, flexibility and portability at the architecture level has been the *Warren Abstract Machine* [Warren, 1977; Aït-Kaci, 1991] for resolution-based theorem proving. The WAM has been the basis for many, if not all, efficient and portable Prolog implementations. In response, Prolog has become the language of choice for many AI scientists.

This paper describes the *Algernon Abstract Machine*[1], a system for executing access-limited logic programs, and rule-based programs in general. It is based on both the WAM and the SECD machine for LISP [Kogge, 1991]. We have implemented the AAM and used it as the underlying architecture of a new version of the Algernon [Crawford and Kuipers, 1991] reasoning system. One of the few drawbacks of the WAM has been the dearth of readable, understandable explanations of the WAM. It is the author's hope that this document is both readable and useful.

The next section describes access-limited logic and the reasoning mechanisms it uses. It is the basis for understanding the abstract machine described in Section 3. Detailed descriptions of AAM operations can be found there and in the Appendices. Section 5 contains a trace of the AAM while executing a program.

[1] This work builds on earlier implementations of Algernon by James Crawford and Benjamin Kuipers, who also provided valuable insights for this work. Spencer Bishop was the principal designer of the current unification algorithm and also made substantial contributions in the areas of theoretical analysis, debugging and testing.

2 Access-limited logic

This section summarizes methods of representation and reasoning using access-limited logic (*ALL*)[2]. *ALL* addresses the problem of tractable reasoning within a large symbolic knowledge base containing objects and relations (e.g. a frame-based representation or its equivalent semantic net). It provides both forward and backward chaining, and imposes one significant constraint on the programmer: only data related to an already-known data item may be accessed. This inhibits the programmer from performing a *global search* based on attributes. Instead, the programmer should perform a *directed search* (access-limited search) starting from a known frame.

2.1 Knowledge representation

The basic representation in *ALL* involves predicates of the form `(r a b)` where `r` is the *relation* and `a` and `b` are the arguments. However, *ALL* presumes that the knowledge base stores these predicates as frames. Figure 2.1 shows the frame-relation equivalence, whe represents the frame, `r` represents the slot, and `b` the value of the slot. This representation can be generalized to *n*-ary relations by storing the value as a list of the last *n-1* arguments of the relation.

```
                                frame:
(slot frame value)  ≡              slot:  value
```

Fig. 2.1. The predicate-frame equivalence.

We will refer to predicates as *clauses*. Sequences of clauses are *paths* and are deemed *access paths* if they satisfy the access limitation requirements:

- an empty path is an access path.
- in a non-empty path, the frame and slot arguments of the first clause must be constants or bound variables, and the remaining clauses must form an access path (possibly using additional bindings created by processing the first clause).

These restrictions do not allow global pattern matching as performed in Prolog, OPS5 or other deductive systems. An example illustrates this restriction. The path:

```
((sister John ?x) (likes ?x chocolate))
```

is a legal access-limited path meaning "*Does John have a sister who likes chocolate?*", but the logically equivalent path:

```
((likes ?x chocolate) (sister John ?x))
```

meaning "*Is there someone who likes chocolate who is John's sister?*" is not legal because the first clause does not have a bound frame argument and thus would require a global search of a

[2] Sections 2.1 through 2.4 are derived from Chapters 1 and 7 of [Crawford, 1990].

(single-indexed, frame-based) knowledge base. In practice, this limitation does not severely restrict expressiveness because a knowledge base can always be constructed with one "root" frame containing relations to every other frame in the KB, thus allowing the equivalent of a global search starting from the "root" frame of the KB. Access limitation is commonly used in many efficient Prolog-style programs to restrict search, even though it is not required by Prolog.

Rules in *ALL* are constructed using access paths in the antecedent and consequent of the rule. Forward-chaining rules are illustrated by the following example:

```
((sister John ?x)       ;; Every sister of John who likes
 (likes ?x chocolate))       ;; chocolate is a chocolate lover.
→
((chocolate-lover ?x true))
```

A forward-chaining rule activates when an **assertion** to the KB matches (unifies with) the first clause of its antecedent. The antecedent path is processed in *query mode*, where the clauses may retrieve values from the KB. The consequent path is processed in *assert mode*, in which fully instantiated clauses are stored in the KB.

A backward-chaining version of the above rule would be:

```
((chocolate-lover ?x true))  ;; A chocolate lover is any
←                            ;; sister of John who likes
((sister John ?x)            ;; chocolate.
 (likes ?x chocolate))
```

Like forward-chaining rules, the antecedent is processed in query mode and the consequent is processed in assert mode. A backward-chaining rule activates when a **query** to the KB matches (unifies with) the first clause of its consequent.

2.1.1 Antecedent processing

In many rule-based systems, the antecedents are patterns that, when fully matched, cause the consequent to be processed. These patterns are order-independent, except perhaps for efficiency. For efficient activation, antecedent patterns require an external pattern-matching component such as a RETE network. However, in *ALL*, antecedent clauses are **not** order-independent. Clauses may use bindings from preceding clauses.

In *ALL*, when a rule activates, the antecedent clauses are processed sequentially. If a clause fails, the antecedent processing stops but a *rule continuation* is created from the antecedent clause that failed. Rule continuations are forward-chaining rules formed from the partial antecedents of other rules. Rule continuations will complete the antecedent processing if information later becomes available that makes an antecedent clause true. The use of rule continuations guarantees that the same deductions will be made independent of the order of *assertions*. However, some results are still dependent on the order of *queries*. Rule continuations implement an active form of the RETE pattern-matcher that allows seamless mixed-mode reasoning as described in Section 2.5.

2.2 Processing modes

The user interacts with the system using the functions `tell` and `ask`. Each takes an access path as an argument. The former processes the path in *assert mode*, while the latter processes the path in *query mode*. The processing modes are described below. The user interface contains various macros for creating frame, slots and rules, as well as other useful routines. Further information about using the Algernon implementation of *ALL* can be found in [Kuipers, 1994]. The tables in Figures 2.2 and 2.3 describe the basic forms that *ALL* can process.

Form	Description
ground assertion or verify	A form containing no variables.
Non-ground assertion, query, or verify, with or without a binding list	A form containing at least one variable.
ground paths (assert, query, verify)	A sequence of ground assertions, queries or verifies.
Non-ground paths	A sequence of forms, each of which may have variables but which must form an access path. The path has an associated binding list.

Figure 2.2. Basic representation forms in access-limited logic.

In either assert or query mode, the system will encounter both ground and non-ground clauses. The two modes differ in how they treat each type of clause.

Form	Example
ground assert/query	`(status door-3 open 19.57)`

assert/verify with variables and binding list	`(status door-3 ?state ?amount)`
	`?state = open, ?amount = 12.12`

query with variables and binding list	`(status door-3 ?state ?amount)`
	`?state = open`

ground assert/verify path	`((status door-3 open 19.90)`
	` (entrance room door-3))`

query path	`((status door-3 ?state ?amount)`
	` (type ?state ?position-type))`

path assert/verify/query with variables & binding list	`((status door-3 ?state ?amount)`
	` (amount space-available ?amount))`
	`?state = open, ?amount = 6.14`

Figure 2.3. Examples of representations in *ALL*.

2.2.1 Assert mode

In assert mode, ground clauses are stored in the KB, while non-ground clauses are treated as knowledge base queries. If a query returns several values for the unbound variable(s) in a non-ground clause, the reasoning process will branch and follow each branch (i.e. each binding) independently. For example, in assert mode the path:

```
((sister John ?x) (likes ?x cats))
```

means "All of John's sisters like cats". Processing it will store the "likes cats" fact for every sister of John. If John has five sisters, processing will make a five-way branch at the first clause, so the second clause will execute five times, once for each different binding of `?x`. The similar path:

```
((sister John Judy) (likes Judy cats))
```

will never branch. Instead it will assert two facts into the KB.

Knowledge base assertions activate relevant forward-chaining rules (called *if-added rules*) unless the assertion was already known in the KB. Rule activation is discussed in Section 2.3.

2.2.2 Query mode

In query mode, non-ground clauses are treated as knowledge base queries, as in assert mode. Likewise, if the KB returns several values for the unbound variable(s) in the clause, the reasoning process will branch and follow each binding independently. However, in query mode ground clauses are *verified*—that is, they return a true result if the clause exists in the KB, and return a false value otherwise.

Using the example above, in query mode the path :

```
((sister John ?x) (likes ?x cats))
```

means "Do any of John's sisters like cats?". The answer is either "No" or one or more "Yes, `?x` does" responses, with `?x` bound to each value that makes the statement true.

Each *new* knowledge base query activates backward-chaining rules (called *if-needed rules*) which are processed prior to the query so that they can generate values to be retrieved. If a query has previously been made, any relevant if-needed rules will have already created rule continuations which would have asserted any new information. So the duplicate query is guaranteed to retrieve every possible piece of information without needing to activate any rules. An exception to this guarantee occurs when new rules are asserted between any two duplicate queries. In that case, the query should fire if-needed rules again. For this reason, runtime rule assertion is not allowed in the *ALL* theory (although it is allowed in the Algernon implementation of *ALL*).

2.3 Rule activation

In *ALL*, rules are assigned to slots (i.e. relations) or sets. Slot rules are activated when a slot is asserted or queried. Set rules are activated when a relation involving a member of the set is asserted or queried. The set member must be the first argument of the relation. During rule activation, rules are first retrieved from these locations. Then the *key clause* of the rule (the first clause of the antecedent for forward-chaining rules; the first clause of the consequent for backward-chaining rules) must unify with the query or assertion in order for the rule to activate. Although in theory rule activations in *ALL* are processed in parallel, in practice the activated rules are placed on a queue and processed sequentially. The user can control the order in which the activated rules are processed.

2.4 Theoretical results for access-limited logic

This section summarizes the completeness and time complexity results from Crawford's thesis cited above.

2.4.1 Completeness

Since *ALL* does not allow a global pattern match over frames, reasoning in *ALL* is not complete. For example, answers to queries such as "What has the color green?" are not directly retrievable even though the knowledge base may contain information that could answer the query. However, a form of completeness called *Socratic completeness* is supported. If a tutor provides the right set of leading questions and the knowledge base supports answering those questions, the query can be answered.

In practice, this means that if one can describe a directed path from a known starting point to the solution, the solution will be found. Usually this restriction does not seriously limit the expressiveness of the language. Many logic programs use access-limitation to reduce the execution time of their systems (by reducing the branching factor), even though they are not required to do so.

2.4.2 Time complexity

As shown in Chapter 7 of [Crawford, 1990], the upper bound on runtime of a primitive query or assertion in an *ALL* system with the restrictions shown below is proportional to:

$$c^5 \times o^2 \times r^5 \times f^{5v+m}$$

where c is the the maximum number of clauses in the antecedent of any rule in the KB, o is the number of unique queries and assertions generated by the operation, r is the number of rules in the knowledge base, f is the number of frames reachable via the operation, v is the maximum number of variables in any of the rules, and m is the maximum arity of a slot. This equation is polynomial in the number of frames in the KB, under the following restrictions:

- The number of variables per rule (v) is bounded.
- The maximum arity of any slot (m) is bounded.
- The program is not allowed to assert new frames or rules during execution.

Under these conditions, *ALL* has a polynomial runtime because it is not expressive enough to represent general solutions to NP-hard problems. Non-polynomial problems *can* be solved, but the implementation of a general solution requires an unbounded number of variables.

Implementation of the polynomial guarantee requires that rules not be fired for duplicate assertions and queries. It is "safe" to not fire rules in these situations because rule continuations guarantee that the correct results will be returned, as discussed in Sections 2.5 and 2.6.

Even though the time complexity equation is polynomial, it is unclear that it is a useful polynomial. Because of the way the time complexity is analyzed (it assumes that the program runs until the knowledge base is "full") it provides an extremely relaxed upper bound on the run time.

For example, what does the equation tell us about the relatively small system described below?

$r = 10$	10	rules
$c = 5$	5	clauses per rule
$o = 5$	5	operations per rule
$v = 5$	5	variables per rule
$m = 2$	2	maximum arity of a slot

The time complexity becomes:

$$7.8125 \times f^{27} \times 10^{9}$$

The table below shows how the upper bound varies based on *f*, the number of reachable frames:

f	**time upper bound**
10	7.8125×10^{36}
100	7.8125×10^{63}
1000	7.8125×10^{90}

Generously estimating one operation per microsecond, the upper bound for the smallest of th above systems corresponds to 2.5×10^{23} years. Since the estimated age of the universe is ap- proximately 10^{10} years, the time complexity formula does not provide any useful non-theoretical results.

Clearly, we must resort to experimental results on a wide range of systems to determine the actual run time effects of access-limited logic.

2.5 Seamless interleaved forward and backward chaining

The *ALL* integration of forward and backward chained reasoning has several nice features:

- Identical syntax for both forward and backward-chaining rules;
- Requires no external pattern-matching network;

- Each unique query need only be derived once. The system guarantees that all available answers can be directly retrieved for subsequent duplicate queries.
- Knowledge base content is independent of the order of assertions.

OPS5-style matching [Brownston et al, 1985] and use of the RETE network have apparently hindered the development of good syntaxes for mixed-mode reasoning. In OPS5, the antecedent clauses of its forward-chaining rules are simply patterns to be matched. When a system is run, the antecedents are compiled away into a RETE network. This makes backward-chaining using the same syntax problematic in several ways:

- The antecedents of a backward-chaining rule are a list of sub-goals to be fulfilled, usually in some reasonable order. But the standard RETE network does not allow for ordered retrieval of the antecedents.
- Since the antecedents are compiled away and unordered, the system can not determine when to fire rules in order to fulfill an antecedent.

