Alma-0: An Imperative Language that Supports Declarative Programming

KRZYSZTOF R. APT
CWI and University of Amsterdam
and
JACOB BRUNEKREEF and VINCENT PARTINGTON
University of Amsterdam
and
ANDREA SCHAERF
Università di Roma "La Sapienza"

We describe here an implemented small programming language, called Alma-0, that augments the expressive power of imperative programming by a limited number of features inspired by the logic programming paradigm. These additions encourage declarative programming and make it a more attractive vehicle for problems that involve search. We illustrate the use of Alma-0 by presenting solutions to a number of classical problems, including α-β search, STRIPS planning, knapsack, and Eight Queens. These solutions are substantially simpler than their counterparts written in the imperative or in the logic programming style and can be used for different purposes without any modification. We also discuss here the implementation of Alma-0 and an operational, executable, semantics of a large subset of the language.

Categories and Subject Descriptors: D.3.2 [**Language Classifications**]: Nondeterministic Languages; F.3.2 [**Semantics of Programming Languages**]: Operational Semantics; F.3.3 [**Studies of Program Constructs**]: Control Primitives; I.2.8 [**Problem Solving, Control Methods and Search**]: Backtracking; I.5.5. [**Implementation**]: Special Architectures

General Terms: Languages

Additional Key Words and Phrases: Declarative programming, imperative programming, search

1. INTRODUCTION

In this article we describe a programming language, Alma-0, that combines advantages of logic and imperative programming in order to deal in a natural way with algorithmic problems that involve search. Alma-0 extends imperative programming

Authors' addresses: K. R. Apt, CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands; J. Brunekreef and V. Partington, Department of Mathematics, Computer Science, Physics & Astronomy, University of Amsterdam, Plantage Muidergracht 24, 1018 TV Amsterdam, The Netherlands; A. Schaerf, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy.

with some features that are inspired by the logic programming paradigm. In our design we were guided by the following four principles:

—The proposed extension should be downward compatible with the underlying imperative programming language.

—This extension should be upward compatible with a future extension that will support constraint programming.

—The proposed constructs should support declarative programming.

—This extension should be small. (In fact, we propose nine new features.)

We believe that these postulates make our proposal distinct and substantially simpler from previous proposals that dealt with integration of constructs inspired by declarative programming languages (for example, automatic backtracking) into imperative programming.

In fact, Alma-0 should not be viewed only as a specific programming language proposal but rather as an instance of a *generic method* for extending (essentially) any imperative programming language with facilities that encourage declarative programming.

To demonstrate the feasibility of our approach we went through the full process of the implementation of the language and the description of its semantics for a specific base imperative language, namely a subset of Modula-2.

The proposed features include

—use of boolean expressions as statements and vice versa,

—a statement dual to the `FOR` statement that introduces ("don't know") nondeterminism in the form of choice points and backtracking,

—a `FORALL` statement that introduces a controlled form of iteration over the backtracking,

—unification—here limited to a use of equality as assignment; this yields a new parameter-passing mechanism.

In such an amalgamated language we can freely profit from the advantages of both programming styles.

The assignment, shunned in declarative programming and, a fortiori, in logic programming, is in our opinion needed in a number of natural situations, which we illustrate by means of several examples. In general, assignment seems to be needed for counting or for recording purposes, and means of expression of such uses offered within the logic programming paradigm are unnatural. In particular, in Prolog, assignment is either used in a space inefficient and limited form, like in `X1 is X+1`, or is simulated using `assert` and `retract`. In our view the direct use of assignment, as in imperative programming, is in such cases simpler and more efficient. Further, we can use a rich variety of data types, including arrays and records, in presence of strong type checking and several traditional control structures that support structured programming.

In turn, the logic programming paradigm provides a number of useful features. The built-in backtracking mechanism supports nondeterministic programming in a simple way. The use of unification to assign values allows us to use the same program for testing, computing one, some, or all solutions, or for completing a partial solution. This versatile use of programs is also available in Alma-0. It should be pointed out, however, that our use of unification is extremely restricted, and consequently another important aspect of logic programming—symbolic programming—is not realized in Alma-0.

Combining two programming styles is always a debatable endeavor, and it is important to reflect what, if any, are the advantages of such an amalgamation. We try to answer this question by presenting solutions to several classical problems. We consider these programs superior to their counterparts written as imperative programs or as programs in the logic programming style for the following reasons:

—In each case the programs are closer to the specifications than the alternative solutions. This suggests that the proposed additions make the programming task simpler and improve readability.

—The presented programs, or program fragments, that do not use assignment can be viewed as declarative in the sense that they admit an alternative reading as logic formulae. Development and verification of such programs is considerably simplified due to their logical meaning. In some cases programs are equal to their specifications—e.g., see our solutions to Problems 3 (*Straight String Search*), 7 (*Remarkable Sequence Revisited*), and 9 (*Linear Search*)—and are therefore obviously correct.

—All the programming constructs introduced in Alma-0 are guaranteed to terminate. As a result we can now write programs, like the solutions to the just mentioned problems or solutions to Problems 6 (*Knapsack*) and 10 (*Squares in the rectangle*), termination of which is guaranteed by their syntactic form.

—When passing from specifications to a solution the introduction of additional variables should be viewed as a drawback, because their relation to the variables present in the specifications has to be properly explained. From this viewpoint constructs or solutions (of the same complexity) that do not call for the use of additional variables should be considered as superior. Now, the proposed solutions do introduce less variables than the traditional ones.

In our opinion, the proposed additions blend well with the conventional way we look at imperative programs.

As the underlying language for Alma-0 we use Modula-2 of Wirth [1985]. More precisely, Alma-0 is an extension of a subset of Modula-2. An alternative choice, C, in contrast to Modula-2, would have required a change of the semantics of the base language. Indeed, in C, boolean expressions followed by a semicolon (`;`) are already legal statements, the presence of which has no effect on the flow of computation.

It should be stressed, however, that the base language is completely inessential in our investigations. The presented programs in Alma-0 should be understandable by anybody familiar with the basics of an imperative language. Moreover, the proposed additions can be naturally incorporated into most of the programming languages supporting the imperative programming paradigm.

To substantiate the claim that Alma-0 supports declarative programming we introduce for it a declarative semantics. Admittedly, this semantics is applicable only to the programs built out of a limited number of constructs. Therefore, we also present an alternative— operational and executable—semantics for a larger subset of Alma-0 that focuses on the most relevant features of the languages.

The implementation of the language is based on an abstract machine that combines the features of a RISC architecture and the WAM abstract machine. In the current implementation the abstract machine instructions are translated into C code. The Alma-0 compiler is available via the Web at `http://www.cwi.nl/alma`.

The article is organized as follows. In Sections 2, 3, 4, and 5, we introduce in stages the extensions of the language, and summarize them in Section 6, where we also discuss the features of Modula-2 which are at this stage not implemented in Alma-0. In Sections 7 and 8 we describe the declarative and operational semantics of Alma-0, respectively, and in Section 9 we explain its implementation. Finally, related and future work are discussed in Section 10.

2. BOOLEAN EXPRESSIONS AND STATEMENTS

We begin by identifying boolean expressions and statements.

2.1 Boolean Expressions as Statements

First, we allow boolean expressions to be used as statements. We denote this extension by **BES**. In what follows we refer to boolean expressions used as statements as *tests*.

An evaluation of a test can yield `TRUE, FALSE` or can cause a run-time error if an uninitialized variable is encountered. The notion of an uninitialized variable is further elaborated in Section 5.1 where we shall also relax the last possibility for tests of the form `s = t`.

A specific interpretation of tests during a computation is crucial for our purposes. We stipulate the following.

Definition 1.

(1) If a test evaluates to `TRUE`, the computation upon reaching the test continues.

(2) If a test evaluates to `FALSE`, the computation upon reaching the test *fails*.

(3) If the subcomputation of a procedure (resp. function) call fails, then the computation upon reaching this procedure (resp. function) call *fails*.

(4) A finite, error-free computation *succeeds* if it does not fail.

Clause (3) explains how the failure propagates due to the use of functions and procedures. In particular, when the computation reaches a test like `f(1) = 0` and the call `f(1)` of the function `f` fails, the test fails, as well. We stress the fact that failure differs from a run-time error.

As a first example of the use of this extension consider the problem of checking whether a sequence represented by an array `a: ARRAY[1..M] OF INTEGER`, where $M \geq 2$, is ordered. The solution is immediate—it suffices to use the following statement:

```
FOR i := 1 TO M-1 DO a[i] <= a[i+1] END
```

When the array is not ordered, the above statement fails, and the loop is exited as soon as the least value of `i` is encountered for which the test `a[i] <= a[i+1]` fails.

2.2 Statements as Boolean Expressions

In the above definition we postulated that finite, error-free computations either succeed or fail. So it is natural to introduce the following definition.

Definition 2.

—If a computation of a sequence of statements succeeds, then we say that this statement sequence *evaluates to* `TRUE`.

—If a computation of a sequence of statements fails, then we say that this statement sequence *evaluates to* `FALSE`.

This definition allows us to use statement sequences as boolean expressions. We call this extension by **SBE**.

We postulate that the control variable of a `FOR` statement retains its value once the `FOR` statement is exited, be it due to a failure or due to a successful termination. This facility is used in the following program fragment that checks whether for two arrays `a` and `b` of type `ARRAY[1..N] OF INTEGER`, where $\texttt{N} \geq 1$, `a` precedes `b` in the lexicographic ordering:

```
NOT FOR i:= 1 TO N DO a[i] = b[i] END;
a[i] < b[i]
```

Operationally, this program fragment searches for the least `i` in the range `[1..N]` such that `a[i]` differs from `b[i]` (and fails if no such `i` exists) and then succeeds if and only if for this `i` the test `a[i] < b[i]` succeeds.

As another example of the use of **BES** and **SBE** consider the problem of counting the number of different elements in an array `x: ARRAY[1..M] OF CHAR`. A natural solution (although not the most efficient one) uses a statement as a boolean expression:

```
count := 0;
FOR i := 1 TO M DO
  IF FOR j := 1 TO i-1 DO x[i] <> x[j] END
  THEN count := count+1
  END
END
```

The identification of boolean expressions and statements allows us to apply negation to a statement. This, in combination with the provision for failures, allows us to realize within Alma-0 the powerful "negation as failure" mechanism of logic programming and Prolog. To illustrate its use consider the following two classical problems that deal with game trees.

By a *game tree* we mean a finite tree such that each leaf of it has an integer value. We call a node a *max-node* (resp. *min-node*) if it is at an odd (resp. even) level. We assume here that the root is at level 1 and that the levels are counted from the root downward.

Recall that the idea of the minimax search is as follows. Given a game tree, the values are assigned in a depth-first search manner to each node of the tree in such a way that the value of each nonleaf node `a` equals

—the minimum of the values of its children if `a` is a min-node, or

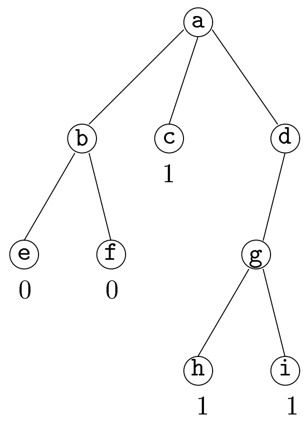

Fig. 1. A 0-1 game tree.

—the maximum of the values of its children if `a` is a max-node.

A *0-1 game tree* is a game tree such that each leaf of it has the value 1 or 0. As an example see the tree in Figure 1.

In what follows we call a node of a 0-1 tree game a *winning position* if by means of the minimax search

—the value 0 is assigned to it when it is a min-node, and

—the value 1 is assigned to it when it is a max-node.

In the 0-1 game tree of Figure 1 the internal nodes `a`, `b`, and `g` are the only winning positions.

From now on assume that the game tree is such that each leaf of it at an even level has the value 1 and at an odd level has the value 0. In other words, we assume that all leaves are losing positions. Under this assumption, the values 0 and 1 associated with the leaves of the tree do not need to be represented explicitly because they can be computed from the level of each leaf.

Problem 1. (*Minimax 0-1 Search*). Determine whether the root of a 0-1 game tree is a winning position using the minimax search (e.g., see Barr et al. [1981]).

To solve the problem we represent a 0-1 game tree by assuming that the labels of its nodes are elements of some further unspecified type `node` and by using a procedure `Move(x:node; VAR y:node)` the successive calls of which for a given node `x` generate in `y` upon backtracking all its direct descendants. For the 0-1 game of Figure 1 the code for procedure `Move` is provided in the next section, in which the programming constructs that support backtracking are introduced.

We define the procedure `Win` that solves this problem in such a way that if a node `a` is the winning position, then the call `Win(a)` succeeds, and otherwise it fails. The procedure `Win` is remarkably concise: it simply defines when a position is a winning one, namely when a move exists which leads to a losing, that is nonwinning, position:

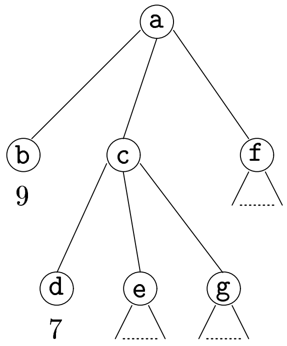

Fig. 2. A search tree for the α-β algorithm.

```
PROCEDURE Win(x: node);
  VAR y: node;
BEGIN
  Move(x,y);
  NOT Win(y)
END Win;
```

In this recursive procedure the base case appears when the internal call to the `Move` fails—then the corresponding call of `Win` also fails. It is useful to note that in this way we obtained a replica of the corresponding solution in Prolog (e.g., see Apt [1997, p. 302]).

Problem 2. (α-β *Search*). Compute the value of the root of a game tree using the α-β search (e.g., see Barr et al. [1981]).