These problems have been addressed in several, mostly awkward, ways. The KnowledgeWorks system [Harlequin] uses a uniform rule syntax, but has two inference engines: OPS-style for forward chaining rules; Prolog for backward chaining rules. The M.4 system [Teknowledge] is primarily a backward chaining system; forward chaining rules have a different syntax.

The MIKE rule-based shell [Eisenstadt and Brayshaw, 1990] has a uniform syntax, but requires the programmer to explicitly state which antecedent clauses should fire backward chaining rules. For example the following forward chaining rule activates a backward chaining rule via the deduce keyword.

```
;; Forward-chaining rule in MIKE
rule refinement_to_subclass forward
  if
    goal(refine) &
    possible(DiseaseClass) &
    deduce allowable(Subclass)
  then
    announce ['just refined down to subclass ',Subclass] &
    add possible(Subclass).
```

In the above example, `allowable(X)` is the consequent of a backward chaining rule defined elsewhere. The purpose of the deduce keyword is presumably two-fold:

- It tells the system not to compile the clause into the RETE network.
- It allows efficient activation of backward-chaining rules by clause name.

However, there are several drawbacks to this method: the language requires an extra keyword; backward chaining does not start until the other clauses are satisfied; and in MIKE the deductions are not asserted into the KB, and thus have to be recreated each time they are needed.

The `Eclipse` system [Haley] comes closest to Algernon in its seamless integration of forward and backward chaining rules. Eclipse has a uniform syntax for forward and backward chaining rules with the addition of a **`goal`** keyword. Below are a pair of rules, one forward-chaining and one backward-chaining, in `Eclipse`.

```
;; backward chaining rule in Eclipse
(defrule cousin
    (goal (cousin ?x ?y))
    (parent ?x ?p1)
    (parent ?y ?p2)
    (sibling ?p1 ?p2)
    =>
    (assert (cousin ?x ?y)))

 ;; forward chaining rule
(defrule cousins-may-inherit-trait-goal-generation-1
    (has ?x ?trait)
    =>
    (assert (goal (cousin ?x ?y))))
```

Although this implementation is more general than the MIKE implementation, we again see the somewhat ugly explicit invocation of backward chaining rules.

In contrast to the above systems, *ALL* prescribes a uniform syntax for both forward and backward chaining rules. It uses query and assert modes to determine which types of rules to fire. Antecedents are processed in *query mode* in which any antecedent clause may fire a backward chaining rule.

2.5.1 Rule families

The ECLIPSE system is the only system other than Algernon that addresses the third point of the features listed at the beginning of this section:

- Each unique query need only be derived once. The system guarantees that all available answers can be directly retrieved for subsequent duplicate queries.

ECLIPSE and Algernon both automatically convert backward chaining rules into *rule families*. A backward chaining rule with n antecedents is converted into a rule family consisting of the backward chaining rule and n forward chaining rules.

The rule family of a backward chaining rule ensures that every possible conclusion will be deduced even if the query is made only once. It does this by instantiating the forward chaining rules as rule continuations when the antecedent clauses are encountered. For example, the backward chaining rule:

```
c1 ← a1 a2 a3                RULE17
```

is automatically transformed into the rule family:

```
c1 ← a1 a2 a3            RULE17
a1 a2 a3 → c1            RULE17-001
   a2 a3 → c1            RULE17-002
      a3 → c1            RULE17-003
```

If RULE17 is activated, RULE17-001 is also instantiated with the bindings from the activation of its *root rule*, RULE17. If and when clause a2 of any instantiation of either RULE17 or RULE17-001 is processed, RULE17-002 is instantiated with the appropriate bindings from the active rule.

To illustrate how this works, consider the cousin rule implemented in *ALL*:

```
((cousin ?x ?y)
 <-
 (parent  ?x  ?p1)
 (sibling ?p1 ?p2)
 (child   ?p2 ?y)
 )
```

Suppose that when the query (cousin Arnold ?c) is made, the knowledge base contains information about only one sibling of Arnold's parents. In that case, the query will compute some of Arnold's cousins. If, later, the new fact (sibling Arnold's-Dad Joe) is asserted, the system may be able to deduce more cousins for Arnold. We want to system to make these deductions without requiring the user to repeat the cousin query again.

The rule family approach will do this for us. When the new sibling fact is asserted, the equivalent of RULE17-002 above will activate (because the sibling clause is the second antecedent of the cousin rule). Since this rule continuation is a forward chaining rule, it will deduce any newly-found cousins of Arnold and the knowledge in the system will be complete. The next time a query is made to retrieve Arnold's cousins, the information will already be there and no further rules need to be fired.

Rule continuations ensure that the set of facts deduced by the knowledge base is independent of the order in which other facts are asserted. This is also one of the guarantees of the Rete network. In many ways, rule continuations are equivalent to a form of the Rete network.

2.5.2 Activating rule continuations

Under the guarantees of access-limited logic, the frame and slot of a clause will be bound when the clause is processed. Rule continuation closures, which are created as clauses are processed, are stored on a special facet of the frame and slot of the clause. If another value is asserted into that slot, the closure will activate and continue processing what was originally a backward chaining rule.

The activation of rule continuations is an exception to the normal *ALL* method of activating rules that belong to sets of which a frame is a member. The activations could be placed in an appropriate set, but the method used in Algernon is more efficient because the rule will be selected for possible instantiation less often. We should note that rule continuations are created for antecedents of forward-chaining rules too in order to preserve the independence of knowledge base contents from order of assertion.

2.5.3 Comparing OPS-style rules and ALL rules

In OPS-style rules, the antecedent clauses are patterns. When all of the patterns match a set of facts, the consequent(s) of the rule are asserted. Treating antecedents as a set of facts makes rule activation independent of the order of assertions. In *ALL*, antecedents are access paths that are processed sequentially in query mode. The use of query mode means that backward chaining rules will automatically be activated as necessary.

The sequential processing of antecedents would seem to make the conclusions dependent on the order in which facts were asserted to the knowledge base. However, as described above, rule continuations perform the same duty without the need for an external pattern matching network.

Much work has been invested over the last decade to make RETE-class activation networks efficient. The match problem is known to be an NP-hard problem [Tambe and Rosenbloom, 1994]. *ALL*'s method, based on rule continuations and activation from rule sets is more efficient because it partitions the set of rules into locally-active subsets, but it could suffer from combinatorial space usage. The number of continuations that could be created from a single rule is F^{ac}, where F is the number of frames capable of instantiating an argument of a slot, a is the arity of the slot, and c is the number of clauses in the antecedent of a rule. In practice, continuations tend to collect on certain frame-slots and are not spread all over the knowledge base.

2.6 Summary of Section 2

In this section we have discussed reasoning in *ALL*, as well as *ALL*'s Socratic completeness guarantee. If an access-limited path can be described to a solution, *ALL* guarantees that it will be found. The time complexity of *ALL* has a very large polynomial upper bound if the number of variables per rule is bounded.

ALL's seamless interleaved forward and backward chaining mechanism is a better design than many, if not all, existing rule-based systems that support mixed-mode reasoning. It uses rule continuations to maintain the same guarantees as RETE regarding independence of results from assertion order.

3 The Algernon Abstract Machine

This section describes the Algernon Abstract Machine (AAM). Included in this section are methods of compiling and processing *ALL* statements. The design of the AAM is influenced by the SECD machine for LISP [Kogge, 1991] and the Warren Abstract Machine (WAM) for Prolog [Warren, 1977; Aït-Kaci, 1991]. Since Prolog reasoning is resolution-based and *ALL* reasoning involves forward-chaining, backward-chaining and rule continuations, the AAM differs significantly from the WAM. A LISP-based implementation of the AAM has been implemented as version 3 of Algernon [Crawford and Kuipers, 1991].

This chapter is intended to serve as a reference manual for future *ALL* implementations. Abstract machines are difficult to describe clearly; witness the profusion of nearly incomprehensible descriptions of the WAM. In hopes of adding clarity to this document, the AAM instructions are introduced gradually as required to handle more complex processing. In addition, the text attempts to explain *why* each implementation detail is necessary.

3.1 What is an abstract machine?

Abstract machines provide an *operational* definition of an algorithm or process. In published papers, a process is often described in terms of data flow, set theory or with a series of theorems and lemmas. Implementing these high-level descriptions is often difficult because they are at a level of abstraction far removed from the machine level of memory accesses, register stores, and elementary computations.

An abstract machine provides a description of a process in terms of registers and atomic instructions, but in the language of the process. The "machine" component of an abstract machine can be described like a standard processor with registers, stacks, and heaps, but the data elements manipulated by the machine are process-specific (e.g. frames and rules) rather than hardware-specific (e.g. bytes). Implementation of a system based on an abstract machine is thus easier because the processing details are explicit and the implementor need only translate process-specific data structures and operations into programming language-level constructs.

In addition, the level of abstraction provided by an abstract machine allows machine-independent implementations of systems. For example, a rule compiled to abstract machine code can execute on any implementation of the abstract machine. The clarity of the implementation and the virtues of portability make abstract machines a popular method for implementing systems that must run on a wide variety of machines. Examples include the programming languages Algol68 [van Wijngaarden et al, 1975], Prolog [Warren, 1977], Smalltalk [Goldberg, 1986] and Java [Lindholm and Yellin, 1996]

One of the main components of an abstract machine is its set of instructions, which approximate an assembly language for a hardware-based machine. A compiler or translator is provided to translate higher-level constructs such as rules into the abstract machine language. The compiler's output, combined with the definition of each abstract machine instruction, defines the operational semantics of the higher-level operation.

The Algernon Abstract Machine (AAM) utilizes LISP-like data elements (symbols and lists) to represent rules and binding lists, much like a typical computer uses bits and bytes to represent strings and structures. The basic AAM operations manipulate data elements or perform knowledge-base store and retrieval operations. Consequently, the AAM is more easily implemented in symbol-processing languages such as LISP.

3.2 Architecture of the AAM

The AAM utilizes LISP as its base language and requires a knowledge base management system, called here the Simple Frame System (SFS). The path compiler and instruction processor each access both the SFS and LISP, as shown in Figure 3.1. The Simple Frame System is completely independent of the AAM and is accessed via an interface defined in Appendix D.

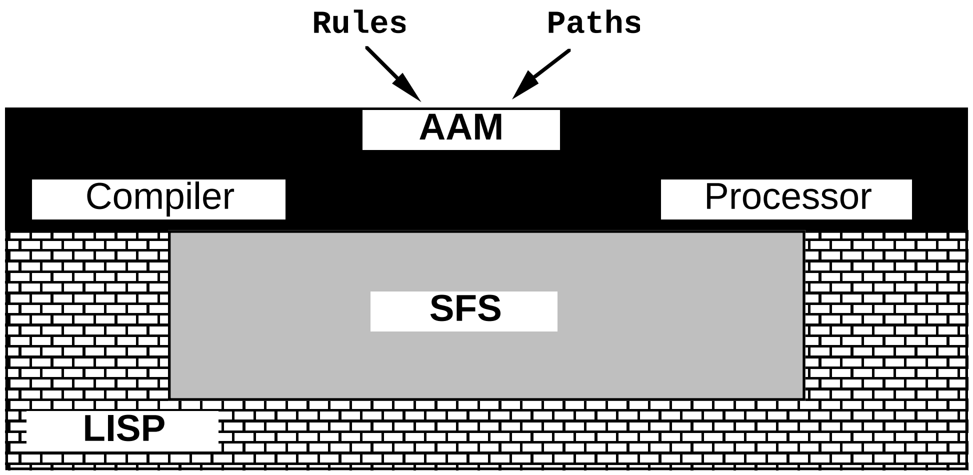

Fig. 3.1. The architecture of the Algernon Abstract Machine.

Internally, the AAM has a set of registers, each of which contains a pointer to a symbc data structure that is stored in a heap. In the implementation of the AAM described in this port, the heap corresponds to the LISP memory area and is managed by LISP. The heap area the SFS is also managed by LISP.

3.2.1 Registers

The AAM contains several registers; the important ones are the CLAUSE, BINDINGS, ACTIONS, RESULTS, CODE, ACTIVATE-IA and ACTIVATE-IN registers. A few other registers, INDEX, RULE, and KEY contain information useful for the user but not essential to the functioning of the machine. A CONTEXT register is reserved for implementation of a context mechanism. Figure 3.2 describes the contents of the important registers.

register	contents
`ACTIONS`	*Queue of pending actions*
`BINDINGS`	*Stack of variable bindings*
`CLAUSE`	*The clause being processed*
`CODE`	*AAM instructions to be processed*
`RESULTS`	*Bindings from successful paths*
`ACTIVATE-IA`	*Clause used to activate if-added rules*
`ACTIVATE-IN`	*Clause used to activate if-needed rules*

Figure 3.2. Registers in the abstract machine.

3.2.2 Initializing the AAM

When the AAM is reset, the memory and frame system are cleared and loaded with an initial knowledge base, as described below. At the beginning of each reasoning sequence, the abstract machine is initialized with the path and code to be processed and the initial binding set. Figures 3.3 and 3.4 illustrate the reset and initialization processes.

reset-machine

```
-- initialize registers
{ACTIONS, BINDINGS, CLAUSE, CODE,
   RESULTS, ACTIVATE-IA, ACTIVATE-IN} ← NIL;

-- initialize the knowledge base
-- (see Appendix C for details of the core KB)
(SFS:RESET)
(load "algy-core-kb");
```

Fig. 3.3. Instructions to reset the abstract machine.

initialize-machine(code, bindings)

```
-- initialize registers
CODE ← code;
BINDINGS ← bindings;
```

Fig. 3.4. Instructions to initialize the abstract machine.

3.3 Access-limited logic forms compiled by the AAM

The AAM compiler, described in Chapter 4, accepts as input an access path as defined in Chapter 2, or a rule whose antecedent and consequent are both access paths. It also accepts a processing mode, *query* or *assert*. The compiler returns a list of AAM instructions that, when processed, will execute the path or rule. Using a relatively simple variable analysis the compiler can enforce the restrictions of access limitation at compile time. It can also do type verification of slot accesses at compile time. Both of these features speed up and simplify the execution model of the abstract machine.

3.4 Compiling and executing ALL forms

This section introduces the core set of AAM instructions by way of several examples. The input to the AAM processing sequence is a *path*, a *mode* (ASSERT or QUERY), and an optional *binding list*. The following subsections detail how each type of path is compiled and executed, introducing new abstract machine instructions as necessary. Appendix A contains formal descriptions of all of the AAM instructions.

3.4.1 Bindings, bindings lists, and binding sets

Before proceeding to the instructions, we must precisely define the format in which results of operations are stored. A ***variable binding*** consists of a variable and its binding, which is a frame or a lower-level data structure. In our list-based data representation, we store bindings as a dotted pair. Some examples of bindings:

```
(?x . oak-tree)
(?mom . Hillary)
(?size . 17)
```

A ***binding list*** is a collection of bindings, containing one binding for each variable in the list. The representation is an association list: a collection of dotted-pairs. An example:

```
((?x . oak-tree)
 (?mom . Hillary)
 (?size . 17))
```

A ***binding set*** is a collection of binding lists. The representation is a list of binding lists, with the constraint that no binding list is duplicated. A binding set is generated when a query returns multiple values for a variable binding. For example, the query `(parent John ?p)` might return two values for `?p`, `Bill` and `Mary`. The binding set created from these return values would contain two binding lists:

```
(((?p . Bill))
 ((?p . Mary)))
```

A subsequent query of `(parent ?p ?gp)` to find the grandparents of John would produce a binding set containing four binding lists:

```
(((?p . Bill) (?gp . Carl))
 ((?p . Bill) (?gp . Connie))
 ((?p . Mary) (?gp . Doug))
```

```
    ((?p . Mary) (?gp . DeAnn))
    )
```

Each binding list in a binding set corresponds to a branch of the search tree. So when an action is activated from the queue, its binding set contains just one binding list, since an activation represents one branch of the tree. As further queries are processed, more binding lists are generated and the binding set grows.

The `BINDINGS` register always contains a binding set. Results are merged into it, activations are created from it, and substitutions are made from it.

3.4.2 Rule closures and activations

Rules to be activated are stored in the knowledge base as *rule closures*, consisting of a rule name and a binding list. The rule name is the name of a frame that contains the rule's antecedent, consequent and compiled code. A rule continuation is simply a rule closure whose rule is a partial rule constructed from the antecedents and consequent of a rule provided by the user.

Rules pending execution are contained in *rule activations*. A rule activation consists of a set of AAM registers and their contents. When a rule closure is first activated, it provides the contents of the `CODE` and `BINDINGS` registers; the other registers contain `NIL`.

3.4.3 Ground assert or verify

The `:ASSERT` and `:VERIFY` instructions operate on the clause in the `CLAUSE` register, so we must introduce an instruction, `:CLAUSE`, that stores an item in the register.

`:CLAUSE <clause>`

Stores <clause> in the `CLAUSE` register.

We also introduce the `:ASSERT` and `:VERIFY` instructions that perform the assert or verify.

`:ASSERT`

Uses the SFS command `KB-PUT-VALUE` to assert the clause in the `CLAUSE` register into the knowledge base. The return value from `KB-PUT-VALUE` is pushed onto the `BINDINGS` register. The contents of the `CLAUSE` register are copied to the `ACTIVATE-IA` register (see Section 3.5.1).

The result value can be `T`, meaning the assertion was successful; `:KNOWN`, meaning the assertion was already in the knowledge base; or `NIL`, meaning that the assertion was unsuccessful. An assertion will fail if the slot or frame was nonexistent or if the slot was full (as defined by its `cardinality` and the current contents of the slot).

:VERIFY

Uses the SFS command `KB-GET-VALUES` to check whether the clause in the `CLAUSE` register is already in the knowledge base. The result is pushed onto the `BINDINGS` register.

The result value can be `T`, meaning the clause was present in the knowledge base; or `NIL`, meaning that it wasn't.

The compiled form of a ground assertion such as `(parent John Mrs-Smith)` is thus:

```
:CLAUSE      (parent John Mrs-Smith)
:ASSERT
```

The compiled form of a verify operation has a similar structure.

3.4.4 Ground query

A simple query, such as `(mother John ?mom)` utilizes similar instructions. The difference is that the query produces a set of bindings for the variable(s) in the query. The binding set is pushed onto the `BINDINGS` register by the `:QUERY` command.

:QUERY

Uses the SFS command `KB-GET-VALUES` to retrieve all known instantiations of the clause in the `CLAUSE` register. A binding set is produced from the results and is pushed onto the `BINDINGS` register. If the binding set is `NIL`, the query was unsuccessful.

We also need some instructions to handle binding sets. Since clause processing is formed in the context of path processing, sometimes when a clause is processed a bindin will be available in the `BINDINGS` register. At other times there will be none. Thus, we nee commands to both create and manage binding lists.

First, we have a general instruction, `:PUSH`, to push a value into a register. The `BINDINGS` register is the primary recipient of these values.

:PUSH <register> <value>

Pushes the value onto the given register.

Using the above instructions, a ground query can be expressed as:

```
:PUSH        :BINDINGS    NIL
:CLAUSE      (parent John ?who)
:QUERY
```

A binding set resulting from a query is combined with the existing binding set so that the com bined results can be used with the next clause. The algorithm for binding set merge is de scribed in Appendix B. After each query then, we must mer ge binding lists:

:MERGE-BINDINGS

Pops the top two binding sets from the `BINDINGS` register and merges them into one binding set, which is pushed onto the `BINDINGS` register.

The basic instruction stream for a query is thus:

```
:PUSH          :BINDINGS    NIL
:CLAUSE        (parent John ?who)
:QUERY
:MERGE-BINDINGS
```

Note that `:ASSERT` instructions are not followed by a `:MERGE-BINDINGS` instruction because they do not generate new binding sets.

Values from binding lists are substituted for variables in clauses before the assertion or query is performed. The AAM instruction to perform this substitution is `:SUBST`. The compiler inserts a `:SUBST` command before every `:ASSERT` or `:QUERY` which it has determined will contain unbound variables at runtime.

:SUBST

Substitutes the values in the topmost binding list into the current clause.

```
:CLAUSE        (parent John ?who)
:SUBST
:QUERY
:MERGE-BINDINGS
```

3.4.5 Terminating a reasoning stream

In practice, the calling program wants a value returned from the machine. The two most useful return values are (1) an indication of the success of the operation; and (2) the binding set created by processing the path. Thus the `:RETURN` command signals the end of a reasoning stream and the need to return some values. It is executed at the end of each path.

:RETURN <return-type>

This instruction always sets the `CODE` register to `NIL`. In addition it does one of the following:

If return-type is `:SUCCESS`, it pushes a new dotted pair onto the `RESULTS` register. The first element of the pair is either `T` or `NIL`, signifying the success or failure of the operation. The second element is the binding set produced by the path.

If return-type is `:FAILURE` it pops the `BINDINGS` register (since the path failed, the bindings are useless).

If return-type is `:NOTHING`, no result is saved.

A standard code sequence is placed at the end of every compiled path. In most cases, th following sequence allows for both failure (see Section 3.4.7) and success exits. In compiled rules, the sequence allows for either failure or "nothing" exits, which simply stop execution. The standard path ending code sequence is:

```
          ...
          ...
   :SUCC  :RETURN  :SUCCESS
   :FAIL  :RETURN  :FAILURE
```

while the standard rule path ending code sequence is:

```
          ...
          ...
   :END   :RETURN  :NOTHING
   :FAIL  :RETURN  :FAILURE
```

The `:SUCC`, `:END`, and `:FAIL` labels are another AAM instruction, which has the form:

`:LABEL <label>`

Provides a destination for a jump command. If encountered during processing, acts as a no-op (i.e. it does nothing).

When printing AAM code, the `:LABEL` is usually dropped and the label itself is printed before the succeeding line, unless it is also a label.

3.4.6 Path processing

We have now defined all of the instructions needed to process simple non-brar paths. The input consists of a *path*, a *mode*, and a (possibly empty) *binding set*. The output success value and a set of binding lists. The AAM instructions for a compiled assertion or query typically include a step to initialize the `BINDINGS` register; an assert, query, or verify for every clause in the path; a command to merge bindings after every query; and a terminating `:RETURN` to prepare the return value for the caller. Some user-level functions are helpful in preparing the input and collecting the output. The functions `TELL` and `ASK` accept a path, call the compiler to convert the path to AAM code, call the processor to process the AAM code, and present the output to the user. Below is the `TELL` command. `ASK` is very similar.

```
(defun TELL (path)
  "Asserts the clauses in the path and presents the results."

 (let (values-and-bindings)
    (setq values-and-bindings
      (aam:aam-process (aam:aam-compile path :ASSERT NIL)
                 NIL))

    (cond (values-and-bindings
           (format T "~2%TELL succeeded.")
           (dolist (result values-and-bindings)
             (show-binding-set (cdr result))))
          (T
           (format T "~2%TELL failed.")))))
```

The function `AAM-PROCESS` executes the compiled code and returns the contents of the `RESULTS` register. It is a relatively simple loop that steps through the instructions and executes each one sequentially.

3.4.7 Disjunctive branching

Consider the following path, which asserts that John is the child of each of his parents:

```
(tell '((parent John ?parent)
        (child ?parent John)))
```

The alert reader will wonder how the child relation is asserted for both parents. Remember that in assert mode a non-ground clause is treated as a query. So the first query will typically return at least two bindings for the variable `?parent`. In *ALL*, any succeeding clauses should be executed once for each binding. Therefore, an implicit branch occurs in the reasoning process at that point. The AAM instruction to perform a branch is `:BRANCH`.

`:BRANCH`

For every binding list in the topmost binding set of the `BINDINGS` register, create an activation from the currently executing code and add it to the `ACTIONS` register. If the `BINDINGS` register is empty, create an activation with an empty binding list and place it in the `ACTIONS` register.

In addition, clear the `CODE` register to force one of the activations in the queue to be selected for execution.

When the activation resumes execution, it begins with the instruction following the `:BRANCH`.

After each query, the AAM must check for a branch opportunity. Since only the `:QUERY` instruction can produce multiple return values, the compiler inserts a `:BRANCH` instruction after each query, but not after an assert. The compiled form of the above TELL is now:

```
          :PUSH    :BINDINGS    NIL
          :CLAUSE  (parent John ?parent)
          :QUERY
          :MERGE-BINDINGS
          :BRANCH
          :CLAUSE  (child ?parent John)
          :SUBST
          :ASSERT
    :SUCC :RETURN  :SUCCESS
    :FAIL :RETURN  :FAILURE
```

Note that the compiler inserts a :SUBST instruction only when compiling the second clause. It can determine at compile time that the binding set will be empty when the first clause is processed, so there will be no need to do variable substitution then.

The :SELECT instruction selects an activation from the ACTIONS register and restores the registers from its contents. The AAM processor will then begin executing the code from that activation.

:SELECT

Selects an activation from the ACTIONS register and restores the AAM registers from the information in it. By default it selects the topmost activation, but it can be programmed to select the actions in any order as determined by a control scheme.

Since the processor does an automatic :SELECT when it encounters an empty CODE register, the compiler rarely generates :SELECT statements.

3.4.8 Conjunctive branching

The :BRANCH instruction described above implements an OR branch–the path succeeds if any branch succeeds. Algernon has an :ALL-PATHS operator which implements AND branching. It succeeds if and only if all of its branches succeed. To implement conjunctive branching, we need two additional instructions, :CONJ-BRANCH and :CONJ-CLEAR. The first is very similar to :BRANCH except that it tags each activation with a unique (group) ID tag.

:CONJ-BRANCH

Like :BRANCH, it creates a set of activations, one for each binding in the current binding set. It tags these activations with a common, unique ID. If any branch fails, the ID is used to find and remove all of the remaining branches from the ACTIONS register.

If any branch of the conjunction should fail, the conjunction fails and the remaining branches do not need to be processed. The :CONJ-CLEAR instruction removes any activations in the ACTIONS register that have the same ID as the instruction that failed.

:CONJ-CLEAR <tag>

Removes any activations in the ACTIONS register that have the same ID tag as a conjunctive branch that failed.

3.4.9 Handling failure

If a reasoning branch fails, for example by being unable to retrieve any values from a query, processing of that branch should terminate. For this reason, the compiler includes a `:FAIL?` instruction after each assert, query or verify.

:FAIL? <label>

If the last operation failed (signified by a `NIL` on top of the `BINDINGS` register), it pops the `BINDINGS` register and jumps to the given label (usually the label called `:FAIL`). If the last operation was successful and returned a set of bindings, this instruction does nothing. If the last operation was successful and returned `T`, the `T` value is removed from the `BINDINGS` register.