Recall that the idea of the α-β search is that, in order to compute the value of a node, it is possible in some cases to identify nodes that cannot contribute to the solution, as a result of which some subtrees do not have to be explored.

As our solution at one point conceptually differs from the customary one, we explain the α-β search in more detail by means of the example in Figure 2, where the root `a` is a max-node (and consequently `b`, `c`, and `f` are min-nodes, and `d`, `e`, and `g` are max-nodes).

In order to compute the value for the root `a`, the α-β search recursively computes the values for all its children starting from the left. The values of α and β initially are equal to $-\infty$ and ∞ and are dynamically adjusted during the search. In particular, during the computation of the value of the min-node `c`, when the value `7` is found at node `d`, there is no more reason to compute the values of nodes `e` and `g`. Indeed, the value returned by node `c` cannot be bigger than `7` which is less than `9` already found at node `b`.

In our solution we exploit the use of failure to implement the procedure in a different (and simpler) way than the customary imperative solution.

In what follows (as is usually done) we dispense with the distinction between max-nodes and min-nodes by alternating the sign and position of α and β while switching levels. Now, during the computation of the value of node `c`, the value `val` returned by each of its children is tested against the current value of `beta`. When the test `val < beta` fails, the procedure call fails, and *no* value is returned. In our example, the opposite (`-9`) of the value found at node `b` is passed as argument `beta` to the invocation of the procedure search at node `c`. Therefore no value is returned by node `c`, because a failure occurs when the value `-7` returned by node `d` fails the test `val < beta` for `beta` equal to `-9`.

Notice that, in the program below, the computation of the value of each child is inside an `IF` statement, so after the failure at node `c`, the computation for node `a` continues with node `f` without getting any value from `c`.

Differently from the preceding `Win` procedure, we assume here to have at our disposal an explicit representation of the tree, together with the customary functions that allow us to traverse the given game tree, the meaning of which should be obvious.

```
PROCEDURE AlphaBeta(node: TreeNode; alpha, beta: INTEGER;
                    VAR val: INTEGER);
  VAR child: TreeNode;
BEGIN
  IF IsLeaf(node)
  THEN val := Value(node)
  ELSE
    child := FirstChild(node);
    WHILE child <> EmptyNode DO
      IF AlphaBeta(child,-beta,-alpha,val)
      THEN val < beta; alpha := Max(alpha,val)
      END;
      child := NextChild(node,child)
    END;
    val := alpha
  END
END AlphaBeta;
```

The difference with respect to the customary imperative solution is in the way the information that the value for node `c` does not have to be computed is carried. In the customary solution (e.g., see Barr et al. [1981]), when the search is interrupted, value `7` is assigned to node `c`, which is somewhat misleading because the actual value of `c` has not been computed and can differ from `7`.

In contrast, in our solution the search procedure for `c` automatically ends in a failure, which supplies the information that node `c` *fails* to contribute to the computation of the value of node `a`. In the initial call to `AlphaBeta` the value of `beta` must be assigned to a value, say `Maxint`, higher than all the values appearing in the leaves of the tree. Analogously, the value of `alpha` must be assigned to an appropriate value, say `Minint`. These settings ensure that no pruning (i.e., failure) takes place before the first value for `val` is computed, and therefore the initial call always succeeds and yields the desired value in the last actual parameter.

3. NONDETERMINISTIC STATEMENTS

Failures on their own can be used only as a means of evaluating a sequence of statements to `FALSE` in the **SBE** extension. In some situations it is useful to employ failures also to generate successive candidates that satisfy some conditions.

To this end we need some language constructs that introduce choice points and backtracking into the computational process.

3.1 ORELSE Statement

We begin by introducing an `ORELSE` statement with the following syntax:

```
EITHER <statement-sequence>
ORELSE <statement-sequence>
...
ORELSE <statement-sequence>
END
```

We denote this extension by **ORELSE**, and we refer to the parts of the `ORELSE` statement as *branches*.

The `ORELSE` statement introduces choice points to which the computation can return. With the introduction of the choice points the rules explaining the computation process given in Definitions 1 and 2 have to be modified to take into account the possibility of backtracking. In the operational semantics given in Section 8 a *choice point* is a pair formed by a statement sequence and an environment in which it is to be executed. The description of the implementation of a choice point will be given in Section 9.2.1.

We postulate the following.

Definition 3. If a computation of a sequence of statements fails, then *backtracking* takes place, which means that

(1) if no choice point exists, the computation fails;

(2) otherwise the control returns to the last created choice point. This implies that the environment is restored, so all the assignments performed since the creation of this choice point are undone.

We can now explain the computational interpretation of the `ORELSE` statement.

Definition 4. The computation of an `ORELSE` statement starts by creating a choice point. This choice point consists of the other branch if only two branches exist and otherwise of the `ORELSE` statement formed by the remaining branches. Then the computation proceeds through the first branch.

This definition implies that if the computation that started with some but not last branch eventually fails, possibly beyond the end of the `ORELSE` statement, backtracking takes place, and the computation resumes with the next branch in the state in which the previous branch was entered. In case the computation that started with the last branch fails, the `ORELSE` statement fails.

As an example consider the program fragment

```
EITHER x := x - 2*a; x > 0
ORELSE x > a; y := x
END
```

If the initial value of `x` is larger than `2*a`, the computation passes through the first branch and succeeds. In turn, if the initial value of `x` is between `a` and `2*a` the computation passes through the first branch and fails upon encounter of the test x > 0. Then backtracking takes place; the initial value of x is restored; and the computation passes through the second branch and eventually succeeds, assigning the initial value of x to y. Finally, if the initial value of x is less than a, both branches fail, and no value is assigned to y.

Consider now another example, where we assume that initially the value of x equals a positive number a:

```
EITHER y := x
ORELSE x > 0; y := -x
END;
x := x + b;
y < 0
```

Here the computation that passes through the first branch eventually fails upon encounter of the test y < 0, and backtracking takes place. The second branch of the ORELSE statement is then entered with the initial value of x restored, and eventually the whole computation succeeds, with x equal to a+b and y equal to -a.

Note that in the second example the failure occurs outside the scope of the ORELSE statement; that is, the backtracking takes place here *after* the control has left the ORELSE statement. The example shows that upon backtracking the assignments outside the scope of the ORELSE statement are also "undone."

This interpretation of the meaning of the ORELSE statement allows the user to write programs in which the creation of choice points and the testing of the selections made by them are done in separate parts of the program. Consider the following typical structure

```
Generate(x);
Test(x)
```

in which the first procedure generates successive values for x by the introduction of choice points and in which the second one tests these values. The correct functioning of this program is achieved only if the choice points remain active after the execution of the procedure Generate.[1]

As an example of use of the ORELSE statement, we now show the definition of the procedure Move of Problem 1 that corresponds to the tree of Figure 1.

```
TYPE node = (a,b,c,d,e,f,g,h,i);

PROCEDURE Move(x:node; VAR y:node);
BEGIN
  IF    x = a THEN EITHER y := b ORELSE y := c ORELSE y := d END
  ELSIF x = b THEN EITHER y := e ORELSE y := f END
  ELSIF x = d THEN y := g
  ELSIF x = g THEN EITHER y := h ORELSE y := i END
  ELSE  FALSE;
  END
END Move;
```

[1]This point will be further illustrated in Section 4.

3.2 SOME Statement

One of the limitations of the ORELSE statement is that it generates a number of choice points fixed in advance. In some situations, for example when processing an array, it is useful to generate choice points the number of which depends parametrically on some constants or is determined only at run-time.

This facility is realized by the **SOME** extension that provides the SOME statement with the following syntax:

```
SOME <ident> := <expression> TO <expression> DO
  <statement-sequence>
END
```

The intention is that the SOME statement is a "dual" of the FOR statement. In particular, given an integer variable i we wish

```
SOME i := 1 TO 10 DO T END
```

to be equivalent to

```
EITHER i := 1; T
ORELSE SOME i := 2 TO 10 DO T END
END
```

More precisely, we stipulate the following meaning of the SOME statement.

Definition 5. Let S be the statement SOME i := e1 TO e2 DO T END, where i is an integer variable, and in the current state e1 evaluates to an integer m1, and e2 evaluates to an integer m2. The following cases arise.

—m2 < m1. Then S is equivalent to FALSE.

—m2 = m1. Then S is equivalent to i := m1; T.

—m2 > m1. Then S is equivalent to

```
EITHER i := m1
ORELSE i := m1+1
...
ORELSE i := m2
END;
T
```

As in the case of the FOR statement we postulate that the control variable of the SOME statement retains its value once the SOME statement is exited, be it due to a success or due to a failure. Also, we assume for simplicity that the variable i is not modified in T.[2]

The next problem illustrates the use of a SOME-FOR combination.

Problem 3. (*Straight String Search*). Consider two arrays of characters, p (the *pattern*) and s (the *string*), declared respectively as variables of the following two types:

[2]This is not required but, like in the case of the FOR statement, is a common-sense restriction. In fact, a variable processed automatically should not be modified explicitly by the programmer.

```
Pattern =  ARRAY [0..M-1] OF CHAR;
String = ARRAY [0..N-1] OF CHAR;
```

with M $\leq$ N. Find the first occurrence of p in s.

The following procedure is a naive solution to this problem. It is much more straightforward than its imperative counterpart given in Wirth [1986, p. 60].

```
PROCEDURE StringMatch(p: Pattern; s: String): INTEGER;
  VAR i, j: INTEGER;
BEGIN
  SOME i := 0 TO N-M DO
    FOR j := 0 TO M-1 DO
      s[i+j] = p[j]
    END
  END;
  RETURN i
END StringMatch;
```

In turn, the following problem illustrates the use of a FOR-SOME combination.

Problem 4. (*Remarkable Sequence*). (See Coelho and Cotta [1988, p. 193].) Call a sequence of 27 elements *remarkable* if it consists of three 1's, three 2's, ..., three 9's arranged in such a way that for all $i \in [1..9]$ there are exactly i numbers between successive occurrences of i. For example, the sequence

```
(1,9,1,2,1,8,2,4,6,2,7,9,4,5,8,6,3,4,7,5,3,9,6,8,3,5,7)
```

is remarkable. Write a program that tests whether an array of 27 elements is a remarkable sequence.

The desired program is almost a verbatim specification of the problem (though not the most efficient solution).

```
TYPE Sequence = ARRAY [1..27] OF INTEGER;
PROCEDURE Remarkable(VAR a: Sequence);
  VAR i, j: INTEGER;
BEGIN
  FOR i := 1 TO 9 DO
    SOME j := 1 TO 25-2*i DO
      a[j] = i;
      a[j+i+1] = i;
      a[j+2*i+2] = i
    END
  END
END Remarkable;
```

The bound 25-2*i comes from the requirement that j+2*i+2 $\leq$ 27. In Section 5 we shall analyze the related problem of finding remarkable sequences.

Finally, we discuss a linear planning problem, known in the Artificial Intelligence literature as the propositional STRIPS problem (see Fikes and Nilsson [1971]). In propositional STRIPS, *actions* and *goals* are members of two (disjoint) alphabets of propositional letters. A STRIPS *action rule* is composed of an action and three sets of goals: the *preconditions*, the *add-list*, and the *delete-list*. A *state* is a set of goals. An action is *applicable* in a given state if all its preconditions are members of the state. The *result* of the application of an action in a current state is a new state where the goals in the add-list and the delete-list of the action are, respectively, added to and deleted from the current state. An *action library* is a set of action rules.

Problem 5. (*Propositional STRIPS Planner*). Given an action library, an initial state, and a final state, find a sequence of actions the application of which leads from the initial state to a state that includes the final state.

The above problem is PSPACE-complete (see Bylander [1991]) and is generally solved using backtracking algorithms. In particular, the so-called STRIPS algorithm works (nondeterministically) as follows: *guess* a goal g in the final state not already satisfied in the current state, *guess* an action a which has g in its add-list, and compute (recursively) the subplan p to reach the preconditions of a. If the algorithm reaches a state in which all goals in the final state are included, then the concatenation of the sequences $p \circ \langle a \rangle$ for all g chosen during the search provides the complete plan.

The STRIPS algorithm involves guessing (realized by backtracking) and consequently it is natural to implement it in Prolog. Such a Prolog implementation is provided, e.g., by Shoham [1994]. In this solution, due to lack of assignment in Prolog, various auxiliary variables are needed to store temporary values of goals and plans. On the other hand, implementation in traditional imperative languages is pretty cumbersome due to lack of facilities that support backtracking.

In contrast, in our language, we can use both guessing (realized by means of the `ORELSE` and `SOME` statements) and assignment; therefore we can produce a conceptually simpler and more readable solution.

We use lists of characters to represent sets of goals and actions. To deal with them, we define the type `List` the elements of which are characters, with various functions with their usual intuitive meaning: `Member`, `Head`, `Tail`, `Subset`, `Add`, `Delete`, and `Append`. We also assume that the calls to `Head` and `Tail` fail if the argument is the empty list.

```
TYPE
  ActionType =
    RECORD
      Name: CHAR;
      PreList: List;
      AddList: List;
      DelList: List
    END;
  ActionLib = ARRAY [1..NumActions] OF ActionType;

PROCEDURE ChooseGoal(VAR goal: CHAR; goals: List; state: List);
BEGIN
  EITHER
    goal := Head(goals);
    NOT Member(goal,state)
  ORELSE ChooseGoal(goal, Tail(goals), state)
  END
```

```
END ChooseGoal;

PROCEDURE ApplyRule(action: ActionType; VAR state: List; VAR plan: List);
BEGIN
  Delete(state, action.DelList);
  Add(state, action.AddList);
  Append(plan,action.Name)
END ApplyRule;

PROCEDURE AchieveGoal(goal: CHAR; lib: ActionLib; VAR forbidden: List;
                      VAR state: List; VAR plan: List);
  VAR i: INTEGER;
BEGIN
  SOME i := 1 TO NumActions DO
    NOT Member(lib[i].Name, forbidden);
    Member(goal,lib[i].AddList);
    Append(forbidden,lib[i].Name);
    Strips(state,lib[i].PreList,forbidden,plan,lib);
    ApplyRule(lib[i],state,plan)
  END
END AchieveGoal;

PROCEDURE Strips(VAR state: List; goals: List; forbidden: List;
                 VAR plan: List; lib: ActionLib);
  VAR goal: CHAR;
BEGIN
  PrintStatus(state, goals, forbidden, plan);
  IF NOT Subset(goals,state)
  THEN
    ChooseGoal(goal,goals,state);
    AchieveGoal(goal,lib,forbidden,state,plan);
    Strips(state,goals,forbidden,plan,lib)
  END
END Strips;
```

The planner is invoked by calling the recursive procedure `Strips` with the initial state as the `state` parameter, the final state as the `goals` parameter, the empty list for `forbidden` and for `plan`, and the given action library (which is not modified) as `lib`.