The `:FAIL?` instruction checks the `BINDINGS` register for the result of the last operation. If it failed, the result is removed from the `BINDINGS` register and the processor jumps to the given label. Often this is a label representing a failure exit from the path.

As an example that illustrates branching and failure, consider a query that retrieves all of the grandfathers of Arnold:

```
(ask '((parent Arnold ?parent)        ; Query 1
       (parent ?parent ?g-parent)     ; Query 2
       (gender ?g-parent male)))      ; Query 3
```

This query should return all bindings of `?parent` and `?g-parent` that satisfy all three queries. The reasoning tree is shown below:

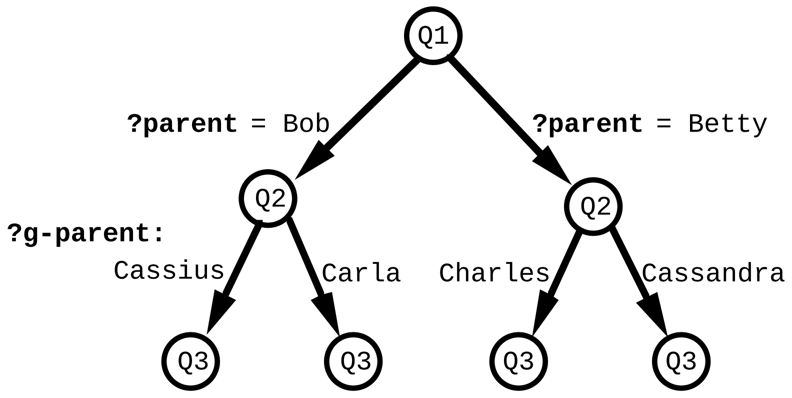

Fig. 3.7. Search tree for retrieving grandfathers.

Therefore, the compiled form should branch twice and check for failure after queries, as showr in the compiled code below:

```
                    :PUSH:BINDINGS  NIL
                    :CLAUSE    (PARENT ARNOLD ?PARENT)
                    :QUERY
                    :FAIL?          :FAIL
                    :MERGE-BINDINGS
                    :BRANCH

                    :CLAUSE    (PARENT ?PARENT ?G-PARENT)
                    :SUBST
                    :QUERY
                    :FAIL?          :FAIL
                    :MERGE-BINDINGS
                    :BRANCH

                    :CLAUSE    (GENDER ?G-PARENT MALE)
                    :SUBST
                    :QUERY
                    :FAIL?          :FAIL
                    :MERGE-BINDINGS

          :SUCC     :RETURN    :SUCCESS
          :FAIL     :RETURN    :FAILURE
```

Assuming that the knowledge base contains relevant information about the family tree, the trace below shows the AAM processing the above code.

```
Ask: ((parent Arnold ?parent)        ; Query 1
      (parent ?parent ?g-parent)      ; Query 2
      (gender ?g-parent male)))       ; Query 3

  Query succeeded.                   ; Query 1
  Query succeeded.                   ; Query 2, branch 1
  Query succeeded.                   ; Query 3, branch 1
  Returning success,
      bindings = ((?parent . bob) (?g-parent . Cassius))

  Query failed.                      ; Query 3, branch 2

  Query succeeded.                   ; Query 2, branch 2
  Query succeeded.                   ; Query 3, branch 3
  Returning success,
      bindings = ((?parent . betty) (?g-parent . Charles))

  Query failed.                      ; Query 3, branch 4

ASK succeeded.
   Bindings:
          ?parent      = Bob
          ?g-parent    = Cassius

          ?parent      = Betty
          ?g-parent    = Charles
```

3.4.10 Examples of path processing

Let's take a look at the code generated and processed for some simple paths. Note that these paths do no branching, and the code has only limited failure checking. First, a simple one-clause assert:

```
(TELL '((mother Arnold Betty))

      :PUSH         :BINDINGS      NIL
      :CLAUSE       (mother Arnold Betty)
      :ASSERT
      :FAIL?        :FAIL
:SUCC :RETURN       :SUCCESS
:FAIL :RETURN       :FAILURE
```

and the corresponding query:

```
(ASK '((mother Arnold ?mom)))

      :PUSH         :BINDINGS      NIL
      :CLAUSE       (mother Arnold ?mom)
      :QUERY
      :FAIL?        :FAIL
      :MERGE-BINDINGS
:SUCC :RETURN       :SUCCESS
:FAIL :RETURN       :FAILURE
```

A multi-clause assert:

```
(TELL '((mother Arnold Betty)
         (father Arnold Bob)))

      :PUSH         :BINDINGS      NIL
      :CLAUSE       (mother Arnold Betty)
      :ASSERT
      :FAIL?        :FAIL
      :CLAUSE       (father Arnold Bob)
      :ASSERT
      :FAIL?        :FAIL
:SUCC :RETURN       :SUCCESS
:FAIL :RETURN       :FAILURE
```

and a combination assert and query:

```
(TELL '((mother Arnold Betty)
         (father Arnold ?dad)

      :PUSH         :BINDINGS      NIL
      :CLAUSE       (mother Arnold Betty)
      :ASSERT
      :FAIL?        :FAIL
      :CLAUSE       (father Arnold ?dad)
      :QUERY
      :FAIL?        :FAIL
      :MERGE-BINDINGS
:SUCC :RETURN       :SUCCESS
:FAIL :RETURN       :FAILURE
```

3.5 Rule Activation

The AAM supports both forward-chaining and backward-chaining rules. Forward-chaining rules, called *if-added rules*, are activated when an assertion unifies with the *key clause* of an if-added rule. Except for continuations, which are described in Section 3.6, the key clause of an if-added rule is the first clause of its antecedent.

Backward-chaining rules, called *if-needed rules*, are activated when a query unifies with the key clause of an if-needed rule. The key clause of an if-needed rule is the first clause of its consequent.

3.5.1 Activating if-added rules

If-added rules are activated after a successful *new* assertion is made. An assertion is new if it was not previously known in the knowledge base. Therefore, after an assertion is made we not only need to check for failure with the `:FAIL?` instruction, we also need to check whether the assertion was already known. We do this using the `:KNOWN?` instruction.

`:KNOWN? <label>`

If the last assertion was known (signified by a `:KNOWN` value on top of the `BINDINGS` register), it pops the `BINDINGS` register and jumps to the given label. If the last operation was successful and returned a set of bindings, this instruction does nothing. If the last operation was successful and returned T, the T value is removed from the `BINDINGS` register.

This instruction is normally used to branch around an instruction that performs if-added rule activation.

The rule activation itself is performed by the `:ACTIVATE-IA` instruction.

`:ACTIVATE-IA`

Activates appropriate if-added rules based on the last clause recorded in the `ACTIVATE-IA` register. Rules are retrieved, unified with the clause and, if successfully unified, an activation is created and placed in the `ACTIONS` register.

The first example of Section 3.4.6 shows the compiled form of a simple assertion. The final addition to its compiled form is to include rule activation as follows:

```
 (TELL '((mother Arnold Betty))

       :PUSH          :BINDINGS      NIL
       :CLAUSE        (mother Arnold Betty)
       :ASSERT
       :KNOWN?        :L-1
       :FAIL?         :FAIL
       :ACTIVATE-IA
       :BRANCH
:L-1   …
:SUCC  :RETURN        :SUCCESS
:FAIL  :RETURN        :FAILURE
```

Note that the `:ACTIVATE-IA` instruction is followed by a `:BRANCH` instruction. A branch occurs whenever a query returns more than one value, or whenever one or more rules have been activated. This allows the most appropriate action on the action queue to be selected for execution. Recall that `:BRANCH` places a copy of the current execution environment on the action queue and then empties the `CODE` register, thereby ensuring that a `:SELECT` statement will be executed to select an action from the queue.

3.5.2 Activating if-needed rules

If-needed rules are activated before a query is attempted. The rules are executed first so that they can generate information in the KB to be retrieved by the query. In terms of lines of reasoning, this means that the current line of reasoning must be suspended (put on the queue) until rule activation has been performed and the activated rules have been executed. Normally, this is done with a `:BRANCH` instruction, but that would leave no way for a rule activation instruction to execute. Therefore, we introduce the `:SAVE` instruction, which is a modified form of a `:BRANCH` instruction.

:SAVE

This instruction is like `:BRANCH` except that it doesn't clear the `CODE` register. It is used only before an `:ACTIVATE-IN` instruction. It also copies the current contents of the `CLAUSE` register into the `ACTIVATE-IN` register.

The rule activation itself is performed by the `:ACTIVATE-IN` instruction. A `:SELECT` instruction after the rule activation causes the activated rules to execute.

:ACTIVATE-IN

Activates appropriate if-needed rules based on the clause to be queried (and recorded in the `ACTIVATE-IN` register). Rules are retrieved, unified with the clause and, if successfully unified, an activation is created and placed in the `ACTIONS` register.

The third example of Section 3.4.6 shows the compiled form of a simple query. The final addition to its compiled form is to include rule activation as follows:

```
(ASK '((mother Arnold ?mom)))

      :PUSH          :BINDINGS      NIL
      :CLAUSE        (mother Arnold ?mom)
      :SAVE
      :ACTIVATE-IN
      :SELECT
      :QUERY
      :FAIL?         :FAIL
      :MERGE-BINDINGS
:SUCC :RETURN        :SUCCESS
:FAIL :RETURN        :FAILURE
```

3.6 Compiling rules

Rules are compiled by compiling the antecedent in query mode and the consequent in assert mode and concatenating the two code segments together. There are two differences between a normal compiled path and a compiled rule. First, the compiled rule either fails, or succeeds and returns "nothing". This means that after executing a rule, the processor proceeds in a seamless fashion to the first instruction of the next action.

The second difference is that an executing rule creates continuations for every antecedent. A continuation, described fully in Section 3.7, is a partial rule consisting of the remaining antecedents and the bindings created from the preceding antecedents. If an assertion is made that satisfies the first of the remaining antecedents, the rule will continue executing from the point where the continuation was made.

In order to efficiently create continuations at runtime, all of the possible partial rules are compiled when the original rule is compiled. If the rule being compiled is Rule57, it may have several continuation rules named Rule57-002, Rule57-003, etc. The appended number denotes the clause where the continuation will begin execution if activated.

3.6.1 Compiling backward chaining rules

Backward chaining rules present one unique problem for access limitation. For a slot with *n* arguments, it is unknown at compile time which of the *n* arguments will be bound by an activating query at runtime. Access limitation states only that the frame and slot will be bound. The code for the compiled rule may depend on which variables are bound or not.

To handle this situation, the compiler generates up to four versions of the compiled rule, corresponding to the situations in which zero, one, two or three arguments are bound. If another calling configuration arises at runtime, the compiler is called at that time to generate an appropriate version of the compiled rule.

3.7 Rule continuations

Rule continuations are an unusual feature of Algernon. They allow the system to guarantee independence of results from the order of assertions. For example, suppose that a rule has three antecedent clauses and that when it activates, the first two are satisfied while the third can not be. Sometime later, when an assertion is made that satisfies the third clause, the rule should resume processing. In many rule-based systems, this will not happen. In Algernon, a rule continuation will exist for the third clause. Upon asserting a clause that satisfies the third antecedent, the continuation will activate and complete execution of the rule.

Rule continuations actually correspond to an inline version of the RETE activation network. They allow rules to fire independent of the order of assertions.

In order for continuations to be created at the appropriate times, a compiled rule has `:CONTINUATION` instructions at the beginning of every clause processing phase.

`:CONTINUATION <continuation-rule-name>`

Creates one or more closures consisting of the continuation rule and
binding list from the current set of bindings and stores them in the KB
future activation.

3.8 Special forms

Algernon has numerous operators that must be handled by the compiler. Some of these can be compiled into the set of instructions described above, but others need their own instruction. The AAM recognizes and implements all of the standard Algernon operators. The instructions needed to handle them are:

`:ANY`

Implements the `:ANY` operator of Algernon. If there is more than one binding list in the topmost binding-set of the BINDINGS register, pick one at random and throw the rest away.

`:ASK`

Implements the `:ASK` operator of Algernon. Queries the user, accepts input and either binds a variable or returns a value as specified by the definition of `:ASK`.

`:BIND <vars> <expr>`

Implements the `:BIND` operator of Algernon. Binds one or more variables to the result of an expression.

`:BOUNDP`

Succeeds if the given variable is currently bound.

`:BRANCH-USER`

Implements the `:BRANCH` operator of Algernon, which is different from the `:BRANCH` instruction in the AAM. Unifies the given variables with the result of evaluating the given expression. Creates a binding list for each unification and places the resulting binding set on the BINDINGS register.

`:CLEAR-SLOT`

Implements the :CLEAR-SLOT operator of Algernon. Deletes all values stored in the frame-slot specified by the current contents of the `CLAUSE` register

`:DELETE <frame> <slot> <value>`

Implements the `:DELETE` operator of Algernon. Deletes the frame-slot-value specified by the current contents of the `CLAUSE` register.

`:EVAL <expression>`

Implements the `:EVAL` operator of Algernon. Calls the underlying language (usually LISP) to evaluate the given expression. This operator always succeeds and places no result on the BINDINGS register.

`:KB-DEF-FRAME <var>`

Creates a frame in the KB whose name is computed according to the current clause. The frame name becomes the binding for `var` and is added to the binding set.

`:NEQ <value1> <value2>`

Implements the `:NEQ` operator of Algernon. If the two arguments of the clause are not EQ, the clause succeeds.