The list of forbidden actions is augmented by the `Append` procedure which is invoked within the body of the `AchieveGoal` procedure, to which the list is passed by variable. This way the selected action becomes forbidden for the subsequent calls of the `Strips` procedure, both in the body of `AchieveGoal` and in `Strips` itself.[3]

Notice that the guess of a goal, typically performed in Prolog using the query `member(Goal,Goals)` with `Goals` instantiated and `Goal` a variable, is implemented here by means of the `ORELSE` statement combined with recursion.

[3]This corrects what we believe is an omission in Shoham [1994] in which the selected action is added only in the first of the two calls, thus making divergence possible.

Notice also that the prescribed semantics of the ORELSE statement is essential for the correct functioning of the program. Namely, the ChooseGoal procedure creates choice points to which the control returns upon a possible failure that can occur also outside the scope of the ORELSE statement, either within the AchieveGoal procedure or within the recursive invocation of the Strips procedure.

At this point let us return to the semantics of **SBE** extension. By the introduction of nondeterminism a possibility now arises that choice points are created by statements used within conditions. In Alma-0 we stipulate that in such circumstances these choice points are discarded upon termination of the evaluation of the condition.

As an example, consider the following naive sorting algorithm:

```
WHILE SOME i:=1 TO M-1 DO a[i] > a[i+1] END
DO Swap(a[i], a[i+1]) END
```

The choice points created by the SOME statement are discarded here each time the first offending value of i is found.

4. BACKTRACKING AND CONTROL FLOW

4.1 COMMIT Statement

In the previous section we have seen two constructs that allow the user to introduce choice points. In large programs it is preferable to restrict the range of action of the choice constructs to some specific parts of the program. This would allow us to dispense with keeping track of too many choice points and would prevent unexpected behavior that could result from existence of active choice points created far back in the program.

To this aim we introduce the **COMMIT** extension which is realized by the COMMIT statement, with the following syntax:

```
COMMIT <statement-sequence>
END
```

The computational interpretation is a follows.

Definition 6. The statement COMMIT S END is executed in the same way as S, except that when the computation of S succeeds, all choice points created during the execution of S are removed. The choice points previously created are left unchanged.

For example, consider the following program fragment in which a is a positive number:

```
COMMIT
  EITHER x > 0; y := x
  ORELSE y := a
  END;
  y > 0
END;
y >= a
```

Its computation fails if the value of x is positive but smaller than a. Namely, when the control leaves the COMMIT statement the value of y is equal to the value of x, and the choice point created by the ORELSE statement is erased. Therefore backtracking to the second branch does not take place once the test `y >= a` fails. On the other hand, if the value of `x` is negative, the test `y > 0` inside the `COMMIT` statement fails, the second choice is performed, and the whole computation succeeds with value `a` for `y`.

Considering the `StringMatch` procedure of the *Straight String Search* problem (Problem 3) we can use the `COMMIT` statement, so write

```
COMMIT
  i := StringMatch(p,s)
END
```

if we wish to test only whether the pattern is present in the string, thus ignoring multiple occurrences. The `COMMIT` statement prevents the program from looking for different occurrences of `p` in `s` in case a later failure is detected.

As another example consider the following way of encoding the lexicographic ordering that is alternative to the one presented in Section 2.2:

```
COMMIT
  SOME i:= 1 TO N DO
    a[i] <> b[i]
  END
END;
a[i] < b[i]
```

Here `COMMIT` is necessary, and this is a rather subtle point. In fact, with the `COMMIT` statement this program fragment returns the value of the test `a[i] < b[i]` for *the least* `i` such that `a[i] <> b[i]`, whereas without the `COMMIT` statement it returns `TRUE` iff the test `a[i] < b[i]` succeeds for *some* `i` such that `a[i] <> b[i]`.

We can now rephrase the stipulation about the semantics of **SBE** extension given at the end of Section 3.2 by simply stating that each condition is surrounded by an implicit `COMMIT`.

4.2 `FORALL` Statement

Consider again the *Straight String Search* problem (Problem 3), and suppose now that we want to compute not just one, but *all* the occurrences of a pattern in a string. In this case we should explore the whole string, and not only the part of it up to the first successful occurrence.

In order to deal with this kind of situations, we introduce a new statement, called `FORALL`, that allows for exploring all the choice points generated by a given sequence of statements. More specifically, we use the following syntax:

```
FORALL <statement-sequence>
DO <statement-sequence>
END
```

and denote this extension by **FORALL**.

The computational interpretation of the `FORALL` statement is as follows.

Definition 7. The statement `FORALL S DO T END` is processed by executing `S` and `T` in sequence.

—If both `S` and `T` succeed, then

  —if there are no choice points created by `S`, the computation succeeds and continues in the initial state that is updated with the changes resulting from the execution of `T`;

  —otherwise the control returns to the last choice point created by `S` (as if a failure were encountered), and the state, as before, is updated with the changes resulting from the execution of `T`.

—If at certain moment `S` fails (even if `S` succeeds 0 times), then

  —if there are no choice points created by `S`, the computation succeeds and continues in a state in which the variables modified in `S` are restored to their values before the `FORALL` statement was entered;

  —otherwise backtracking takes place to the last choice point created by `S`.

—If `S` succeeds but `T` fails, then the computation fails.

The choice points created during each execution of `T` are discarded as soon as control returns to the successive choice point left within `S` or to a choice point created earlier. So, in effect, there is an implicit `COMMIT` statement surrounding `T`.

Thus statements within `S` are undone upon backtracking, whereas those in `T` are not, i.e., they have a *permanent* effect within and after the execution of the `FORALL` statement. This allows us to include in `T` any permanent operations that should be completed upon finding each solution to `S` (in logic programming they are generally implemented by means of input/output operations or `assert` and `retract`).

This permanent effect of `T` is relative to the environment of the `FORALL` statement. For example, if the `FORALL` statement is inside a branch of an `ORELSE` statement, and eventually a failure takes place, the state of the variables before entering a new branch is restored, thus removing the effects of the `DO` part of the `FORALL` statement.

To clarify these explanations consider some examples. The program fragment

```
y := 0;
x := 0;
FORALL
  x := x + a;
  EITHER x := x + b
  ORELSE x := x + c
  ORELSE x := x + d
  END
DO
  WRITELN(x);
  y := y + x
END;
```

prints the values of `a+b`, `a+c`, and `a+d` and assigns the value of `3*a+b+c+d` to `y`. As prescribed by the second case of Definition 7, when all branches have been processed, the computation succeeds, with `x` equal to its original value `0`, and leaves no choice points.

In turn, the following program fragment counts the number of occurrences of a pattern in a string:

```
count := 0;
FORALL
```

```
  k := StringMatch(p,s);
DO
  count := count + 1
END;
```

where the `StringMatch` function is defined in our solution to the *Straight String Search* problem (Problem 3).

The above two examples clarify the first two cases of Definition 7, which account for the natural way of using the `FORALL` statement (see also Problem 6 below). The last case, though not useful in practice, is illustrated by the following program fragment:

```
x := 0;
FORALL
  EITHER x := a
  ORELSE x := b
  END
DO
  x = b
END;
```

Here, after the first branch of the `ORELSE` statement is chosen, the `DO` part fails, and therefore the whole computation fails, restoring the value `0` for `x`.

Although we do not impose any syntactic restrictions on the form of the `FORALL` statement, its correct use imposes some common-sense limitations. Namely, no variable should be modified both in the body of the `FORALL` part and in the body of the `DO` part. In fact, these parts serve different purposes. In particular, the assignments in the `FORALL` part are meant to be nonpermanent, so they can be undone, while the ones in the `DO` part are meant to be permanent, so they should not be undone. This limitation resembles the already discussed common-sense restriction concerning the `FOR` and `SOME` statements that the loop control variable should not be modified within the loop body.

It is worth noting that the statement `FORALL S DO T END` is not equivalent to

```
EITHER S; T; FALSE
ORELSE TRUE
END
```

that mimics the so-called *failure-driven loop*, a standard technique in logic programming (e.g., see Sterling and Shapiro [1994]) used to deal with this kind of situations. The difference stems from the fact that in `FORALL S DO T END` the `T` statement is not undone upon backtracking. Also `FORALL COMMIT S END DO T END` is not equivalent to `S; T`, as the latter statement fails if `S` does. Moreover, the variables modified in `S` are not restored to their original values.

Let us consider now a more substantial example of the use of the `FORALL` statement.

Problem 6. (*Knapsack*). Given the real-valued objects $a_1, \ldots, a_n$ (*volumes*), $b_1, \ldots, b_n$ (*values*), and c (*capacity*), find the binary-valued objects $x_1, \ldots, x_n$ (*solutions*) such that $\sum_{i=1}^{n} b_i x_i$ is maximized subject to the constraint $\sum_{i=1}^{n} a_i x_i \leq c$.

We present here a solution that encodes a depth-first branch–and–bound algorithm. That is, the solution is constructed step by step by determining at each step i whether x_i is assigned to 1 or 0. Each partial solution is discarded if either

(1) it violates the capacity constraint or

(2) it cannot be completed to a solution better than the current best one.

The branch–and–bound algorithm is implemented by means of a `FORALL` statement over a `FOR` loop with an `ORELSE` statement inside.

Calling `volume` the total volume of the objects for which we have set x_i to 1, we can test condition 1 by checking if `volume` in the given partial solution is smaller or equal than the capacity. Calling `waste` the total value of the objects for which we have set x_i to 0, we can test condition 2 by checking if `waste` in the given partial solution is larger than the waste in the current (complete) best solution. Therefore, conditions 1 and 2 are taken care of in a very simple way by means of the tests `volume <= capacity` and `waste < total_value - current_best`, respectively.

Notice that condition 1 should be tested only when an object is chosen (when `solution[i] := 1`), whereas condition 2 should be tested only when an object is not chosen (when `solution[i] := 0`). These considerations bring us to the following program:

```
TYPE  RealVector = ARRAY [1..N] OF REAL;
    BinaryVector = ARRAY [1..N] OF [0..1];

PROCEDURE knapsack(volume, value: RealVector; capacity: REAL;
                   VAR solution: BinaryVector);
  VAR i: INTEGER;
    current_best, total_value, current_volume, waste: REAL;
    current_solution: BinaryVector;
BEGIN
  current_best := 0.0;
  total_value := 0.0;
  FOR i := 1 TO N DO
    total_value := total_value + value[i];
  END;
  current_volume := 0.0;
  waste := 0.0;
  FORALL
    FOR i := 1 TO N DO
      EITHER
        current_solution[i] := 1;
        current_volume := current_volume + volume[i];
        current_volume <= capacity;
      ORELSE
        current_solution[i] := 0;
        waste := waste + value[i];
        waste < total_value - current_best;
      END
    END
  DO
    current_best := total_value - waste;
```

```
    solution := current_solution;
  END;
END knapsack;
```

The assignment to the variable `current_best` is within the `DO` part of the `FORALL` statement, and therefore it is not undone upon backtracking. This is crucial for maintaining the current best solution while exploring different branches.

5. MULTIPLE USES OF A PROGRAM

In logic programming it is sometimes possible to use the same procedure for a number of different purposes. For example, the same program can be used both for testing a solution and for computing one. This multiple use of a single program is absent in the imperative programming paradigm. In this section we explain how this facility can be realized within our framework.

5.1 Generalization of Equality

By way of example reconsider the *Remarkable Sequence* problem (Problem 4) and suppose we wish to solve a more general problem.

Problem 7. (*Remarkable Sequence Revisited*). Find an array of 27 elements that forms a remarkable sequence in the sense of Problem 4.

To obtain a single solution to both problems we generalize the use of equality. In imperative programming languages a variable upon its declaration is usually either initialized to a default value or to some "garbage" value—an arbitrary value that happens to be present in the storage area allocated to the variable.

For our purposes it is important to be more precise. In what follows, we assume that a variable upon its declaration is *uninitialized* and remains so until a value of an expression is assigned to it. If this expression uses an uninitialized variable or this value lies outside the domain of the variable, then we postulate that a run-time error arises. Otherwise, from that moment on the variable is *initialized*. So in our approach an uninitialized variable has no value associated with it. This viewpoint is usually not adopted in imperative programming languages.

Further, we stipulate that if all the variables in an expression are initialized, then the expression has a *known* value, and otherwise it has an *unknown* value. Now we introduce the following more general interpretation of equality.

Definition 8. Consider a test `s = t`.

(1) Suppose both sides are expressions with known values. Then we treat it as in Definition 1.

(2) Suppose that
—one side, say `s`, is an uninitialized variable of a simple type,
—the other side, `t`, is an expression with known value, and
—their types are compatible.
Then we treat it as an *assignment*, which means that the value of `t` is assigned to `s`.

(3) The remaining cases yield a run-time error.