`:POP <register>`

The complement of the previously-described :`PUSH` instruction.

`:RULES <class> <rule1> <rule2>…`

Implements the `:RULES` operator of Algernon. It calls DEF-RULE on its arguments.

`:SHOW`

Displays a frame using the SFS `print` function.

`:SKIP <label>`

Performs an unconditional jump to a label that occurs later in the code.

`:SLOT <name> <domains>`

Calls DEF-SLOT on the slot and declares it a member of the SLOT class.

`:SRULES <slot> <rule1> <rule2>…`

Implements the `:SRULES` operator of Algernon. It calls DEF-RULE on its arguments.

`:STOP`

Halts AAM processing and exits from the top-level call to the AAM processor.

`:SUBR <var> <code>`

Executes the given AAM `code` in a recursive call to the AAM processor. If `var` is bound by the code, its binding(s) are retained and available for use in subsequent clauses. If no variable is given, all results are retained.

:SUBR is used to implement many Algernon instructions, especially those containing complex forms, such as `:OR`, and `:ALL-PATHS`.

`:SUCCEED? <label>`

The opposite of `:FAIL?`, it jumps to the given label if the previous operation succeeded.

`:TAXONOMY`

Implements the `:TAXONOMY` operator in Algernon. Defines frames and taxonomic relations at runtime.

`:TEST <expression>`

Implements the `:TEST` operator in Algernon. Succeeds if the expression evaluates to a non-NIL result.

`:UNBOUNDP`

Succeeds if the given variable is not bound.

`:UNIQUE? <var> <label>`

Implements the `:THE` operator in Algernon. If `var` has exactly one binding, the statement succeeds and jumps to the given label. If the variable has more than one binding, it fails.

`:VERIFY`

Succeeds if the current clause already exists in the knowledge base. Similar to `:QUERY`, but it doesn't activate rules.

4 The AAM compiler

The compiler's goal is to turn a path into a list of AAM instructions. The instructions generated depend on the mode, binding lists (if any) and whether or not a clause is ground. The compiler is highly recursive and is right-recursive on the path so that it generates the instructions in the correct order.

aam-compile

```
(defun aam-compile (path mode bindings
                    &OPTIONAL (compiling-rule? NIL))
  "Returns AAM machine code to be executed.  Bindings only
occur in the case where a form is compiled during execution
of a path.  For example, when a rule is defined during
execution of a rule, or when a rule is compiled at runtime.

Mode is ASSERT or QUERY."

  ;; Error checking goes here - make sure the arg is a path.
  (let ((the-path   (preprocess path))
        (index      1))

    (compile-path the-path index mode bindings
                  (mapcar #'car (car bindings))
                  compiling-rule?)))
```

compile-path

```
(defun compile-path (path index mode bindings
                     bound-variables
                     &OPTIONAL (compiling-rule? NIL))
  "Compiles a path."

  (append
   (unless *no-bindings*            ;; Used by :SUBR
     `(:PUSH :BINDINGS ,bindings))
   (compile-path-aux path index mode bound-variables
                     (if compiling-rule?
                         (list :LABEL :END  :RETURN :NOTHING
                               :LABEL :FAIL :RETURN :FAILURE)
                       ;;else
                       (list :LABEL :SUCC :RETURN :SUCCESS
                             :LABEL :FAIL :RETURN :FAILURE))
                     compiling-rule?)))
```

The topmost functions, shown above, set up the call to `compile-path-aux`, which recursively calls itself to compile the given path.

compile-path-aux

```
(defun compile-path-aux (path index mode bound-variables
                         code
                         &OPTIONAL (compiling-rule? NIL))
  "Recursive form of compile-path."

  (if (null path)
      code
    ;;ELSE
    (if (null (cdr path))
        (compile-clause (car path) index mode bound-variables
                        code :LAST-CLAUSE compiling-rule?)
      ;;ELSE
      (compile-clause (car path) index mode bound-variables
                      (compile-path-aux (cdr path) (1+ index)
                         mode (union (variable-arguments
                                         (car path))
                                      bound-variables)
                         code compiling-rule?)
                      NIL compiling-rule?))))
```

`Compile-path-aux` detects whether it is compiling the last clause or not. The last clause does not need a `:BRANCH` statement because there will be no further processing of this branch of the search tree.

The `compile-clause` function checks for violation of access limitation using the list of bound variables that is constructed in the above functions. It also checks that slot domains have the correct type by propagating types among the variables.

It then calls `compile-query` or `compile-assert` depending on the current mode and whether or not the clause is ground. Asserts and queries are compiled as described in Section 3. It may also call a special-purpose function to compile one of Algernon's special forms such as `:ASK`.

5 Examples of access-limited logic program execution

In this section we show an example of execution of an Algernon program. We show the results of compiling each Algernon path and illustrate the register changes caused by each command.

5.1 Defining the taxonomy

In an Algernon program, the first operations performed are additions to the standard taxonomy. In the example of this section, we will define the classes `People` and `Vehicles`, with relevant subclasses and instances:

```
(tell '((:TAXONOMY (Things
                      (People Jeff Karen)
                      (Vehicles
                            (Automobiles montero accord)
                            (Motorcycles pacific nighthawk))
                      (Colors blue red white)
                      ))))
```

In Algernon, `:TAXONOMY` has a fairly complex Algernon expansion, and the corresponding AAM implementation is just as complex. We will show just the code for defining the `People` class. First, Algernon forms the expression `(:THE ?x (name ?x "People"))`, which either retrieves the predefined class named `People`, creates a new one, or fails if there is more than one class named `People`. It then asserts that the frame `People` is a subclass of `Sets`, and that it is a taxonomic descendant of `Things`. The equivalent Algernon code for the definition of `People` is shown below:

```
(tell '((:THE ?x (name ?x "People"))
        (isa ?x Sets)
        (imp-superset ?x Things)))
```

The corresponding AAM code is shown in the box below. Similar code is produced for each new class or instance in the taxonomy definition. Lines 1-2 use a `:SUBR` to search for a frame named "People". The `:SUBR` is a recursive call to the AAM processor using the supplied code. If it fails, processing proceeds to label `:L-533` at line 6. If it succeeds with a binding to the argument variable `?$523`, the binding is checked for uniqueness at line 4. If it is unique, the code jumps to label `:L-534` at line 17. If the search for `People` succeeded, but not uniquely, the path fails in line 5 and jumps to the `:FAIL` label at the end of the compiled code (not shown).

If the frame `People` did not previously exist, lines 6-14 create a new frame, assert its name, check for failure, and activate if-added rules as needed.

```
 1        :PUSH           :BINDINGS  NIL
 2        :SUBR           ?$523      (:CLAUSE (NAME ?$523 "PEOPLE") :SAVE
                                      :ACTIVATE-IN :SELECT :QUERY :FAIL?
                                      :FAIL :MERGE-BINDINGS :LABEL :SUCC
                                      :RETURN :SUCCESS :LABEL :FAIL
                                      :RETURN :FAILURE)
 3        :FAIL?          :L-533
 4        :UNIQUE?        ?$523      :L-534
 5        :SKIP           :FAIL
 6  :L-533 :CLAUSE        (:A ?$523 (NAME ?$523 "PEOPLE"))
 7        :KB-DEF-FRAME   ?$523
 8        :MERGE-BINDINGS
 9        :CLAUSE         (NAME ?$523 "PEOPLE")
10        :SUBST
11        :ASSERT
12        :KNOWN?         :L-535
13        :FAIL?          :FAIL
14        :ACTIVATE-IA
15        :BRANCH
16  :L-535
17  :L-534 :MERGE-BINDINGS
18        :CLAUSE         (ISA ?$523 SETS)
19        :SUBST
20        :ASSERT
21        :KNOWN?         :L-536
22        :FAIL?          :FAIL
23        :ACTIVATE-IA
24        :BRANCH
25  :L-536 :CLAUSE        (IMP-SUPERSET ?$523 THINGS)
26        :SUBST
27        :ASSERT
28        :KNOWN?         :L-537
29        :FAIL?          :FAIL
30        :ACTIVATE-IA
31        :BRANCH
```

Lines 17-24 assert that `People` is a subclass of `Sets`, including checks for failure and if-added rule activation. Lines 25-31 do the same for a relation between `People` and `Things`.

The `:SUBR` instruction used above is needed because we need to know the result of processing a sub-path, but the result should not be retained permanently. The solution is to run it in a separate instance of the AAM virtual machine and note, but not store, the result.

5.2 Defining relations

After the taxonomy is defined, the relations between frames should be declared. Remember that a relation in logic notation corresponds to a slot in frame notation. The definitions in Algernon are of slots:

```
(tell '((:SLOT owns   (People Vehicles))
        (:SLOT drives (People Vehicles))
        (:SLOT color  (Things Colors))))
```

The AAM code to assert the above Algernon path is much simpler than the taxonomy description above. Most of the work is done by the `:SLOT` instruction, which implements the Algernon `:SLOT` operator using knowledge base functions. The code is straightforward and presented without comment below:

```
1          :PUSH            :BINDINGS  NIL
2          :CLAUSE          (:SLOT OWNS (PEOPLE VEHICLES))
3          :SLOT
4          :CLAUSE          (:SLOT DRIVES (PEOPLE VEHICLES))
5          :SLOT
6          :CLAUSE          (:SLOT COLOR (THINGS COLORS))
7          :SLOT
8   :SUCC   :RETURN          :SUCCESS
9   :FAIL   :RETURN          :FAILURE
```

5.3 Defining rules

First we'll define a simple forward-chaining rule that states "*Any person who drives an Accord drives a white one*."

```
(tell '((:RULES People
          ((drives ?person ?v)
           (isa ?v accord)
           ->
           (color ?v white))
       )))
```

Like the code above for `:SLOT`, the compiled code is uninteresting in that all of the work is done by the `:RULES` instruction using knowledge base functions. However, the compiled code for the rule is worth looking at and is shown below.

```
 1         :CLAUSE          (ISA ?V ACCORD)
 2         :SUBST
 3         :CONTINUATION    RULE22-002
 4         :SAVE
 5         :ACTIVATE-IN
 6         :SELECT
 7         :QUERY
 8         :FAIL?           :FAIL
 9         :MERGE-BINDINGS
10         :BRANCH
11         :CLAUSE          (COLOR ?V WHITE)
12         :SUBST
13         :ASSERT
14         :KNOWN?          :L-633
15         :FAIL?           :FAIL
16         :ACTIVATE-IA
17         :BRANCH
18  :L-633
19  :END   :RETURN          :NOTHING
20  :FAIL  :RETURN          :FAILURE
```

First, note that the initial clause in Line 1 is the second clause of the antecedent. Since the rule is activated by an assertion that matches the first clause, there is no need to process the first clause again. The rule is initially activated with the bindings created by matching the first clause. So Line 2 is a `:SUBST` to substitute those bindings in the second clause. A rule continuation (described in Chapters 2 and 3) is created in Line 3.

Since the first clause processed is a query (because it is in the antecedent), if-needed rules must be activated before the query is made. This is handled in Lines 4-6 where the state of the current path is saved, if-needed rules are activated, and an activation is selected for execution. Finally, the original query is made. Assuming it succeeds, it merges its bindings with the existing bindings (Line 9) and proceeds to the consequent of the rule. Note that there is no line of demarcation to note that the rule passes from the antecedent to the consequent. The only change is that the clauses of the consequent are compiled in assert mode and no continuations are created for the consequent.

In this case, the consequent clause is a fairly simple assertion. Note that if-added rule activation (Line 16) is skipped (Line 14) if the assertion was already in the knowledge base. Note also the `:BRANCH` (Line 17) following the if-added rule activation. As stated above, reasoning may branch because of multiple values returned by a query or because of rules that activate. Finally, in Line 19 successful completion of a rule is signified by an empty `:RETURN`.

Lets define a simple backward chaining (if-needed) rule that states that a person owns a certain kind of motorcycle if they also own a blue Mustang.

```
(tell '((:RULES People
          ((owns ?person nighthawk)
           <-
           (owns ?person ?v)
           (isa ?v mustang)
           (color ?v blue)))))
```

The compiled rule is shown below. Lines 1-10, 11-20 and 21-30 implement clauses one, two and three of the antecedent, respectively. The consequent clause follows immediately in Lines 31-38. Notice that the compiled code for a backward chaining rule is indistinguishable from the compiled code for a forward chaining rule. The difference is in how each is activated.

Other expert system shells create a forward chaining version of every backward chaining rule for completeness. In these systems, every possible deduction will be made, even though a relevant query may not yet have been made. Other systems, such as Algernon, prefer to wait for the query to be made before deducing some facts.

```
        :CLAUSE         (OWNS ?PERSON ?V)
        :SUBST
        :CONTINUATION   RULE23-001
        :SAVE
        :ACTIVATE-IN
        :SELECT
        :QUERY
        :FAIL?          :FAIL
        :MERGE-BINDINGS
        :BRANCH
        :CLAUSE         (ISA ?V MUSTANG)
        :SUBST
        :CONTINUATION   RULE23-002
        :SAVE
        :ACTIVATE-IN
        :SELECT
        :QUERY
        :FAIL?          :FAIL
        :MERGE-BINDINGS
        :BRANCH
        :CLAUSE         (COLOR ?V BLUE)
        :SUBST
        :CONTINUATION   RULE23-003
        :SAVE
        :ACTIVATE-IN
        :SELECT
        :QUERY
        :FAIL?          :FAIL
        :MERGE-BINDINGS
        :BRANCH
        :CLAUSE         (OWNS ?PERSON NIGHTHAWK)
        :SUBST
        :ASSERT
        :KNOWN?         :L-644
        :FAIL?          :FAIL
        :ACTIVATE-IA
        :BRANCH
:L-644
:END    :RETURN         :NOTHING
:FAIL   :RETURN         :FAILURE
```

5.4 Asking questions and making assertions

Since rules are formulated from queries and assertions, we illustrate the AAM instruction set using a pair of forward and backward chaining rules. The rules below use 15 of the 18 core instructions of the AAM (excluding `:VERIFY`, `:SKIP` and `:PUSH`).