In particular, if both sides are expressions with unknown values (for example uninitialized variables), a run-time error arises. Note that—conforming to the logical interpretation—we treat here both sides of equality in a symmetric way.

We denote this extension by **EQ**. As we already mentioned in Section 1, **EQ** resembles a limited form of unification. The differences stem from the fact that in our case unification is allowed only for variables (of simple type), and it is not extended to compound terms. In addition, our equality operator includes arithmetic evaluation of the known side of the operator, which is not done while unifying terms in logic programming languages. This suggests that **EQ** actually mimics the `is` statement of Prolog. The difference is that `is` is not symmetric.

Before we proceed, we need to clarify a number of points. First, let us take a closer look at the interplay between the generalized use of equality and the call by variable (i.e., by reference) parameter mechanism. When a parameter of (for simplicity) a simple type is declared as a call by variable parameter and its value is computed by means of generalized equality, this equality can be used in two ways. If the actual parameter is an uninitialized variable, then it acts as an assignment and if the actual parameter is an initialized variable, then it acts as a test. As we shall see in the examples below, it is exactly this double use of equality that makes it possible to use the same procedure for a number of purposes.

Next, generalized equality introduces a possibility of creating side-effects during evaluation of tests and conditions. This leads to certain complications in case of some ill-designed programs. For example, logically `NOT (x = s)` is equivalent to `x <> s`, but this equivalence does not carry through to Alma-0. Indeed, if `x` is uninitialized and the value of `s` is known, the first statement assigns to `x` the value of `s` and fails, while the latter one yields a run-time error.

Finally, this generalized use of equality can in principle conflict with the prescribed meaning of it within Modula-2. But this could only happen if the original Modula-2 program used equality `x = t` (or `t = x`) within a condition with `x` uninitialized. So such a program would be certainly not a meaningful one.

An alternative, which at this stage we did not pursue, was to introduce another symbol, say `:=:`, for such a use of equality. Our generalized use of equality in a very limited way treats equalities as constraints. We shall return to this point in Section 10.2.

We can now return to the *Remarkable Sequence Revisited* problem (Problem 7). Thanks to the generalized use of equality the original program is now a solution to both problems, 4 and 7!

In this program the double role of equality as test and as assignment is now intertwined in a complex way. From the computational point of view the equalities in the `Remarkable` procedure serve now both to assign a value to an (uninitialized) subscripted variable and to test a value of an (initialized) subscripted variable. The assignments to the subscripted variables `a[j]`, `a[j+i+1]`, and `a[j+2*i+2]` that are generated by the equalities can be retracted at any later stage, if for some future value of `i` the tests `a[j] = i, a[j+i+1] = i, a[j+2*i+2] = i` fail for all values of `j` in `[1..25-2*i]`.

Note that the use of equality instead of assignment is crucial here. In the two most extreme cases, if the actual array parameter is completely uninitialized, the equalities are used both as assignments and tests, and if the actual array parameter is completely initialized, these equalities are used only as tests. An alternative program that uses only assignment and normal equality is more elaborate.

Consider now the following simple solution to the Eight Queens problem as another example.

Problem 8. (*Eight Queens).* Place eight queens on the chess board so that they do not attack each other.

The solution given below simply states that each queen should be placed in a legal field that does not come under attack by the already placed queens.

```
CONST N = 8;
TYPE board = ARRAY[1..N] OF [1..N];
PROCEDURE Queens(VAR x: board);
  VAR i,column,row: [1..N];
BEGIN
  FOR column := 1 TO N DO
    SOME row := 1 TO N DO
      FOR i := 1 TO column-1 DO
        x[i] <> row;
        x[i] <> row+column-i;
        x[i] <> row+i-column
      END;
      x[column] = row
    END
  END
END Queens;
```

In this solution the array `x` is declared as a `VAR` parameter, and the assignments to its elements take place by means of equalities. As a result, as already mentioned above, this procedure can be used in a number of different ways, other than just finding a solution.

First, it can also be used to test whether an array `a` *is* a solution. Indeed, if the actual array `a` is initialized before the call `Queens(a)`, then all the equalities `x[column] = row` become interpreted as tests.[4]

Second, this procedure can also be used to look for a *specific* solution. For example, to find a solution `a` to the Eight Queens problem such that `a[1] = 4` it suffices to write

```
a[1] = 4;
Queens(a)
```

And to find a solution `a` such that `a[1] > 4` it suffices to write

```
Queens(a);
a[1] > 4
```

etc. Finally, to count the number of solutions such that `a[1] > 4` we can write

[4]It is useful to point out that out of all the uses of the procedure `Queens` only this one requires that equality instead of assignment is used. Also, note that each variable `x[i]` is first used in an equality `x[column] = row`, so no run-time error can arise here.

```
i := 0;
FORALL
  Queens(a);
  a[1] > 4
DO i := i+1
END
```

So the procedure `Queens` can be used to compute, to test, to search for a specific solution, and to count the number of all solutions (that satisfy some property). In all these cases the text of the original procedure does not have to be changed. This is in contrast to the customary solution (e.g., see Wirth [1986, pp. 153–157]) which in each case has to be modified.

5.2 New Parameter Mechanism

We just noticed that the procedures `Remarkable` and `Queens` could be used both for testing and for computing. To this end it was crucial that their parameter (which is of an array type) was declared as a call by variable parameter.

In the case of simple types this double use of a single procedure is possible only to a limited extent because nonvariable expressions are also possible. For example, in the case of the `INTEGER` type, expressions such as `7` or `x + 7` can be passed as actuals. In this case only call by value is legal.

We now introduce a parameter-passing mechanism that overcomes this limitation and makes possible such a double use of procedures—for testing and for computing—also in case of simple types. We call this parameter mechanism *call by mixed form*, denote its use by the keyword `MIX`, and call this extension **MIX**. We stipulate the following.

Definition 9. Assume that the formal parameter is of a simple type.

—If the actual parameter is a variable, then it is passed by variable.

—If the actual parameter is an expression that is not a variable, its value is computed and assigned to a new variable v (generated by the compiler): it is v that is then passed by variable. So in this case the call by mixed form boils down to call by value.

Additionally, for compound types we postulate that call by mixed form coincides with call by variable. The reason is that for compound types no nonvariable expressions exist.

For example, if the actual parameter of a procedure `Proc` is an integer variable `x`, it is passed to `Proc` by variable, and if the actual parameter is `x + 7`, then it is passed to `Proc` by value. The latter takes place by replacing internally the call `Proc(x+7)` by the program fragment

```
VAR v: INTEGER;
BEGIN
  v := x+7;
  Proc(v)
END
```

So in the call by mixed form the decision whether a specific parameter is to be passed by variable or by value is determined for each procedure (or function) call separately and thus not on the basis of the procedure declaration, as is common for other type of parameters.

To see the advantages of the call by mixed form consider the following problem.

Problem 9. (*Linear Search*). Check if an integer `e` is present in an array of integers.

We write the solution as a procedure.

```
TYPE IntegerVector = ARRAY[1..N] OF INTEGER;

PROCEDURE Find(MIX e: INTEGER; a: IntegerVector);
  VAR i: INTEGER;
BEGIN
  SOME i := 1 TO N DO e = a[i] END
END Find;
```

Suppose now that `x` is an uninitialized integer variable and that `a` and `b` are initialized arrays of integers of type `IntegerVector`. Then

—the call `Find(7,a)` tests if 7 appears in `a`;

—the call `Find(x,a)` assigns upon backtracking successively all elements of `a` to `x`;

—the program fragment

```
Find(x,a);
Find(x,b)
```

tests if the arrays of integers `a` and `b` have an element in common; if so it computes such an element, and otherwise it fails;

—the program fragment

```
FORALL Find(x,a)
DO Find(x,b)
END
```

tests if all elements of `a` are also elements of `b`; if so it succeeds, and otherwise it fails;

—the program fragment

```
FORALL
  Find(x,a);
  Find(x,b)
DO
  WRITELN(x)
END
```

prints all elements that `a` and `b` have in common.

In the last three cases, the first occurrence of `x` is called by variable and the second by value. So, thanks to the fact that we declared the first parameter as a `MIX` parameter and used equality to assign values to it, we can use the procedure `Find` both to check whether an element is present in a given array and to generate all the elements of an array. Combining both types of calls we can build implicit loops.

The above instances of behavior of the `Find` procedure cannot be reproduced using the customary parameter mechanisms of Modula-2. Indeed, suppose that instead of the call by mixed form we would use call by value. Then if `x` were uninitialized, the call `Find(x,a)` would result in a run-time error, and if `x` were initialized, the program fragment `Find(x,a); Find(x,b)` would rather check if `x` occurs both in `a` and in `b`. If we used call by variable instead, the program fragment `Find(x,a); Find(x,b)` would exhibit the same behavior as for call by mixed form, but the call `Find(7,a)` would yield a compile-time error.

5.3 Testing the Status of a Variable

The additions discussed in Sections 5.1 and 5.2 relied in a crucial way on the distinction between initialized and uninitialized variables. In this section we go one step further and add to the language a relation that allows us to perform this test.

More specifically, we introduce a unary relation `KNOWN` with the following interpretation.

Definition 10.

—If `x` is a variable of a simple type, the test `KNOWN(x)` succeeds if and only if `x` is initialized.

—If `s` is an expression which is not a variable of a simple type, the test `KNOWN(s)` yields a compile-time error.

We denote this extension by **KNOWN**. As an example, following Sterling and Shapiro [1994, p. 176], consider the following procedure that computes the unknown element of the ternary relation representing the addition operator:

```
PROCEDURE Plus(MIX x,y,z: INTEGER);
BEGIN
  IF    KNOWN(x); KNOWN(y) THEN z = x+y
  ELSIF KNOWN(y); KNOWN(z) THEN x = z-y
  ELSIF KNOWN(x); KNOWN(z) THEN y = z-x
  END
END Plus;
```

For example, if we invoke `Plus(x,y,10)` with `x` uninitialized and `y` with value `7`, then the procedure assigns value `3` to `x`. Note that the use of the `MIX` parameter mechanism and of equality as an assignment is crucial here.

To illustrate another natural use of the `KNOWN` relation consider now the following variant of a problem from Colmerauer [1990].

Problem 10. (*Squares in the Rectangle.*) Cover an integer-sized $nx \times ny$ rectangle with squares $S_1, \ldots, S_m$ of integer sizes $s_1, \ldots, s_m$. “Covering” means that no two squares overlap and that the rectangle is completely filled in.

To solve this problem we use a backtracking algorithm that fills in all the cells of the rectangle one by one. For each cell, it checks if it is already covered by some square placed to cover a previous cell; if it is not covered, it looks for a square not already placed to be located with the top-left corner in the given cell. The algorithm backtracks when none of the available squares can cover the given cell without sticking out of the rectangle.

Backtracking is implemented by a `SOME` statement that checks for each square whether it can be put to cover a given cell. The solution is returned via two arrays `posX` and `posY` such that for square k (of size `sizes[k]`) `posX[k]`, `posY[k]` are the coordinates of its top-left corner.

The two equalities `posX[k] = i` and `posY[k] = j` are used both to construct the solution and to prevent a placed square to be used again in a different place.

We use the `AlreadyCovered` procedure to deal with cells that are covered by squares already used to fill other cells. For checking that a cell is already covered we look—by means of the `KNOWN` relation—for an "already placed" square that covers the cell. The call of `AlreadyCovered` is used as a test.

The variables `posX` and `posY` as `VAR` parameters allow us to use the program both to check a given solution or to complete a partial solution.

```
CONST NX = 33; NY = 32; (* size of the rectangle *)
    M = 9; (* number of small squares *)
TYPE  SquaresVector = ARRAY [1..M] OF INTEGER;

PROCEDURE AlreadyCovered(i, j: INTEGER; sizes: SquaresVector;
            VAR posX, posY: SquaresVector);
  VAR h  : INTEGER;
  BEGIN
    SOME h := 1 TO M DO
      KNOWN(posX[h]) AND KNOWN(posY[h]);
      (posX[h] <= i) AND (i < posX[h] + sizes[h]);
      (posY[h] <= j) AND (j < posY[h] + sizes[h])
    END
  END AlreadyCovered;

PROCEDURE Squares(sizes: SquaresVector;
        VAR posX, posY: SquaresVector);
  VAR i, j, k  : INTEGER;
  BEGIN
    FOR i := 1 TO NX DO
      FOR j := 1 TO NY DO
        IF NOT AlreadyCovered(i,j,sizes,posX,posY) THEN
          SOME k := 1 TO M DO
            sizes[k] + i <= NX + 1;
            sizes[k] + j <= NY + 1;
            posX[k] = i;
            posY[k] = j
          END
        END
      END
    END
  END Squares;
```

Note that this program does not use any assignment.

6. SUMMARY OF ALMA-0 FEATURES

In this article we described Alma-0 by discussing the extensions of Modula-2 that are included in it. We successively introduced the following nine extensions:

—**BES**: Add boolean expressions to statements.
—**SBE**: Add statement sequences to boolean expressions.
—**ORELSE**: Add the `ORELSE` statement.
—**SOME**: Add the `SOME` statement.
—**COMMIT**: Add the `COMMIT` statement.
—**FORALL**: Add the `FORALL` statement.
—**EQ**: Generalize equality.
—**MIX**: Introduce a new parameter mechanism: call by mixed form.
—**KNOWN**: Introduce the `KNOWN` relation, to test whether a variable of simple type is initialized.

At this stage, the following features of Modula-2 have been omitted in the current implementation of Alma-0:

—The `CARDINAL` type, sets, variant parts in records, open array parameters, procedure types, and pointer types.
—The `CASE`, `WITH`, `LOOP`, and `EXIT` statements.[5]
—Nested procedures.
—Modules, and therefore the `EXPORT` and `IMPORT` declarations.

It is worth remarking that these features have been omitted only to keep the implementation simple, and they will be considered for future improvements of the language. We do not expect that these features will introduce any additional problems at the implementation level.

7. DECLARATIVE SEMANTICS

In what follows we introduce two semantics for two fragments of Alma-0. The one presented in this section is declarative and is applicable only to the programs built out of a limited number of constructs that do not involve assignment. In the next section we present an alternative, operational, semantics for a larger subset of Alma-0.

Alma-0 has been designed with the view of promoting declarative programming. As this term is often used to denote different things, let us clarify that in the context of this article we consider a program declarative if its meaning can be described by means of a logical formula that can be obtained by means of a syntax directed translation. We call then this formula the *declarative interpretation* of the program. By assigning to this formula its semantic meaning that agrees with the operational semantics of the original program we obtain *declarative semantics* of the program under consideration.

Consider now Table I, in which we denote by $\mathcal{T}(S)$ the translation of the program S and where B denotes a primary boolean expression. Several remarks are in order.

First, the logical language should be extended to allow subscripted variables (like in Marcus [1996]) to render correctly the use of these variables. For brevity, we omit here a description of the details of this extension.

[5]Note, however, that Modula-2 statement `LOOP S; IF B THEN EXIT END; T END` can be be modeled in Alma-0 as `WHILE S; NOT B DO T END`.

Table I. Declarative Interpretation

Language construct	Logical formula
B	B
NOT S	$\neg\mathcal{T}(S)$
$S_1; S_2$	$\mathcal{T}(S_1) \wedge \mathcal{T}(S_2)$
IF T THEN S END	$\mathcal{T}(T) \rightarrow \mathcal{T}(S)$
IF T THEN S_1 ELSE S_2 END	$(\mathcal{T}(T) \wedge \mathcal{T}(S_1)) \vee (\neg\mathcal{T}(T) \wedge \mathcal{T}(S_2))$
EITHER S_1 ORELSE S_2 END	$\mathcal{T}(S_1) \vee \mathcal{T}(S_2)$
FOR $i := 1$ TO n DO S END	$\forall i \in [1..n]\mathcal{T}(S)$
SOME $i := 1$ TO n DO S END	$\exists i \in [1..n]\mathcal{T}(S)$
FORALL S DO T END	$\forall \mathbf{x}(\mathcal{T}(S) \rightarrow \mathcal{T}(T))$ (where $\mathbf{x}$ is the list of all free variables of $\mathcal{T}(S)$)

Second, the semantics of formulas of this logical language has to differ from that of the customary first-order logic. For example, due to the use of generalized equality the programs `x = 0; y = x` and `y = x; x = 0` are not equivalent. Consequently, the conjunction $\wedge$ is not commutative. Further, the scope of both bounded quantifiers in the formulas $\forall i \in [1..n]\mathcal{T}(S)$ and $\exists i \in [1..n]\mathcal{T}(S)$ should extend beyond $\mathcal{T}(S)$ to render correctly the meaning of the FOR and SOME statements.

To illustrate the problem consider the task of finding the number of the first all-zero row of an `N * N` matrix `a` of integers, if any.[6]

In Alma-0 it can be easily encoded as follows, where for the sake of further discussion we introduced an integer variable `found` and used an unspecified statement `S` that should deal with the case when no all-zero row exists:

```
EITHER
  SOME i := 1 TO N DO
    FOR j := 1 TO N DO
      a[i,j] = 0
    END
  END;
  found = i
ORELSE
  S
END
```

This program gets translated to the formula

$$((\exists i \in [1..N]\, \forall j \in [1..N]\, a[i,j] = 0) \wedge found = i) \vee \mathcal{T}(S).$$

With the customary interpretation of the scope of the quantifiers, the final occurrence of i is not bound, while the semantics of the SOME statement stipulates that this occurrence of i is within the scope of the $\exists i \in [1..N]$ quantifier.

To see the arising complications assume now that in the program the variable `found` is initialized to some value in the range `[1..N]`. Then this program checks whether $a[found, j] = 0$ holds for all j in $[1..N]$.

[6]This problem is taken from a contribution to ACM Forum in *Communications of the ACM*, March 1987, pp. 195-196 by F. Rubin. It generated a lot of controversy, including a response by E.W. Dijkstra in the August 1987 issue, because of Rubin's claim that the most natural solution involves a GOTO statement.

Such a logic interpretation can be achieved by reconsidering the resulting formula after the translation process has been completed. At this stage the bounded quantifiers could be moved to other places within the formula (like outside of the conjunction in the above formula) to ensure the correct scope. Alternatively, a larger scope could be postulated by assuming that the bounded quantifiers bind all occurrences of the quantified variable till the end of each disjunct.

These considerations show that our future work on semantics of the introduced logical language could profit from Groenendijk and Stokhof [1991] where an alternative semantics of first-order logic is provided. In this semantics both the connectives and the quantifiers obtain a different, dynamic, interpretation that better suits their use for natural language analysis.

Third, the occurrences of the `SOME` and `FOR` statements within a condition should be translated differently because of the implicit `COMMIT` statement surrounding each condition. Consider for example the program

```
IF
  SOME i:= 1 TO N DO
    FOR j := 1 TO N DO
      a[i,j] = 0
    END
  END
THEN
  found = i
END
```

again with the variable `found` initialized. Because the choice points created by a statement used within a condition are discarded upon termination of the evaluation of the condition (see the end of Section 4.1), this program tests whether `found` is the *least* value i in the range $[1..n]$ for which $a[i,j] = 0$ holds for all j in $[1..N]$ (assuming such a value exists).

Consequently, its correct declarative interpretation is obtained by means of the formula

$$\mu i : i \in [1..n] \wedge \forall j \in [1..N]\, a[i,j] = 0 : found = i$$

where the binding operator $\mu i : \phi : \psi$ stands for

"if ϕ holds for some value of i, then ψ holds for the least such value of i."

In general, a program of the form

```
IF
  SOME i:=1 TO n DO S END; T
THEN U
END
```

should be translated to the formula

$$\mu i : i \in [1..n] \wedge \mathcal{T}(S) \wedge \mathcal{T}(T) : \mathcal{T}(U).$$

Similar considerations hold for the `FOR` statement.

Fourth, this declarative interpretation does not deal correctly with equality used as assignment within a condition of the conditional statements. This has to do with the fact that assignments used in conditions have a permanent effect. For example, given an uninitialized variable `x`, the statement `IF NOT (x = 0) THEN TRUE END; y = x` assigns to `y` the value `0`, but this cannot be deduced from its declarative interpretation $(\neg(x = 0) \rightarrow \texttt{TRUE}) \wedge y = x$.

Fifth, this view of declarative programming is very restrictive, since it rules out programs involving the `WHILE` and `REPEAT` loops and recursion. By admitting in the logical language some form of the least fixpoint operator (in the style of μ-calculus of Scott and de Bakker [1969]) we could also assign a declarative interpretation to programs involving these constructs, in particular to programs involving procedure declarations and procedure calls. However, in presence of negation and recursion a problem arises how to associate then a declarative semantics to the resulting formulas, like to the formula $p \leftrightarrow \neg p$ representing the procedure

```
PROCEDURE p;
BEGIN
  NOT p
END p;
```

These difficulties are analogous to the ones that motivated the study of negation in logic programming (e.g., see Apt and Bol [1994] for a survey of these issues).

Using the above translation process we can assign to several programs here discussed a logical formula that represents their declarative interpretation. By way of example take our solution to the *Eight Queens* problem (Problem 8). The following formula constitutes its declarative interpretation:

$$\begin{aligned}\phi(x) \equiv\ & \forall column \in [1..N]\, \exists row \in [1..N]\, \forall i \in [1..column - 1] \\ & (x[i] \neq row\ \wedge \\ & \ x[i] \neq row + column - i\ \wedge \\ & \ x[i] \neq row + i - column\ \wedge \\ & \ x[column] = row)\end{aligned}$$

In turn, consider the following program

```
FORALL
  queens(x);
  x[1] > 4
DO
  EITHER x[2] < 4 ORELSE x[3] < 4 END
END
```

that tests whether for all solutions x to the Eight Queens problem such that $x[1] > 4$ also $x[2] < 4$ or $x[3] < 4$ holds. Its declarative interpretation consists of the following formula:

$$\forall x((\phi(x)\ \wedge\ x[1] > 4) \rightarrow (x[2] < 4\ \vee\ x[3] < 4))$$

The right-hand side of Table I determines a logical language that could be used to specify programs. By using this table we could translate a specification written in this language into a program that meets this specification. As an, admittedly contrived, example consider the formula

$$\forall i \in [1..N] \exists j \in [1..N]\ b[j] = a[i]$$

that specifies that an array b is a permutation of an array a. (Note that this specification is correct only if a does not contain repeated elements.) It translates into the following program that given an array `a` with no repeated elements generates in `b` (upon backtracking) all its permutations:

```
FOR i := 1 TO N DO
  SOME j := 1 TO N DO
    b[j] = a[i]
  END
END
```

However, such a "reverse translation" cannot be used in an undiscriminate way, as it can yield programs that lead to run-time errors. As an example consider the following most natural specification of the Eight Queens problem:

$$\forall i \in [1..N-1]\, \forall j \in [i+1..N]\, (x[i] \neq x[j] \wedge x[i] \neq x[j]+j-i \wedge x[i] \neq x[j]+i-j)$$

Its reverse translation yields a program that for an uninstantiated array x causes a run-time error because the test $x[i] \neq x[j]$ involves uninstantiated variables.

8. OPERATIONAL SEMANTICS

We now move on to the presentation of operational semantics in the style of Hennessy and Plotkin [1979]. This semantics provides a better insight into the operational aspects of the introduced language constructs. An interesting aspect of the semantics here provided is that it is *executable*, i.e., one can use it to execute a program starting in a given initial state. In this way we could test it by executing it on a number of test programs, including the ones presented here.

The work discussed here is a summary of a larger effort, reported in Brunekreef [1998], in which an operational semantics in the same style has been provided to a substantially larger subset of Alma-0. Here we limit ourselves to a subset that involves the most relevant features of the language.

An operational semantics for a simple imperative language in the style of Hennessy and Plotkin [1979] involves pairs that consist of two components: a program and an environment. Alma-0 extends an imperative language by facilities that introduce "don't know" nondeterminism. This is handled in the semantics by introducing a third component that allows us to manage stacks of choice points. The resulting triples are called below *configurations*.

Before we proceed we provide a short explanation of the ASF+SDF system that was used to define this semantics.

8.1 ASF+SDF Meta-environment

The ASF+SDF Meta-environment of Klint [1993] is an interactive development environment for the generation of interactive programming environments. The generation process is controlled by the definition of a programming language, which may include such features as syntax definition/checking, type checking, prettyprinting, and semantics of programs.

SDF is a shorthand for Syntax Definition Formalism. In SDF both the lexical syntax and the context-free syntax of a language are specified in an algebraic style. ASF is a shorthand for Algebraic Specification Formalism. In ASF any function may be specified on terms that are constructed according to the syntax defined in an SDF specification. The ASF+SDF specifications have a modular structure. Different parts of a specification can be written down in separate modules. A module can be imported by another module.

ASF+SDF specifications are executable. This is achieved by transforming the algebraic equations into a term–rewriting system. In the specifications it is possible to use so-called default equations. A default equation is applied in case no other equation is applicable to a particular term. A more extensive introduction to the ASF+SDF Meta-environment can be found in van Deursen et al. [1996]. The ASF+SDF Meta-environment runs on Unix platforms.

In the presentation below we first discuss the syntax of the considered subset of Alma-0, review several predefined modules, and finally present the axioms and rules that define the semantics. These rules are given in a LaTeX format that is automatically generated by an "ASF+SDF–to–LaTeX" program.

8.2 Syntax

In what follows we consider statements defined by the syntactic category `Stat` (for statements) using the syntactic categories `Var` (for variables), `Exp` (for expressions), and `Bool` (for boolean expressions) that are further unspecified, and the syntactic category `StSeq` (for statement sequences).

```
Stat ::= Var '':='' Exp |
         Bool |
         KNOWN Var |
         IF StSeq THEN StSeq ELSE StSeq END |
         WHILE StSeq DO StSeq END |
         EITHER StSeq ORELSE StSeq {ORELSE StSeq} END |
         FORALL StSeq DO StSeq END |
         COMMIT StSeq END
StSeq ::= {Stat '';''} Stat
```

This subset abstracts away from a number of crucial aspects of Alma-0. In fact, in the syntactic definition of Alma-0 there is no distinction between expressions, boolean expressions, and statements. Consequently, it is syntactically possible to assign a statement to a variable, something that is semantically correct only if the variable is of type `BOOLEAN`. In the operational semantics that follows these issues are ignored.

Also, we assume that the program evaluated by the semantics is type correct and that during its evaluation no run-time errors arise.

It is straightforward to specify the syntax defined above in SDF. We omit this specification.

8.3 Predefined Modules

In what follows we shall assume the following ASF+SDF modules.

—Basic modules defining integer constants, boolean constants, and the customary operations on these constants.

—The module **Environments** that defines an "environment" for storing and retrieving variable values. An environment records the bindings of values to variables. It is a list of atomic environments, where each atomic environment $x \mapsto v$ binds a value v to a variable x. In this module the following operations on environments are defined:

—$\mathcal{E}(x)$: lookup the value of variable x in environment $\mathcal{E}$.

—$\mathcal{E}_1 \rhd \mathcal{E}_2$: destructively update environment $\mathcal{E}_2$ with environment $\mathcal{E}_1$. The bindings in $\mathcal{E}_2$ for variables which have a binding in $\mathcal{E}_1$ are discarded by this operation, e.g., $([x \mapsto v] \rhd \mathcal{E})(x)$ is v and not the value of x in $\mathcal{E}$.