A rule to infer Bill's aunts can be written as either a forward-chaining rule or a backw chaining rule:

```
((mother Bill ?mom)      ;;forward-chaining
 (sister ?mom ?aunt)
 ->
 (aunt Bill ?aunt))

((aunt Bill ?aunt)       ;;backward-chaining
 <-
 (mother Bill ?mom)
 (sister ?mom ?aunt))
```

The main difference is how they are activated. The first is activated by asserting a `mother` relation, while the second is activated by querying the `aunt` relation with a first argument of `Bill`. After activation, the backward chaining rule first retrieves all bindings for `?mom` in `(mother Bill ?mom)`. After that, the remainder of the rule is processed *exactly the same as the forward-chaining rule*. Below are the AAM instructions that implement the backward-chaining rule. The code for the forward-chaining rule is identical to the last two-thirds of this code.

```
        :CLAUSE (MOTHER BILL ?MOM)
        :CONTINUATION   RULE376-001
        :SAVE
        :ACTIVATE-IN
        :SELECT
        :QUERY
        :FAIL?  :FAIL
        :MERGE-BINDINGS
        :BRANCH

        :CLAUSE (SISTER ?MOM ?AUNT)
        :SUBST
        :CONTINUATION   RULE376-002
        :SAVE
        :ACTIVATE-IN
        :SELECT
        :QUERY
        :FAIL?  :FAIL
        :MERGE-BINDINGS
        :BRANCH

        :CLAUSE (AUNT BILL ?AUNT)
        :SUBST
        :ASSERT
        :KNOWN? :L-66
        :FAIL?  :FAIL
        :ACTIVATE-IA
        :BRANCH
  :L-66
  :END  :RETURN :NOTHING
  :FAIL :RETURN :FAILURE
```

6 Performance monitoring

Our implementation of the AAM has built-in performance monitoring and statistic gathering mechanisms. Below is a table that shows the number of calls for each AAM instruction for an example program. We also gather information on time per call, but the majority of the times are below the resolution of the system clock, and are therefore unreliable. The total runtime was about 50ms on a Sun UltraSparc 140 running Franz Allegro Common LISP v4.3. About 15KBytes of memory was allocated during the run. The time includes time to compile the initial tell instruction, but not the time used to compile the rules.

Some of the metered operations do not correspond directly to AAM instructions. For example, `:INSTANTIATE-SUCC 3` denotes how many of the rule instantiations `:INSTANTIATE 4` were successful. The rule instantiations themselves are a subset of the `:ACTIVATE-IA 8` and `:ACTIVATE-IN 1` instructions.

AAM Instruction	Calls
`:SELECT`	15
`:BRANCH`	12
`:LABEL`	12
`:CLAUSE`	11
`:SUBST`	10
`:ASSERT`	9
`:FAIL?`	9
`:KNOWN?`	9
`:RETRIEVE-RULES`	9
`:ACTIVATE-IA`	8
`:UNIFY`	5
`:INSTANTIATE`	4
`:RETURN`	4
`:UNIFY-INST`	4
`:INSTANTIATE-SUCC`	3
`:REASONING-OK`	3
`:MERGE-BINDINGS`	2
`:ACTIVATE-IN`	1
`:CONTINUATION`	1
`:KB-DEF-FRAME`	1
`:PUSH`	1
`:QUERY`	1
`:REASONING-SUCC`	1
`:UNIFY-FAILED`	1
Total Calls	**136**

The table below shows statistics for the underlying frame system for the same example.

Of interest to note are the number of retrievals, which is more than ten times the number of stores, even though the main operation in question is a store, followed by an if-added rule that does another store. Also, about two-thirds of the operations are doing simple type-checking.

SFS Instruction	Calls
`:FRAME-P`	198
`:SLOT-P`	178
`:FACET-P`	171
`:RETRIEVE`	161
`:SLOT-FULL-P`	11
`:STORE`	11
`:SELECT`	2
`:STORE-NAME`	2
`:NEW-FRAME`	1
`:STORE-KNOWN`	1
Total Calls	**736**

6.1 Potential AAM optimizations

The execution trace in Section 5 shows several possible optimizations. For example, the code sequence `:BRANCH :SELECT` occurs several times without having any effect on the reasoning process. This happens when a branch opportunity (either a query or rule activation) did not produce a branch. An optimization would be to have the query or rule activation note when a branch does not occur. The subsequent branch and select could then be bypassed. A similar optimization could be made for `:SAVE` when used with `:ACTIVATE-IN`.

Another minor optimization involves multiple labels at one place. A post-compilation optimization pass could consolidate multiple labels into one. However, this optimization will probably not significantly increase the reasoning speed.

7 Conclusions and future research

In this report we have described access-limited logic (*ALL*) programs and how they are executed. The *Algernon Abstract Machine* provides an execution base for *ALL* programs and an assembly language into which they can be compiled. Access-limited logic itself requires approximately eighteen AAM instructions. We have extended our version of the AAM to provide full support for the Algernon language, which is a superset of access-limited logic.

Access limitation is enforced by the compiler, which implies that the AAM instructions could be used in many general-purpose rule-based and logic-based systems. We speculate that the AAM, WAM and other similar machines could be generalized into a general-purpose *Abstract Reasoning Machine* capable of supporting any of the current major models of reasoning.

In addition, an abstract machine can serve as an effective interface between external rule-based systems and a network-accessible knowledge base. Remote rule-based systems compile user queries, assertions and rules into abstract machine code, send the code to the KB for execution and await the results. Assuming that the abstract machine is general enough, many existing rule-based systems and shells could access online KBs without the need for a common high-level reasoning language. The AAM provides a solid foundation for a remote KB inferencing interface.

Further information about the Algernon system can be found at the Algernon web site: `http://www.cs.utexas.edu/users/qr/algy/`.

Bibliography

Aït-Kaci, H. (1991). *Warren's Abstract Machine: a tutorial reconstruction*. MIT Pr ess, Cambridge.

Brownston, L., R. Farrell, E. Kant and N. Martin (1985). *Programming Expert Systems in OP An Introduction to Rule-Based Pr ogramming, Addison-Wesley, Reading, MA.*

Crawford, J. (1990). *Access-Limited Logic -- A Language for Knowledge Representation.* PhD Thesis, University of Texas at Austin, TR AI90-141.

Crawford, J. M. and B. J. Kuipers (1991). Algernon -- a tractable system for knowledge representation. *SIGART Bulletin* **2**(3): 35-44, June 1991.

Doorenbos, R. B. (1993). Matching 100,000 Learned Rules, In: *Proceedings of AAAI-93*, Washington, D.C.

Eisenstadt, M. and M. Brayshaw (1990). A Knowledge Engineering Toolkit, *BYTE* **15**(10/11), October/November 1990. http://kmi.open.ac.uk/pr ojects/projects.html

Forgy, C. L. (1982). RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern M Problem, *Artificial Intelligence*, **19**:17-37, September, 1982.

Goldberg, A. and D. Robson (1983). *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley.

The Haley Enterprise. *The Eclipse system.* http://www.haley.com.

Harlequin Corporation. *Knowledge Works.* http://harlequin.com.

Hewett, R. and M. Hewett (1997). Efficiency Mechanisms for a class of Blackboard Systems, *International Journal of Artificial Intelligence Tools* **6**(1):97-125.

Hillis, D. (1985) *The Connection Machine*, MIT Pr ess, Cambridge, Massachusetts.

Karp, P. D., K. Myers and T. Gruber (1995). The Generic Frame Protocol. In: *Proceedings of the 1995 International Joint Confer ence on Artificial Intelligence* , pp. 768-774.

Kogge, P. M. (1991). *The Architecture of Symbolic Computers.* McGraw-Hill, New York.

Kuipers, B. J. (1994). http://www.cs.utexas.edu/users/qr/algy/.

Lindholm, T. and F. Yellin (1996). *The Java™ Virtual Machine Specification.* Addison-Wesley.

Miranker, D. P. and B. J. Lofaso (1991). The Organization and Performance of a TREAT-Bas Production System Compiler. *IEEE Transactions on Knowledge and Data Engineering* **3**(1):3- 10.

Rice, J., Aiello, N. and H. P. Nii (1989). See How They Run…The Architecture and Performance of Two Concurrent Blackboard Systems, In: *Blackboard Architectures and Applications*, V. Jagannathan, R. Dodhiawala and L. S. Baum, Eds. Academic Press.

Rumelhart, D. E. and J. L. McClelland. (1986). *Parallel Distributed Processing, Vols I and II.* MIT Press, Cambridge, Massachusetts.

Tambe, M. and P. S. Rosenbloom (1994). Investigating Production System Representations f Non-combinatorial Match. *Artificial Intelligence* **68**(1), 1994.

Teknowledge Corporation. *M.4 Expert Systems Software*, http://www.teknowledge.com/.

Van Wijngaarden, et al. (1975). Revised Report on the Algorithmic Language ALGOL68, *A Informatica* **5**:1-236.

Warren, D. H. D. (1977). *Implementing PROLOG–Compiling Predicate Logic Programs, Vols. 1 an 2, Reports Nos. 39 and 40,* Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, May.

Appendix A: Formal definitions of the AAM instructions

Following are the formal definitions of the eighteen core AAM instructions. Each is defined by its effect on the AAM registers. In the notation, the name of the register is followed by the original contents of a register and the contents after the instruction has been executed. The → symbol should be read as "becomes". X denotes the original contents of the register being manipulated, while **[Reg]** denotes the contents of a another register. LISP notation is used to describe the data. For example an instruction whose action is described by:

```
B:  (NIL . X) → X
IA:           → [C]
```

means that NIL was originally at the top of the **B** register and was removed by the instruction. In addition, the contents of the **IA** register were replaced by the contents of the **C** register, regardless of the original contents. The registers are represented by letters in **bold** style as follows:

A	Actions
B	Bindings
C	Clause
IA	If-added activation clause
IN	If-needed activation clause
P	Program Code
R	Results

:ACTIVATE-IA

```
A: X → X + activate-if-added-rules(IA)
```

- The '+' is a set addition operation. Entries in the **A** register may be ordered according to the implementation or the application. The application programmer should be allowed to control the ordering of items in the **A** register.

:ACTIVATE-IN

```
A: X → X + activate-if-needed-rules(IN)
```

- The '+' is a set addition operation. Entries in the **A** register may be ordered according to the implementation or the application. The application programmer should be allowed to control the ordering of items in the **A** register.

:ASSERT

```
IA:  → [C]
B: X → (SFS:PUT-VALUE(frame([C]),slot([C]),facet([C])) . X)
```

:BRANCH

```
for every b ∈ [B]:
  A: X → X + make-activation(b, [A], [C], [IA], [IN])
```

- All other registers are set to `NIL` in order to force a `:SELECT` operation.
- Entries in the **A** register may be ordered according to the implementation or the application. The application programmer should be allowed to control the ordering of items in the **A** register.

:CLAUSE <clause>

```
C: → <clause>
```

:CONTINUATION <rule>

```
for every b ∈ [B]:
  store-rule-closure(<rule>, b, frame([C]), slot([C]),
                     rule-facet([C]))
```

:FAIL? <label>

```
if null(car([B]))
 P: FAIL? <label> … :LABEL <label> . X → X
else if atom(car([B]))
 B: (a . X) Æ X
else
   do nothing
```

- This checks for failure of the previous action.

:KNOWN? <label>

```
if (eq :KNOWN (car [B])
  P:  :KNOWN? <label> … :LABEL <label> . X → X
  B:  (:KNOWN . X) → X
 else
  do nothing
```

:LABEL

P: :LABEL <label> . X → X

:MERGE-BINDINGS

```
B: (bset1 bset2 . X) → merge-bindings(bset1, bset2) . X
```

:PUSH <value> <register>

```
<register>: X → <value> . X
```

:QUERY

```
let values = SFS:get-values(frame([C]),slot([C]),facet([C]))

if null(values)
 B: X → NIL . X
else
  let r = algy-unify(values, args([C]))
 B: X → (mapcar #'bindings r) . X
```

- NIL indicates a failed query.

:RETURN <type>

```
if <type> is:
:SUCCESS: P: → NIL
           if null(car([B])
             do nothing
           else
      for every b ∈ car([B])
      R: X → (T . (b)) . X

 :NOTHING: do nothing

 :FAILURE: B: (b . X) → X
```

:SAVE

- Exactly the same as :BRANCH except that the other registers are not set to NIL.

:SELECT

```
A: X → X - select(X)
```

- select is a user-definable function that selects one activation from the set of activations in **A**. By default, the AAM selects the last action stored. The other registers are restored from the contents of the selected activation.

:SUBR <var> <code>

```
let (result . binding-set) = process(<code>);

if null(result)
 B: X → (NIL . X)
else if null(<var>)
 B:   → (binding-set)
else
 B: X → (filter(binding-set, <var>) . X)
```

- If a variable is specified, the binding set returned from the recursive call to the AAM processor is filtered, leaving only bindings of <var>.