—$\mathrm{def}(\mathcal{E}, x)$: determine whether variable x is defined in environment $\mathcal{E}$.

—The module **Values** that declares integer and boolean constants as admitted values for an environment. Furthermore, this module defines equality of values by means of the function *eq*.

—The module **Stack** that specifies a simple generic stack with the customary operations *push*, *pop*, and *top*. The symbol $\perp$ denotes the empty stack, and the operation $\bowtie$ specifies the constructor function for the stack. This module is needed to manage the stack of choice points (defined below) created by the nondeterministic statements of Alma-0.

—The module **Configuration** that manages "configurations." A *configuration* is a triple

$$\ll S, \mathcal{E}, C \gg$$

that contains a statement sequence S, an environment $\mathcal{E}$, and a stack of choice points C. In turn, a *choice point* is a pair $\prec S, \mathcal{E} \succ$ that contains a statement sequence S and an environment $\mathcal{E}$. These data structures are specified in this module. Furthermore, we have two functions (*fst* and *snd*) that yield respectively the first and the second element of a choice point.

The full description of these modules can be found in Brunekreef [1998] and is omitted.

8.4 Semantics

The core of the definition of the Alma-0 semantics consists of the specification of two functions: the function *eval*, defining the evaluation of an expression, and the function *sem*, defining the semantics of a statement sequence.

The function *eval* has two arguments: the expression to be evaluated and the environment. The function produces a pair with a new environment (recall that in Alma-0 by virtue of the **SBE** extension an assignment can be a part of an expression, like in `(x:=1) AND TRUE`) and the result of the evaluation. The rules defining the evaluation of expressions are omitted with the exception of the following ones.

8.4.1 *The **EQ** Extension.* This feature of Alma-0 is specified by three rules. In their conditions the boolean function *uninitVar* is used. This function indicates whether an expression equals an uninitialized variable. We omit the rules defining this function.

The first **EQ** rule deals with the possibility (1) of Definition 8: both sides of the equality are expressions with known values. We have then the usual equality test:

$$\frac{\texttt{uninitVar}(e_1, \mathcal{E}) = \texttt{false},\ \texttt{uninitVar}(e_2, \mathcal{E}) = \texttt{false}, \quad \texttt{eval}[\![e_1]\!](\mathcal{E}) = <\mathcal{E}_1, v_1>,\ \texttt{eval}[\![e_2]\!](\mathcal{E}_1) = <\mathcal{E}_2, v_2>}{\texttt{eval}[\![e_1\ =\ e_2]\!](\mathcal{E})\ =\ <\mathcal{E}_2, \texttt{eq}(v_1, v_2)>}$$

The second rule deals with the possibility (2) of Definition 8: the left-hand side of the equality test is an uninitialized variable, and the right-hand side is an expression with known value. Then the value of the expression at the right-hand side is assigned to the variable by applying an assignment statement and the function *sem*:

$$\frac{\texttt{uninitVar}(e_1, \mathcal{E}) = \texttt{true},\ \ e_1 = x\ \ ,\ \texttt{sem}(\ll x := e_2, \mathcal{E}, \perp \gg) = \ll\ , \mathcal{E}_1, \perp \gg}{\texttt{eval}[\![e_1\ =\ e_2]\!](\mathcal{E})\ =\ <\mathcal{E}_1, \texttt{true}>}$$

The third rule is the symmetric counterpart of the second one and is omitted.

8.4.2 *The* KNOWN *Statement.* This statement is a boolean expression. It is checked, using the environment, if a variable is initialized.

$$\frac{\texttt{uninitVar}(x\ \ , \mathcal{E}) = \texttt{false}}{\texttt{eval}[\![\texttt{KNOWN}(x)\ \ ]\!](\mathcal{E})\ =\ <\mathcal{E}, \texttt{true}>}$$

$$\texttt{eval}[\![\texttt{KNOWN}(x)\ \ ]\!](\mathcal{E})\ =\ <\mathcal{E}, \texttt{false}> \quad \texttt{otherwise}$$

The second rule is a default equation. By definition, it is applied in case no other rule is applicable to a particular term.

8.4.3 *Handling of Success and Failure.* We continue with the definition of the semantics of the program statements using the function *sem*. The *sem* function operates on a *sequence* of program statements, denoting the still–to–be–executed part of the program. Together with the environment and a stack of choice points, this statement sequence forms a configuration triple (see Section 8.3). The *sem* function takes as input a configuration and produces a new configuration with the sequence of remaining program statements, a new environment, and a new stack of choice points. The recursive application of the *sem* function to the input configuration mimics the program computation and yields the semantics of the initial sequence of statements.

More precisely, if a computation succeeds in the sense of part (4) in Definition 1, it results in a configuration with the empty statement sequence. This is specified by the following axiom:

$$\texttt{sem}(\ll\ , \mathcal{E}, C \gg) = \ll\ , \mathcal{E}, C \gg$$

In turn, if a computation fails in the sense of part (1) in Definition 3, a nonempty statement sequence is produced to which none of the rules for the *sem* function applies and to which backtracking is not possible (the stack of choice points is empty).

A failure is indicated by a nonempty statement sequence S^+ as the first element of the configuration triple. Together with the second element of the configuration triple (the environment), the first statement of S^+ reveals the cause of the failure. This is specified by a default equation for the function *sem*:

$$\texttt{sem}(\ll S^+, \mathcal{E}, \perp \gg) = \ll S^+, \mathcal{E}, \perp \gg \quad \texttt{otherwise}$$

8.4.4 *Assignment.* The assignment is dealt with in the customary way. The expression at the right-hand side of the assignment is evaluated; its value is assigned to the variable at the left-hand side; and the environment is updated.

$$\frac{\texttt{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1, v>,\ \ [x \mapsto v] \triangleright \mathcal{E}_1 = \mathcal{E}_2}{\texttt{sem}(\ll x := ex;\, S, \mathcal{E}, C \gg) \ = \ \texttt{sem}(\ll S, \mathcal{E}_2, C \gg)}$$

8.4.5 *The* **BES** *Extension.* The use of a boolean expression as statement is dealt with by evaluating the boolean expression with the function *eval*. If the outcome is true, the computation continues in the new environment.

This corresponds to part (1) of Definition 1.

$$\frac{\texttt{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1, \texttt{true}>}{\texttt{sem}(\ll ex;\, S, \mathcal{E}, C \gg) \ = \ \texttt{sem}(\ll S, \mathcal{E}_1, C \gg)}$$

If the outcome is false, two cases need to be distinguished.

Case 1. The stack of choice points is empty. Then the computation fails but changes in the environment are retained. (This is necessary in case the boolean expression is used within a condition.) This corresponds to part (1) of Definition 3.

$$\frac{\texttt{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1, \texttt{false}>}{\texttt{sem}(\ll ex;\, S, \mathcal{E}, \perp \gg) \ = \ \ll ex;\, S, \mathcal{E}_1, \perp \gg}$$

Case 2. The stack of choice points is not empty. Then backtracking takes place. This corresponds to part (2) of Definition 3.

$$\frac{\texttt{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1, \texttt{false}>}{\texttt{sem}(\ll ex;\, S, \mathcal{E}, CP \bowtie C \gg) \ = \ \texttt{sem}(\ll \texttt{fst}(CP), \texttt{snd}(CP), C \gg)}$$

8.4.6 *The* IF *Statement:* IF T THEN S_1 ELSE S_2 END. The condition, i.e., the statement sequence T, is evaluated using the function *sem*. If the computation succeeds (that is, the result is a configuration with the empty statement sequence), the statement sequence in the THEN branch is evaluated. Otherwise, the statement sequence in the ELSE branch is evaluated.

$$\frac{\texttt{sem}(\ll T, \mathcal{E}, \perp \gg) = \ll\ , \mathcal{E}_1, C_1 \gg}{\texttt{sem}(\ll \texttt{IF}\ T\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2\ \texttt{END};\, S_3, \mathcal{E}, C \gg) \ = \ \texttt{sem}(\ll S_1;\, S_3, \mathcal{E}_1, C \gg)}$$

$$\frac{\texttt{sem}(\ll T, \mathcal{E}, \perp \gg) = \ll S^+, \mathcal{E}_1, \perp \gg}{\texttt{sem}(\ll \texttt{IF}\ T\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2\ \texttt{END};\, S_3, \mathcal{E}, C \gg) \ = \ \texttt{sem}(\ll S_2;\, S_3, \mathcal{E}_1, C \gg)}$$

Note that, conforming to the point discussed at the end of Section 3.2, the remaining statement sequence is executed in the new environment $\mathcal{E}_1$ generated by T, but with respect to the *initial* stack of choice points C.

8.4.7 *The* `WHILE` *Statement:* `WHILE` T `DO` S `END`. The condition is evaluated. Depending on the outcome, the loop is unrolled one step or skipped.

$$\frac{\mathtt{sem}(\ll T, \mathcal{E}, \perp \gg) = \ll\ , \mathcal{E}_1, C_1 \gg}{\begin{array}{l}\mathtt{sem}(\ll \mathtt{WHILE}\ T\ \mathtt{DO}\ S_1\ \mathtt{END};\ S_2, \mathcal{E}, C \gg) \ = \\ \mathtt{sem}(\ll S_1;\ \mathtt{WHILE}\ T\ \mathtt{DO}\ S_1\ \mathtt{END};\ S_2, \mathcal{E}_1, C \gg)\end{array}}$$

$$\frac{\mathtt{sem}(\ll T, \mathcal{E}, \perp \gg) = \ll S^+, \mathcal{E}_1, \perp \gg}{\mathtt{sem}(\ll \mathtt{WHILE}\ T\ \mathtt{DO}\ S_1\ \mathtt{END};\ S_2, \mathcal{E}, C \gg) \ = \ \mathtt{sem}(\ll S_2, \mathcal{E}_1, C \gg)}$$

Here the same remarks concerning the new environment and the initial stack of choice points apply as in the case of the `IF` statement.

8.4.8 *The* `ORELSE` *Statement:* `EITHER` S_1 `ORELSE` S_2 {`ORELSE` U} `END`. The semantics formalizes Definition 4.

Case 1. The `ORELSE` statement consists of two branches. Then the computation continues with the first branch, and the second branch is pushed on the stack.

$$\begin{array}{l}\mathtt{sem}(\ll \mathtt{EITHER}\ S_1\ \mathtt{ORELSE}\ S_2\ \mathtt{END};\ S_3, \mathcal{E}, C \gg) \\ = \ \mathtt{sem}(\ll S_1;\ S_3, \mathcal{E}, \mathtt{push}(\prec S_2;\ S_3, \mathcal{E} \succ\ , C) \gg)\end{array}$$

Case 2. The `ORELSE` statement consists of more than two branches. Then the computation continues with the first branch, and the `ORELSE` statement formed by the remaining branches is pushed on the stack.

$$\begin{array}{l}\mathtt{sem}(\ll \mathtt{EITHER}\ S_1\ \mathtt{ORELSE}\ S_2\ \mathtt{ORELSE}\ U\ \mathtt{END};\ S_3, \mathcal{E}, C \gg) \\ = \ \mathtt{sem}(\ll S_1;\ S_3, \mathcal{E}, \mathtt{push}(\prec \mathtt{EITHER}\ S_2\ \mathtt{ORELSE}\ U\ \mathtt{END};\ S_3, \mathcal{E} \succ\ , C) \gg)\end{array}$$

where U is the remaining part of the `ORELSE` statement.

8.4.9 *The* `COMMIT` *Statement:* `COMMIT` S `END`.

Case 1. The computation of S succeeds. Then the computation continues without the modification of the stack. This corresponds to Definition 6.

$$\frac{\mathtt{sem}(\ll S, \mathcal{E}, \perp \gg) = \ll\ , \mathcal{E}_1, C_1 \gg}{\mathtt{sem}(\ll \mathtt{COMMIT}\ S\ \mathtt{END};\ S_1, \mathcal{E}, C \gg) \ = \ \mathtt{sem}(\ll S_1, \mathcal{E}_1, C \gg)}$$

Case 2. The computation of S fails. Then backtracking takes place.

$$\frac{\mathtt{sem}(\ll S, \mathcal{E}, \perp \gg) = \ll S^+, \mathcal{E}_1, \perp \gg}{\mathtt{sem}(\ll \mathtt{COMMIT}\ S\ \mathtt{END};\ S_1, \mathcal{E}, CP \bowtie C \gg) \ = \ \mathtt{sem}(\ll \mathtt{fst}(CP), \mathtt{snd}(CP), C \gg)}$$

Here and elsewhere the case of the backtracking with the empty stack of choice points is taken care of by the default equation introduced at the end of Section 8.4.3.

8.4.10 *The* `FORALL` *Statement:* `FORALL` S `DO` T `END`. The semantics of the `FORALL` statement is defined in a separate function *semFA*, specified below. A separate function is needed because evaluation of the `FORALL` statement requires a local stack of choice points, generated by the statement sequence S. Within the context of the *semFA* function, the configuration stack is used for this local stack.

The function *semFA* specifies the semantics of an isolated `FORALL` statement. In its description we closely mimic Definition 7, though we found it convenient to change the order of the cases.

Case 1. The computation of S fails.

Subcase 1.1. The stack of choice points is empty.

Then the `FORALL` statement is skipped. This means that the *semFA* function returns a configuration with the empty statement sequence, the initial environment, and the empty stack.

$$\frac{\texttt{sem}(\ll S, \mathcal{E}, \perp \gg) = \ll S^+, \mathcal{E}_1, \perp \gg}{\texttt{semFA}(\ll \texttt{FORALL } S \texttt{ DO } T \texttt{ END}, \mathcal{E}, \perp \gg) \;=\; \ll\, , \mathcal{E}, \perp \gg}$$

Subcase 1.2. The stack of choice points is not empty.

Then backtracking takes place by selecting the next choice point from the stack created by S.

$$\frac{\texttt{sem}(\ll S, \mathcal{E}, \perp \gg) = \ll S^+, \mathcal{E}_1, \perp \gg}{\begin{array}{l}\texttt{semFA}(\ll \texttt{FORALL } S \texttt{ DO } T \texttt{ END}, \mathcal{E}, CP \bowtie C \gg) \;=\\ \texttt{semFA}(\ll \texttt{FORALL } \texttt{fst}(CP) \texttt{ DO } T \texttt{ END}, \texttt{snd}(CP), C \gg)\end{array}}$$

Case 2. The computation of S succeeds, but the computation of T fails. Then the computation of the `FORALL` statement fails.

$$\frac{\texttt{sem}(\ll S, \mathcal{E}, C \gg) = \ll\, , \mathcal{E}_1, C_1 \gg, \;\; \texttt{sem}(\ll T, \mathcal{E}_1, \perp \gg) = \ll S^+, \mathcal{E}_2, \perp \gg}{\texttt{semFA}(\ll \texttt{FORALL } S \texttt{ DO } T \texttt{ END}, \mathcal{E}, C \gg) \;=\; \ll S^+, \mathcal{E}_2, \perp \gg}$$

Case 3. The computations of both S and T succeed.

Subcase 3.1. After the computation of S no stack of choice points is left.

Then the computation of the `FORALL` statement succeeds. The resulting environment is the initial environment, updated with the changes that resulted from the computation of T. These changes are computed using the function *changes* specified below. The choice points created by `T` are discarded.

$$\frac{\begin{array}{c}\texttt{sem}(\ll S, \mathcal{E}, \perp \gg) = \ll\, , \mathcal{E}_1, \perp \gg,\\ \texttt{sem}(\ll T, \mathcal{E}_1, \perp \gg) = \ll\, , \mathcal{E}_2, C_2 \gg, \;\; \texttt{changes}(\mathcal{E}_1, \mathcal{E}_2) = \mathcal{E}_3\end{array}}{\texttt{semFA}(\ll \texttt{FORALL } S \texttt{ DO } T \texttt{ END}, \mathcal{E}, \perp \gg) \;=\; \ll\, , \mathcal{E}_3 \rhd \mathcal{E}, \perp \gg}$$

Subcase 3.2. After the computation of S the stack of choice points is not empty.

Then the control returns to the last choice point, so the function *semFA* is called recursively with a statement sequence taken from the top of the stack and the environment taken from the top of the stack, updated with the changes resulting from the computation of T. The resulting environment is obtained by updating the initial environment with the changes due to the computation of T and the changes due to the recursive call of *semFA*.

$$\frac{\begin{array}{c}\texttt{sem}(\ll S,\mathcal{E},C\gg)=\ll\,,\mathcal{E}_1,CP\bowtie C_1\gg,\ \ \texttt{sem}(\ll T,\mathcal{E}_1,\perp\gg)=\ll\,,\mathcal{E}_2,C_2\gg,\\ \texttt{changes}(\mathcal{E}_1,\mathcal{E}_2)=\mathcal{E}_3\,,\\ \texttt{semFA}(\ll \texttt{FORALL}\ \texttt{fst}(CP)\ \texttt{DO}\ T\,\texttt{END},\mathcal{E}_3\rhd\texttt{snd}(CP),C_1\gg)=\ll\,,\mathcal{E}_4,\perp\gg\end{array}}{\begin{array}{l}\texttt{semFA}(\ll\texttt{FORALL}\ S\ \texttt{DO}\ T\,\texttt{END},\mathcal{E},C\gg)\ =\\ \ll\,,\texttt{changes}(\mathcal{E}_3\rhd\texttt{snd}(CP),\mathcal{E}_4)\rhd(\mathcal{E}_3\rhd\mathcal{E}),\perp\gg\end{array}}$$

If the recursive call of *semFA* fails, then the whole computation fails.

$$\frac{\begin{array}{c}\texttt{sem}(\ll S,\mathcal{E},C\gg)=\ll\,,\mathcal{E}_1,CP\bowtie C_1\gg,\ \ \texttt{sem}(\ll T,\mathcal{E}_1,\perp\gg)=\ll\,,\mathcal{E}_2,C_2\gg,\\ \texttt{changes}(\mathcal{E}_1,\mathcal{E}_2)\rhd\texttt{snd}(CP)=\mathcal{E}_3\,,\\ \texttt{semFA}(\ll \texttt{FORALL}\ \texttt{fst}(CP)\ \texttt{DO}\ T\,\texttt{END},\mathcal{E}_3\rhd\texttt{snd}(CP),C_1\gg)=\ll S^+,\mathcal{E}_4,\perp\gg\end{array}}{\texttt{semFA}(\ll\texttt{FORALL}\ S\ \texttt{DO}\ T\,\texttt{END},\mathcal{E},C\gg)\ =\ \ll S^+,\mathcal{E}_4,\perp\gg}$$

The function *changes*($\mathcal{E}_1,\mathcal{E}_2$) isolates the changes that have been made to the environment while executing the T part of the `FORALL` statement (the "permanent" changes). $\mathcal{E}_1$ is the environment before the computation of T and $\mathcal{E}_2$ is the environment after the computation of T.

The first rule isolates a variable binding the value of which has been changed in the computation of `T`.

$$\frac{\texttt{eq}(v_1,v_2)=\texttt{false}}{\begin{array}{l}\texttt{changes}([\mathcal{A}_1\ x\mapsto v_1\ \mathcal{A}_2],[\mathcal{A}_3\ x\mapsto v_2\ \mathcal{A}_4])\ =\\ [x\mapsto v_2]\rhd\texttt{changes}([\mathcal{A}_1\ \mathcal{A}_2],[\mathcal{A}_3\ \mathcal{A}_4])\end{array}}$$

Here each $\mathcal{A}_i$ is a sequence of zero or more atomic environments. The second rule isolates the binding of a new variable, introduced in the computation of `T`.

$$\frac{\texttt{def}(\mathcal{E},x)=\texttt{false}}{\texttt{changes}(\mathcal{E},[\mathcal{A}_1\ x\mapsto v\ \mathcal{A}_2])\ =\ [x\mapsto v]\rhd\texttt{changes}(\mathcal{E},[\mathcal{A}_1\ \mathcal{A}_2])}$$

If none of these rules apply, the function results in the empty environment, as specified by the default rule.

$$\texttt{changes}(\mathcal{E}_1,\mathcal{E}_2)=[]\quad\texttt{otherwise}$$

This completes the description of *semFA*. To link the *semFA* function with the *sem* function we distinguish two cases.

Case 1. The computation of the `FORALL` statement succeeds. Then we continue with the new environment and the initial stack.

$$\frac{\texttt{semFA}(\ll\texttt{FORALL}\ S\ \texttt{DO}\ T\,\texttt{END},\mathcal{E},\perp\gg)=\ll\,,\mathcal{E}_1,\perp\gg}{\texttt{sem}(\ll\texttt{FORALL}\ S\ \texttt{DO}\ T\,\texttt{END};\,S_1,\mathcal{E},C\gg)\ =\ \texttt{sem}(\ll S_1,\mathcal{E}_1,C\gg)}$$

Case 2. The computation of the `FORALL` statement fails. Then backtracking takes place.

$$\frac{\texttt{semFA}(\ll\texttt{FORALL}\ S\ \texttt{DO}\ T\,\texttt{END},\mathcal{E},\perp\gg)=\ll S^+,\mathcal{E}_1,\perp\gg}{\texttt{sem}(\ll\texttt{FORALL}\ S\ \texttt{DO}\ T\,\texttt{END};\,S_1,\mathcal{E},CP\bowtie C\gg)\ =\ \texttt{sem}(\ll\texttt{fst}(CP),\texttt{snd}(CP),C\gg)}$$

This concludes our presentation of the operational semantics of the fragment of Alma-0 introduced in Section 8.2.

8.5 Discussion

Let us summarize now the salient aspects of the operational semantics.

(1) In contrast to the customary structured operational semantics of Hennessy and Plotkin [1979] there is no rule that deals with the statements composition (";"). Instead, the *sem* function operates on a *sequence* of program statements that form the still–to–be–executed part of the program.
This choice turned out to be necessary to implement backtracking from an arbitrary position in the program text. This feature of Alma-0 was taken care of while dealing with the `ORELSE` statement. In this case the whole alternative $S_2; S_3$ to the current sequence of statements $S_1; S_3$ was pushed on the stack.

(2) To deal with backtracking the stack of choice points was introduced. It was explicitly manipulated in a number of places, namely
—in the **BES** and **COMMIT** extensions, to handle backtracking,
—in the **ORELSE** extension, to retain the remaining alternative,
—while dealing with the `IF` and `WHILE` statements, to ensure that computation continues with the original stack of the choice points,
—in the **FORALL** extension, to implement the iteration over all choice points.

(3) The auxiliary function *semFA* was introduced to deal with the most complicated case, that of the `FORALL` statement. This was needed to handle the execution of each `FORALL` statement separately with the stack of choice points initially empty.

9. IMPLEMENTATION

In this section we describe the implementation of Alma-0. The compiler consists of about 6000 lines of ANSI C, Flex (see Paxson [1995]) and Bison (see Donnoly and Stallman [1995]) code. Its detailed description can be found in Partington [1997]. (Actually, this report describes a previous version of the compiler, which has been successively improved since then). At this stage no error recovery is provided, and no optimization has been yet considered. The compiler runs on all Unix platforms.

9.1 Alma Abstract Architecture

The Alma Abstract Architecture (AAA) is the virtual architecture used during the intermediate code generation phase of the Alma-0 compiler.

The AAA combines the features of the abstract machines for imperative languages and for logic programming languages. The compiler compiles the Alma-0 programs into AAA programs. In a second phase the AAA instructions are translated into C statements.

As the Alma-0 language itself, the AAA aims to combine the best of both worlds; elements were taken from virtual machines used to compile imperative languages (in particular the RISC architecture described in Wirth [1996, pp. 55–59], and from the WAM machine used to compile a logical language (see Aït-Kaci [1991]).

Still, the AAA resembles most the virtual machines used in the compilation of imperative languages. The additions made to provide for the extensions of the Alma-0 language are

—the failure handling instructions `ONFAIL`, `FAIL`,

—the log control instructions `CREATELOG` and `REPLAYLOGS`, and

—the automatic recording of old values in the assignment instructions.

Although the current implementation of the AAA entails translating the AAA instructions into C statements, the design of the AAA is such that it should be possible to translate them into machine code. In fact, the AAA extends in a minimal way an assembly language.

9.1.1 *Backtracking: Choice Points, Failure Handling, and Log Creation.* In the AAA the notion of a *choice point* is divided into the separate notions of *failure handling* and *log creation* which, when taken together, can be used to implement a choice point.

A *failure handler* is installed by the `ONFAIL` instruction, whose execution saves the location at which execution should continue in case of a failure. When a failure is subsequently generated by the `FAIL` instruction, execution continues at this previously saved location or yields a failure (that generates an error message) if no failure handler has been installed. This failure handling notion is inspired by the exception handling mechanism in languages such as C++ (see Ellis and Stroustrup [1990]) and Java (see Gosling et al. [1996]) and is used in the Alma-0 compiler to implement the **BES** and **SBE** extensions.

When a *log* is created by the `CREATELOG` instruction, from that point on, every value that is about to be changed for the first time since the log creation is recorded in the log. When the log is played back by the `REPLAYLOGS` instruction, the recorded values are restored. The log is used in the Alma-0 compiler to implement the **ORELSE**, **SOME**, and **FORALL** extensions.

Due to the presence of the `COMMIT` statement, more than one log may have to be replayed at the same time. (For a more detailed discussion of this point see Section 9.4.) For this reason, the `REPLAYLOGS` instruction has the operand `v` that indicates the log that should become active. It replays all the logs from the current one to the one indicated by `v`.

A choice point in the sense of Section 3 is implemented by creating a new log, setting up a failure handler, and executing the first branch. When a failure occurs, the failure handler will be called, which will replay the created logs and execute the second branch.

Referring to the notion of the environment introduced in Section 8, a log can be seen as the difference between two environments. In fact, for efficiency reasons, instead of saving the current environment on the stack, in the implementation we incrementally store in the log the fragments of the old environment that are changed during the execution of the program.

9.1.2 *The Log Administration System.* More than one log may have been created at one time, but only one log is the *active log.* Each log corresponds with a created choice point. The active log corresponds with the last generated one. A log pointer register is used to refer to the active log. Consequently, when a value is to be saved, it is recorded in the active log, if there is one. The log administration system behaves as follows:

—At the beginning of the execution of an Alma-0 program there is no (active) log.

—When a log is created by the CREATELOG instruction, then the currently active log is deactivated, and the new log becomes the active log.

—When an assignment instruction that requires recording is executed, the current value of the target is saved in the active log, if any, before the assignment is performed.

—When a log is replayed by the REPLAYLOGS v instruction, the values which have been recorded in the log are restored, the active log is discarded, and the previous log is activated (if there was no previous log, there is no longer an active log). This process is repeated for all logs until the log pointer register equals to the REPLAYLOGS operand v.

As we can see, the implementation deals in the same way with the logs as the operational semantics deals with the choice points: both are treated in a stack-like fashion. However, the COMMIT and the FORALL statement break the analogy and need to be dealt with differently. In fact, the COMMIT statement requires that the logs are not modified and that the correct implementation of FORALL S DO T END is achieved by maintaining two logs that are activated in alternation depending on whether the control is within S or T.

The log mechanism corresponds to the *trail* mechanism of the WAM described in Aït-Kaci [1991]. However, comparing the two mechanisms, we see two main differences. First, in the WAM the only operations to be undone are variable instantiations, whereas in AAA they also include assignments to already instantiated variables. Second, in WAM all instantiations are undone, while in AAA not all assignments must be retracted due to the **FORALL** extension.