:SUBST

```
C: X → subst(X, (caar [B]))
```

- The variable bindings in the first binding list of **B** are substituted for variables in the current clause.

:VERIFY

```
let values = SFS:get-values(frame([C]),slot([C]),facet([C]))

if member(args([C]), values)
 B: X → T . X
else
 B: X → NIL . X
```

- The equivalent of the LISP `equalp` function should be used to compare values in the call to member.

Appendix B: Utility functions from Algernon

This section presents the implementation of several non-obvious routines used in Algernon for unification and binding set manipulation.

B.1 Unification in access-limited logic

This section defines the unify function as used in *ALL*. Since *ALL* does not have complex patterns, the function is more of a pattern matching function than a general-purpose unification algorithm such as the one used in Prolog. Also, this function maintains two binding lists; one for the pattern and one for the clause being matched. This eliminates the need to rename variables to handle situations in which the pattern and the clause have a variable of the same name.

The input to unify is a pattern to match and an expression to match with it. Both the pattern and the expression have the form of a clause such as `(action robot ?location ?behavior)`. Variables may occur anywhere in either clause.

```
(defun unify (pattern expression)

  (let ((p-and-e-bindings
           (unify-aux pattern expression '(NIL . NIL))))

    (if (eq (car p-and-e-bindings) :FAILED)
       (cons :FAILED  NIL)
    ;;else
    ;; Some variables in the pattern may have been bound to
    ;; variables in the expression that now have bindings.
    ;; Assign bindings to those.

       (cons T
             (fixup-variables (car p-and-e-bindings)
                              (cdr p-and-e-bindings))))))
```

```
(defun algy-unify-aux (pattern expression p-and-e-bindings)

  (if (eq (car p-and-e-bindings) :FAILED)
    p-and-e-bindings
    ;;else
    (let ((p-bindings  (car p-and-e-bindings))
          (e-bindings  (cdr p-and-e-bindings))
          )

      ;; Instantiate the variables, if possible.
      (when (variable-p pattern)
        (setq pattern (var-lookup pattern p-bindings)))

      (when (variable-p expression)
        (setq expression (var-lookup expression e-bindings)))

      (cond ((null pattern)
             (if (null expression) p-and-e-bindings
                 (cons :FAILED :FAILED)))
            ((consp pattern)
             (algy-unify-aux (cdr pattern) (cdr expression)
                (algy-unify-aux (car pattern)
                         (car expression) p-and-e-bindings)))

            ((variable-p pattern)
             (setq p-bindings (bind pattern expression
                                    p-bindings))
             (cons p-bindings e-bindings))

            ((variable-p expression)
             (setq e-bindings (bind expression pattern
                                               e-bindings))
             (cons p-bindings e-bindings))

            ;; Names are strings, so handle them separately.
           ((and (stringp pattern)
                 (not (stringp expression)))
             (if (funcall *equalp-test*
                       (read-from-string pattern) expression)
                (cons p-bindings e-bindings)
                (cons :FAILED e-bindings)))
```

continued below

```
algy-unify-aux continued…

            ;; Names are strings, so handle them separately.
           ((and (stringp expression)
                 (not (stringp pattern)))
             (if (funcall *equalp-test*
                     (read-from-string expression) pattern)
                (cons p-bindings e-bindings)
                ;;else
                (cons :FAILED e-bindings)))

            (T   (if (funcall *equalp-test*
                         pattern expression)
                   (cons p-bindings e-bindings)
                   ;;else
                   (cons :FAILED e-bindings)))))))
```

```
(defun var-lookup (var bindings)
  "Returns the var if it has no binding."

  ;; Returns either the value of the variable or the variable
  ;; itself if there is no binding for it.

  (let ((entry  (assoc var bindings)))

    (if entry
      (cdr entry)
      ;;else
      var)))
```

```
(defun bind (var value bindings)

  ;; Puts (var . value) on the bindings list unless it finds
  ;; a binding for that variable already there.
  ;; If the existing binding is the same, that's fine.
  ;; If it's different, unification fails.

  (let ((entry (assoc var bindings)))

    (cond (entry  (if (eq value (cdr entry))
                      bindings :FAILED))
          (T      (cons (cons var value) bindings)))))
```

A final function, `fixup-variables`, makes a final pass through the binding list and fixes or removes any variables that are bound to other variables, since the caller is only interested in bindings that go from a variable to a non-variable.

```
(defun fixup-variables (p-bindings e-bindings)

  ;; All variables in the pattern should now have bindings.
  ;; Some of them need to be found transitively by searching
  ;; the e-bindings list.
  ;; This loop fixes up the bindings by finding the correct
  ;; value.  If there is no value, the binding is removed.

  (dolist (binding p-bindings p-bindings)
    (when (variable-p (cdr binding))
      (unless (variable-p (var-lookup (cdr binding)
                                      e-bindings))
        (setf (cdr binding) (var-lookup (cdr binding)
                                        e-bindings)))))

  ;; Some bindings still may be to variables.  Remove them as
  ;; they are of no use to us.

  (delete-if #'variable-p p-bindings :key #'cdr))
```

B.2 Binding set merge

The function below is used to merge two binding sets in the AAM instruction :MERGE-BINDINGS. It assumes that there are no duplicate bindings in the two binding sets.

```
(defun merge-binding-sets (new-blset BLSet)
  "Returns a set of binding lists.  Each input is a set
of binding lists."

  ;;; We need to do a sort of cross-product.

  (cond ((null new-blset)  BLSet)
        ((null BLSet)      new-blset)
        (T
         (mapcan #'(lambda (bl1)
                     (mapcar #'(lambda (bl2)
                                  (append bl1 (copy-list bl2)))
                                 new-blset))
                   BLSet))))
```

Appendix C: Initializing the Knowledge Base

The following forms bootstrap and initialize the knowledge base so that ALL reasoning can begin. Some of the functions used are compatible with the Generic Frame Protocol [Karp, 1995], which provides a KB interface layer above the Simple Frame System. In addition to the code shown below, ALL requires that the base types `:number`, `:string`, `:symbol` and `:list` be recognized and understood by the AAM compiler.

The following forms are copied from the file "`algy-core-kb`" in the Algernon source directory.

```
(in-package :CL-USER)


#|
 Appendix A:
Knowledge Base Initialization The following forms will initialize the
knowledge base so that A-L-L reasoning can begin.  These forms are written
using the Generic Frame Protocol.  In addition to the code shown below, ALL
requires that the base types :number, :string, :symbol and :list be
defined.
|#



(aam:with-aam-silent         ;; We don't want output from these assertions.

;;;  create the basic facets

(GFP:create-facet 'value            NIL NIL) ;; (slot frame <value>)
(GFP:create-facet 'n-value          NIL NIL) ;; (not (slot frame <value>))
(GFP:create-facet 'if-added         NIL NIL) ;; Forward-chaining rules
(GFP:create-facet 'n-if-added       NIL NIL) ;; FC on (not ...)
(GFP:create-facet 'if-needed        NIL NIL) ;; Backward-chaining rules
(GFP:create-facet 'n-if-needed      NIL NIL) ;; BC on (not ...)

(GFP:create-facet 'slot-if-added     NIL NIL) ;; Forward-chaining on slots
(GFP:create-facet 'slot-n-if-added   NIL NIL) ;; FC on slot (not ...)
(GFP:create-facet 'slot-if-needed    NIL NIL) ;; Backward-chaining on slots
(GFP:create-facet 'slot-n-if-needed  NIL NIL) ;; BC on slot (not ...)

(GFP:create-facet 'self-if-added     NIL NIL) ;; Rule Continuations
(GFP:create-facet 'self-n-if-added   NIL NIL) ;; Rule Continuations, negated

(GFP:create-facet 'cache-if-added     NIL NIL) ;; Forward-chaining on slots
(GFP:create-facet 'cache-n-if-added   NIL NIL) ;; FC on slot (not ...)
(GFP:create-facet 'cache-if-needed    NIL NIL) ;; Backward-chaining on slots
(GFP:create-facet 'cache-n-if-needed  NIL NIL) ;; BC on slot (not ...)

(GFP:create-facet 'cache-slot-if-added     NIL NIL) ;; FC on slots
(GFP:create-facet 'cache-slot-n-if-added   NIL NIL) ;; FC on slot (not ...)
(GFP:create-facet 'cache-slot-if-needed    NIL NIL) ;; BC on slots
(GFP:create-facet 'cache-slot-n-if-needed  NIL NIL) ;; BC on slot (not ...)
```

```
(GFP:create-facet 'queries            NIL NIL) ;; Storage for Query History


;;;  create the basic slots

(with-no-rule-caching     ;; We haven't bootstrapped enough yet...
    (let ((aam::*forward-chain*   NIL)
          (aam::*backward-chain*  NIL)
          (*check-slot-domains*   NIL))

(tell '((:slot isa    (things sets))
        (:slot name   (things :string))
        (:slot arity  (Slots :NUMBER))
        ))

;; Generic must come before the first time we do an assert,
;; which comes in the next 'tell' after this.

(tell '((:slot generic  (things rules))
        (arity isa      2)      ;; the ARITY slot wasn't available above...
        (arity name     2)      ;; so we assert these by hand.
   ))


;;;  Bootstrap the top of the hierarchy
;;;  We have to be careful to not use slots that aren't defined yet.

(SFS:kb-def-frame 'things)
(SFS:kb-def-frame 'slots)
(SFS:kb-def-frame 'rules)
(SFS:kb-def-frame 'objects)
(SFS:kb-def-frame 'physical-objects)
(SFS:kb-def-frame 'booleans)

(SFS:kb-def-frame 'true)
(SFS:kb-def-frame 'false)


;;;  IMP-SUPERSET is related to activation of rules and
;;;  propagation of the ISA relation.

(tell '((:slot imp-superset (sets sets))))     ;; taxonomy

;;;  Slots related to the inheritance hierarchy

(tell '((:slot member (sets things))))

;;;  Define the major class/subclass relation
(tell '((:slot   subset     (sets sets))       ;; class/subclass
        (:slot   superset   (sets sets))
   ))

))  ;; end of WITH-NO-RULE-CACHING
```

```
;;; --------  Done with important bootstrapping  ---------------


(let ((*check-slot-domains*  NIL))  ;; Don't have enough info to check yet.

;;; 'cardinality' and 'inverse' are already defined as
;;; facets in GFP, but A-L-L views them as slots.

(tell '((:slot cardinality    (Slots :NUMBER))
        (:slot inverse        (Slots Slots))
        (:slot complete       (Sets  Booleans))
        (:slot generalization (Slots Slots))
        ))

(tell '((inverse subset     superset)
        (cardinality cardinality 1)
        (cardinality inverse     1)
        ))



;;;; In Algernon, Rules and Rule Continuations are stored
;;;; in the knowledge base on frames and facets of frames.
;;;; So the components of a rule are stored in slots.

;;;  slots related to rule components

(GFP:create-class 'directions  '(objects))
(GFP:create-instance '-> '(directions))
(GFP:create-instance '<- '(directions))

(tell '((:slot antecedent   (rules  :LIST))
        (:slot consequent   (rules  :LIST))
        (:slot code         (rules  :LIST))
        (:slot direction    (rules  directions))
        (:slot index        (rules  :NUMBER))
        (:slot key          (rules  :LIST))
        (:slot root         (rules  :SYMBOL))
        (cardinality antecedent 1)
        (cardinality consequent 1)
        (cardinality direction  1)
        (cardinality key        1)
        (cardinality index      1)
        (cardinality root       1)))


;;;; Relations between rules and rule continuations

(tell '((:slot       instance-of  (rules  rules))
        (cardinality instance-of  1)
  (:slot       disjoint     (sets   sets))))

;;; Now we can create some rules     The DEFAULT RULES

;;;  For every 'member' link we want an 'isa' link, but
;;;  not necessarily vice-versa.  So we don't make
;;;  'member' and 'isa' be inverses.

(tell '((:srules member
         ((member ?set ?x) -> (isa ?x ?set))
         ((not (member ?f1 ?f2)) -> (not (isa ?f2 ?f1))))))


;;;  This rule propagates inverse links.
(tell '((:srules SLOTS
                 ((inverse ?s1 ?s2)
                  ->
                  (:SRULES ?s1 ((?s1 ?x ?y) -> (?s2 ?y ?x)))
                  (:SRULES ?s1 ((not (?s1 ?x ?y)) -> (not (?s2 ?y ?x))))

                  (:SRULES ?s2 ((?s2 ?y ?x) -> (?s1 ?x ?y)))
                  (:SRULES ?s2 ((not (?s2 ?y ?x)) -> (not (?s1 ?x ?y))))
                  )
                 )))


;;;  This rule propagates a "one-way" inverse link for imp-superset.
(tell '((:srules IMP-SUPERSET
                 ((imp-superset ?s1 ?s2)
                  ->
                  (subset ?s2 ?s1)))))

;; Important transitive closure of imp-superset
(tell '((:srules IMP-SUPERSET
      ((imp-superset  ?a ?b)
       (imp-superset  ?b ?c)
       ->
       (imp-superset ?a ?c)))))


;; The Generalization slot implements a hierarchy of slots.
;; For example, 'spouse' is a generalization of 'wife'
(tell '((:srules generalization
      ((generalization ?P1 ?P2)
       (arity ?P1 2)
       (arity ?P2 2)
       ->
       (:srules ?P1
          ((?P1 ?x1 ?x2) -> (?P2 ?x1 ?x2)))))

   (:srules generalization
      ((generalization ?P1 ?P2)
       (arity ?P1 3)         ; and similar for arities 3,4,5.
       (arity ?P2 3)
       ->
       (:srules ?P1
          ((?P1 ?x1 ?x2 ?x3) -> (?P2 ?x1 ?x2 ?x3)))))

   (:srules generalization
      ((generalization ?P1 ?P2)
       (arity ?P1 4)         ; and similar for arities 3,4,5.
       (arity ?P2 4)
       ->
       (:srules ?P1
          ((?P1 ?x1 ?x2 ?x3 ?x4) -> (?P2 ?x1 ?x2 ?x3 ?x4)))))

   (:srules generalization
      ((generalization ?P1 ?P2)
       (arity ?P1 5)         ; and similar for arities 3,4,5.
       (arity ?P2 5)
       ->
       (:srules ?P1
          ((?P1 ?x1 ?x2 ?x3 ?x4 ?x5) -> (?P2 ?x1 ?x2 ?x3 ?x4 ?x5)))))
   ))


;;; Now reassert the taxonomy to get all the necessary relations defined.

;;; It is okay to reassert the classes because GFP:CREATE-CLASS
;;; will reuse the old frame.  But we need to get rid of the
;;; instances before we do the taxonomy.  Otherwise we get
;;; frames like 'TRUE1'

;;; Instances
(sfs::kb-delete-values 'directions 'member)   ;; Delete links to ->   and <-
(sfs::kb-delete-values 'booleans   'member)   ;; Delete TRUE and FALSE

(SFS:kb-delete-frame '->)
(SFS:kb-delete-frame '<-)
(SFS:kb-delete-frame 'TRUE)
(SFS:kb-delete-frame 'FALSE)


;; Recreate the taxonomy
;; This should cause rules to fire.
(GFP:create-class  'things   NIL)    ;; The two key classes.
(GFP:create-class  'sets     NIL)

(tell '((:taxonomy (things
         (slots)
         (sets)
         (rules)
         (objects
          (sets things objects sets slots)
          (booleans false true :complete)
          (contexts global-context)
          (directions  ->   <-    :complete)
          (physical-objects))
         ))))


;; The imp-superset links came in "too late" for frames created in
;; bootstrapping so we have to add isa links explicitly:
;;
```

```
(tell '(
   (isa things  objects)
   (isa objects objects)
   (isa sets    objects)
   (isa slots   objects)

   (isa things  things)
   (isa objects things)
   (isa sets    things)
   (isa slots   things)

   (isa things  sets)
   (isa objects sets)
   (isa sets    sets)
   (isa slots   sets)
   )
  :comment "Algernon Core KB loaded.")


;;; Delete some inconsistencies
(tell '((:delete (not (isa things  objects)))
        (:delete (not (isa things  sets)))
        (:delete (not (isa objects sets)))
        ))


) ;; End of let .. *check-slot-domains* NIL

) ;; End of WITH-AAM-SILENT
```

Appendix D: Specification of the Simple Frame System

This appendix contains a specification of the frame system used in Algernon, an implementation of access-limited logic. The frame system is used to define all frames, slots and facets; to keep track of names of frames; and to retrieve, store and delete values from the knowledge base.

The AAM accesses values from the KB via frame, slot, and facet names, but it does not know or care about the internal format of the KB. The SFS does not know anything about the AAM and is not dependent on the characteristics of any program that uses it.

As described in Chapter 2, the AAM reasons using clauses of the form `(r a b ...)`, but the SFS stores the relations as frames, where `a` is the frame name, `r` is the slot name, and the facet is `value` if the clause is positive, and `n-value` if the clause is negated (the facet is determined by the AAM).

The SFS implements a straightforward frame-slot-facet representation. Each facet can have multiple values, up to the value specified by the cardinality of its slot. Since a slot may have any arity, values are stored as lists. For example, the clause `(r F a b c)` will be stored as value `(a b c)` of slot `r` of frame `F` Even binary relations are stored as a list: `(r a b)` stores the value `(b)`. All functions that retrieve values should be consistent in returning values in this manner.

D.1 General notes

Since the SFS was designed with LISP as the base language, terms referring to language-level data types such as symbol refer to the definition of the data type in LISP. Similarly, the values `TRUE` and `FALSE` used below refer to `T` and `NIL` when the SFS is implemented in LISP. Equality tests should be performed using the equivalent of the Common LISP `equalp` function.

D.1.1 Frame names

Frame, slot and facet names are symbols. A frame may have one or more **public names**, which are strings. The SFS maintains a Public Name Dictionary to translate to and from public names. The Public Name Dictionary indexes frames by their (symbolic) name.

Often, a program will attempt to create a frame with a suggested name (a symbol) that is already in use. It is recommended that a new unique name be created from the symbol by appending a number to the end of the suggested name. For example, repeated attempts to define a frame named `Box` should result in the frames `Box1`, `Box2`, ... being created.

Since the user defines frame, slot and facet names, they are interned in the CL-USER package when implemented in LISP. Since the SFS is in its own package, the programmer should remember to access facet names such as `VALUE` from the `CL-USER` package. The AAM is also in its own package. Packages are not discussed further in this document.

D.1.2 Efficiency

The functions `FRAME-P`, `SLOT-P`, and `FACET-P` are called much more often than than any other functions in the system, so they should be optimized. Similarly, retrieval normally occurs an order of magnitude more often than storage, so retrieval should be optimized.

The `SELECT` function used in `GET-VALUES` is a simple filter. Its implementation is given in Section D.10.

D.2 Initializing the knowledge base

`RESET`

Clears the knowledge base completely, erasing all frames, slots and facets. Clears the Public Name Dictionary.

D.3 Defining knowledge base components

`DEF-FACET facet-name`

Creates a frame to represent the facet.

`DEF-FRAME suggested-name &KEY (create TRUE)`

```
If a frame by that name already exists
  If create is TRUE
    Generate a new unique name for the frame and
      create the frame as below
  else
    Print a warning that the frame is being redefined
    Return the name of the frame

Create a new frame
Store <frame, name> in the Public Name Dictionary
Return the name of the frame.
```

`DEF-SLOT  slot-name domains`

```
If a slot by that name exists already,
  Print a warning that the slot is being redefined.
  Delete the old slot and associated information.

Create a frame to represent the slot.
Store the slot and its domains.
Return the name of the slot frame.
```

`DECLARE-CONTINUATION frame-name`

Declares that the frame is a rule continuation. This status is used in the function `CONTINUATION-P`.

DECLARE-RULE frame-name

Declares that the frame is a rule. This status is used in the function RULE-P.

D.4 Knowledge base component type checking

CONTINUATION-P name

Returns TRUE if the name given is a rule continuation, FALSE otherwise.

FACET-P name

Returns TRUE if the name given is a facet, FALSE otherwise.

FRAME-P name

Returns TRUE if the name given is a frame, FALSE otherwise.

RULE-P name

Returns TRUE if the name given is a rule, FALSE otherwise.

SLOT-P name

Returns TRUE if the name given is a slot, FALSE otherwise.

D.5 Knowledge base storage

`PUT-VALUE frame slot facet value`

```
If the value was already present in the facet, (using EQUALP)
  return :KNOWN
 else
  if the frame is a frame type and
     the slot  is a slot type and
     the facet is a facet type and
     the frame/slot/facet is not SLOT-FULL-P

   Store the value in the frame/slot/facet in a FIFO order.
   When the slot is NAME,
      add the pair <frame, "value">  to the Public Name Dictionary

  else
   generate an appropriate error message and return FALSE.
```

D.6 Knowledge base retrieval

`ALL-CLAUSES frame`

For the given frame, returns all the information stored on it in the `VALUE` and `N-VALUE` facets of every slot, as a list of clausal forms. For example, if the frame `Bessie` is the argument, the result might be:

```
((provides Bessie  milk)
 (age      Bessie  7)
 (color    Bessie  brown)
 (legs     Bessie  4))
```

`GET-ALL-FRAMES`

Returns a list of every frame defined in the system.

GET-VALUES frame slot facet &OPTIONAL pattern

```
If the slot name is NAME and
    the frame is a string type and
    the facet is VALUE
   Return a list of every frame with that name.

 else
   if the frame is a frame type and
     the slot  is a slot type and
     the facet is a facet type and

    Retrieve the set of values from the given frame/slot/facet.
    If there is a pattern,
      Return only those values that are selected by the pattern
    else
      Return all values
   else
    Generate an appropriate error message and return FALSE.
```

GET-SLOT-DOMAINS slot

Returns the list of slot domains associated with the slot when it was defined.

ALL-SLOTS-OF-FRAME frame

Returns a list of the slots in use on the frame (ones that have any value for any facet).

ALL-FACETS-OF-SLOT frame

Returns a list of the facets in use on the given frame and slot.

SLOT-FULL-P frame slot facet

```
If the facet is the VALUE facet and
   the frame is a frame type,
 Find the length of the contents of the frame/slot/facet.
 Find the cardinality of the slot (as defined by the
    CARDINALITY relation on the slot's frame, if any).
  If the cardinality exists and
     the cardinality is <= to the length of the contents,
   Return TRUE
   else
    Return FALSE
else
  Return FALSE.
```

D.7 Knowledge base deletion

```
DELETE-FRAME frame

    Get all of the public names of the frame (from its NAME slot).
    Delete all of the pairs <frame, public name> from the
      Public Name Dictionary.
    Reinitialize the frame, erasing all slots, facets and notations
      of its type (rule, slot, ...).
    Delete the frame from any list kept of all known frames.
```

```
DELETE-VALUE frame slot facet value

    Delete the value from the frame/slot/facet (using EQUALP).
    If that was the last value for the facet,
      Remove the facet from the slot.
      If that was the last facet for the slot,
        Remove the slot from the frame.
```

```
DELETE-VALUES frame slot &OPTIONAL facet

    If the facet was given,
      (delete-all-values frame slot facet)
     else
      (delete-all-values frame slot VALUE)
      (delete-all-values frame slot N-VALUE)

;; Delete-all-values is for internal use only.
delete-all-values frame slot facet

     Remove the facet from the slot.
     If that was the last facet on the slot,
      Remove the slot from the frame.
```

D.8 Displaying KB information

Most of the print functions are described by examples showing suggested output, since print routines vary from language to language. These functions should probably belong to the AAM since they afford special handling to certain facets that the AAM uses. The facets given special treatment by the print routines are:

```
n-if-added         if-added
n-if-needed        if-needed
slot-n-if-added    slot-if-added
slot-n-if-needed   slot-if-needed
self-if-added      self-n-if-added
queries
```

PRINT frame &OPTIONAL stream

```
If the frame is a facet type
  (print-facet frame stream)
else if the frame is a slot type
  (print-slot  frame stream)
else if the frame is a frame type
  (print-frame  frame stream)
else
  Print an error message.
  Return FALSE.
```

PRINT-FACET frame &OPTIONAL stream

Example: `(print 'self-if-needed)`

```
SELF-IF-NEEDED is a facet.
```

PRINT-FRAME frame &OPTIONAL stream

Example: `(print 'bessel)`

```
BESSEL:
           NAME:  ("Bessel")
            ISA:  (WORDS)
             S1:  (X)
         NOT-S1:  (C)
     SELF-IF-ADDED:  (#S(RULE-CLOSURE :RULE RULE15-002
                   :BINDINGS ((?B . B) (?A . A)) :KEY NIL))
                     (#S(RULE-CLOSURE :RULE RULE15-002
                   :BINDINGS ((?B . B) (?A . C)) :KEY NIL))
           QUERIES:  (S1 B ?C)
```

PRINT-FRAME-NO-RULES frame &OPTIONAL stream

Example: `(print-no-rules 'bessel)`

```
BESSEL:
           NAME:  ("Bessel")
            ISA:  (WORDS)
             S1:  (X)
         NOT-S1:  (C)
```

PRINT-SLOT frame &OPTIONAL stream

Example: `(print 'color)`

```
COLOR is a slot with domains (PHYSICAL-OBJECTS COLORS)
 and slots:
  <slot-info here, as in PRINT-FRAME>
```

D.9 Public Name Dictionary functions

`DELETE-NAME frame public-name`

Deletes the entry `<frame, public-name>` from the Public Name Dictionary.

`GET-ALL-NAMES`

Returns all `<frame, public-name>` entries in the Public Name Dictionary.

`NAMES-RESET`

Empties the Public Name Dictionary.

`OBJECTS-NAMED public-name`

Returns a list of all frames whose public-name is as given.

`STORE-NAME frame public-name`

Stores the pair `<frame, public-name>` in the Public Name Dictionary, if it is not there already.

D.10 The SELECT function

Select is used in GET-VALUES to do the equivalent of a database select operation, returning those values that match a given pattern.

```
(defun select (pattern values)
  "Not a user-level function.  Returns the members
of the VALUES list that match the pattern."

  (remove-if-not #'(lambda (value)
                     (values-match value pattern))
                 values))
```

```
(defun values-match (value pattern)
  "Length value = length pattern."

  ;; If the pattern element is not a variable, see whether it
  ;; matches the corresponding value element.

  (do ((v-arg  value    (cdr v-arg))
       (p-arg  pattern  (cdr p-arg))
       (match  T        (or (variable-p (car p-arg))
           (funcall *equalp-test*
                       (car v-arg) (car p-arg)))))
      ((or (not match)
     (null v-arg))
       match)))
```