These differences justify the introduction of the log and its administration mechanism, which is absent in the trail. In the WAM instead of the CREATELOG instruction one single assignment instruction is used that saves the current trail pointer (register TR) on the stack. In turn, instead of the REPLAYLOGS instruction the unwind_trail operation is used that erases all the instantiations included between the current value of TR and its previous value stored in the stack.

9.1.3 AAA *Registers.* The AAA has eight registers, the most peculiar ones being the following four.

LP. the log pointer register. It contains an opaque value used by the run-time system to handle log administration; one writes to it values that have been read from it before, or lets the CREATELOG and REPLAYLOGS instructions handle this register.

BP. the failure frame pointer register that contains a pointer to the last failure frame allocated on the stack.[7] Failure frames are created by the ORELSE, SOME, and FORALL statements, as well as when a sequence of statements is used as a boolean expression. They hold the saved values of a number of registers (depending upon the statement that created the frame), and the address of the failure handler (see Section 9.1.1).

[7]We denote this register by "BP" instead of "FP" for two reasons; the abbreviation "FP" is usually reserved for the frame pointer, which is more like the AAA's EP register, and "B" is the name of the register in the WAM, that provides a similar function.

EP. the environment frame pointer register that contains a pointer to the last procedure call stack frame (comparable to the frame pointer found in actual CPU architectures). Environment frames are created when a procedure is called, and they hold the actual parameters, the saved values of a number of registers, the return address, and the local variables.

SP. the stack pointer register that points to the top of the stack. It is used to manage the allocation and deallocation of the stack frames.

The register LP is the only one which does not have a correspondence in the WAM architecture. In fact, as explained above, the log administration is not present in the WAM.

Furthermore, the usual role of the stack pointer register is played here by two different registers, namely EP and SP, which point respectively to the active frame and the last allocated frame.

These two values may be different because in the AAA architecture a stack frame is not always deallocated after the execution of its corresponding procedure is terminated. In fact, if a choice point was created in the procedure, control may, after the termination of the execution of the procedure, return to the body of the procedure. When that happens, its local variables should be accessible and should have the values they had the first time. Therefore the stack frame is not destroyed if the failure frame register is equal to or greater than the stack pointer.

This mechanism is typical of languages with choice points and automatic backtracking and is present, in the WAM, under the name of "enviroment protection." For a more detailed description of it and an example of its use see Section 4.1 of Aït-Kaci [1991].

9.2 Intermediate Code Generation

Next, we describe the details of the AAA code generation for the language constructs that deal with Alma-0 extensions.

We use a syntax–directed translation technique. Therefore each Alma-0 language construct is translated into AAA instructions, as soon as it has been recognized by the parser. The parsing strategy is bottom-up, which ensures that code has already been generated for the language constructs contained by the current construct, i.e., those language constructs that are its descendants in the abstract syntax tree. This means that the result of computational code can be used and that conditional code and its failure handling label can be correctly placed to get the correct flow of control.

The translation of the traditional language constructs is as usual, and we only discuss those translations that deal with Alma-0's extensions. For the sake of brevity, we confine ourselves to the **BES**, **SBE**, **ORELSE**, **COMMIT**, and **FORALL** extensions and that of the procedure call, which are the most interesting ones.

9.2.1 *Pseudocode.* Because the actual instruction sequences generated can be quite long, we will use pseudocode to illustrate the idea. The following language constructs are used in the pseudocode:

—`create_frame_and_save_values(<frame-type>, <registers>)` is a function the call of which creates room on the stack for the specified type of frame, and stores the values of the specified registers in the frame. The base address of the new frame is returned.

—destroy_frame(<frame-type>) is a function the call of which destroys the specified type of frame.

—<registers> := restore_values(<frame-type>, <frame- base-address>) is a statement the call of which extracts the values of the registers from the specified type of frame.

—<registers> := restore_values_and_destroy_frame(<frame-type>, <frame -base-address>) is a statement the call of which destroys the specified type of frame and returns the values of the registers.

—create_log() is a function the call of which invokes the CREATELOG instruction and returns the pointer to the newly created log.

—replay_logs(<frame-type>, <frame-base-address>) is a function the call of which invokes the REPLAYLOGS instruction with the argument equal to the value of LP stored in the indicated frame. Upon termination this value is returned.

—x := y and IF x op y THEN a ELSE b END are statements that have the obvious meaning and are translated into AAA instructions in a straightforward way.

—S.instr denotes the AAA code generated for the statement S, and S.false_lab denotes its false continuation label.

9.2.2 *The* **BES** *Extension.* When a boolean expression B is used as a statement, the following code is generated:

```
  B.instr;
  GOTO true_lab;

B.false_lab:
  FAIL;

true_lab:
```

—If the boolean expression evaluates to TRUE, execution continues normally, after the label true_lab.

—If the boolean expression evaluates to FALSE, the FAIL instruction is executed, causing a jump to the last failure point if it exists or otherwise yielding a failure (see Section 9.1.1).

This closely corresponds to the operational semantics of the **BES** extension (see Section 8.4.5).

9.2.3 *The* **SBE** *Extension.* When a statement sequence S is used as a boolean expression, the following code is generated:

```
  BP := create_frame_and_save_values(SBE_FRAME, BP, EP);
  temp := BP;
  ONFAIL fail_lab;

  S.instr;
```

```
    BP := restore_values_and_destroy_frame(SBE_FRAME, temp);
    GOTO success_lab;

  fail_lab:
    (BP, EP) := restore_values_and_destroy_frame(SBE_FRAME, BP);
    GOTO S.false_lab;

success_lab:
```

—If `S` succeeds, the execution continues normally. Because `BP` may point to a frame created during the execution of `S`, `temp` is used instead as the pointer to the original frame.

—If `S` fails, a jump is made to `S.false_lab`, the newly allocated false continuation label. Because this is the failure handler installed at the beginning, the register `BP` will now point to the correct frame.

In both cases the failure frame pointer `BP` is restored to its original value. This corresponds to the way the initial stack of choice points is used after a condition is evaluated in the operational semantics of the `IF` and `WHILE` statements (see Sections 8.4.6 and 8.4.7).

9.2.4 *The* `ORELSE` *Statement.* The statement `EITHER S1 ORELSE S2 ORELSE S3 END` is translated into

```
    BP := create_frame_and_save_values(ORELSE_FRAME, LP, BP, EP);
    LP := create_log();
    ONFAIL second_branch_lab;
    S1.instr;
    GOTO continue_lab;

  second_branch_lab:
    LP := replay_logs(ORELSE_FRAME, BP);
    EP := restore_values(ORELSE_FRAME, BP);
    LP := create_log();
    ONFAIL last_branch_lab;
    S2.instr;
    GOTO continue_lab;

  last_branch_lab:
    LP := replay_logs(ORELSE_FRAME, BP);
    (BP, EP) := restore_values_and_destroy_frame(ORELSE_FRAME, BP);
    S3.instr;

  continue_lab:
```

—A log is created, and a failure handler is installed for all but the last branch. This corresponds in the operational semantics to pushing the enviroment on the stack (see the use of the function `push` in the semantics of the `ORELSE` statement given in Section 8.4.8).

—If the execution of a branch (but not the last one) fails, the logs are replayed, and the next branch is tried. This corresponds in the operational semantics to backtracking in which the first enviroment from the stack is restored.

—If the execution of the last branch fails, no special action should be performed by the `ORELSE` statement; therefore for the last branch no log is created, and no failure handler is installed. Correspondingly, in the operational semantics no choice point is created for the last branch.

This implementation resembles the way choice points are dealt with in WAM (see Section 4.2 of Aït-Kaci [1991]), with the addition of the log administration, which is specific for the design of the AAA.

Another difference with respect to WAM is that in (pure) logic programming choice points can be created only by alternative clause selections, which are dealt with by the execution of the three instructions `try_me_else`, `retry_me_else`, and `trust_me`. In Alma-0, instead, the choice points can be created in arbitrary positions in the program, during the execution of the `ORELSE`, `SOME`, and `FORALL` constructs. Therefore each of these constructs needs to be implemented using the lower–level primitives described in the pseudocode defined above.

9.2.5 *The* `COMMIT` *Statement.* The statement `COMMIT S END` is translated into

```
savesp := SP;
savebp := BP;
S.instr;
BP := savebp;
SP := savesp;
```

—Before the execution of `S` its “context” is saved. This involves recording the values of the failure pointer register `BP` and the stack pointer register `SP`.

—After the execution of `S` the context is restored. The log pointer register `LP` is not modified for the reasons explained in Section 9.4.

9.2.6 *The* `FORALL` *Statement.* The implementation of the `FORALL S DO T END` statement uses the temporaries `savessp, savesbp`, and `saveslp`. They are the `SP`, `BP`, and `LP` that are used in `S`. When `saveslp = 0`, `S` is executing, and otherwise `T` is executing; this is used to determine whether the failure occured in `S` or in `T`. Further, `savetbp` and `savetlp` are the `BP` and `LP` that are used in `T`.

Since the translation is quite subtle, we annotate it with comments.

```
(* create frame and set up the context for T *)
BP := create_frame_and_save_values(FORALL_FRAME, LP, BP, EP);
saveslp := 0;
savetbp := BP;
savetlp := LP;
LP := create_log();
ONFAIL forall_done_lab;

(* execute S *)
S.instr;

(* save the context of S *)
savessp := SP;
savesbp := BP;
```

```
  saveslp := LP;

  (* restore the context of T *)
  LP := savetlp;
  BP := savetbp;

  (* execute T *)
  T.instr;

  (* save the context of T *)
  savetlp := LP;

  (* restore the context of S *)
  LP := saveslp;
  BP := savesbp;
  SP := savessp;

  (* continue at the next choice point in S *)
  saveslp := 0;
  FAIL;

  (* FORALL completed *)
forall_done_lab:
  EP := restore_values(FORALL_FRAME, BP);
  IF saveslp <> 0 THEN
      (* abnormal FORALL completion (failure in T) *)
      LP := replay_logs(FORALL_FRAME, BP);     (* replay logs in T *)
      LP := saveslp;                           (* set up log in S *)
      LP := replay_logs(FORALL_FRAME, BP);     (* replay logs in S *)
      BP := restore_values_and_destroy_frame(FORALL_FRAME, BP);
      FAIL;
  ELSE
      (* normal FORALL completion (failure in S) *)
      LP := replay_logs(FORALL_FRAME, BP);     (* replay logs in S *)
      BP := restore_values_and_destroy_frame(FORALL_FRAME, BP);
  END
```

—The execution alternates between two "contexts"—that of `S` and that of `T`. `T` is executed in the initial context. This ensures that the assignments in `T` are not undone when backtracking takes place in `S`. `S` is executed in its last saved context modified by the last execution of `T`.

—Each time `T` is executed the values of the `SP` and `BP` registers are not retained, so the created choice points are discarded. Upon its termination the `FAIL` instruction causes a jump to the last handler installed in `S`. When no more failure handlers are left in `S`, execution continues at `forall_done_lab`. This approach is similar to that of the failure-driven loop in Prolog (see Sterling and Shapiro [1994, p. 229]).

9.2.7 *The Procedure Call.* A procedure call in the AAA is a handled slightly differently from a procedure call in a classic virtual machine. The procedure call translates to

```
push_actual_parameters;
EP := create_frame_and_save_values(PROCCALL_FRAME, EP, SP);
JSR proc.label;
(EP, R1) := restore_values(PROCCALL_FRAME, EP);
IF R1 < BP THEN
        destroy_frame(PROCCALL_FRAME);
END;
```

where R1 is a general–purpose register to which the value of the stack pointer register is assigned and JSR is the AAA jump-to-subroutine instruction.

—If a choice point was created in the callee, execution may, at a later point, continue in the body of the procedure. When that happens, the stack frame is not destroyed if the failure frame register is equal to or greater than the stack pointer. This implements the environment protection mechanism explained in Section 9.1.3.

9.3 Implementation of the AAA

Finally, we discuss the translation of the AAA instructions into C statements. For most AAA statements such translation is straightforward. Therefore, we only explain one specific aspect of translation, namely the *log administration*.

The log administration system is an important part of the AAA, and its performance has a large impact on the overall performance of the AAA. The logs are kept in a linked list. The active log is at the front of the list, and the previously active log is its successor.

For every memory block the value of which is recorded in the log, a *log entry* is created. The log entries are kept in a binary search tree, as well as in a singly linked list. The binary search tree, which uses the address of the memory block as its key, is used in the log administration system to determine whether a memory block starting at the same address has already been recorded in this log. The linked list keeps the log entries in the order they were recorded; new log entries are added to the front of the list. Since traversing a binary tree can be computationally expensive, when the log is replayed, just the linked list is traversed front-to-back.

Because only the address of a memory block, and not its size, is used as the key for the binary search tree, one memory location is recorded in the log twice, when it is contained by two overlapping memory blocks being recorded. Fortunately, the front-to-back traversal of the singly linked list, used when replaying the log, causes its oldest value to be restored last. Therefore, the singly linked list is actually essential to the correct functioning of the log administration system.

9.4 Discussion

Of course, each of the numerous language proposals that has dealt with automatic backtracking within the imperative programming style had to address similar implementation issues. To the best of our knowledge our approach based on the AAA that combines the RISC architecture with the WAM is new though we should mention here an early proposal to design a Prolog–oriented RISC processor named Pegasus (see Seo and Yokota [1988]).

The log mechanism of the AAA allows us to refrain from saving the full environment at the time a choice point is created. Even though it is based on a simple and intuitive idea, there are some subtleties here due to the interplay of the choice points and the `COMMIT` statements. Consider the following program fragment:

```
x := a;
EITHER
  COMMIT
    EITHER
      x := b
    ORELSE
      any_statement;
    END
  END;
  FALSE
ORELSE
  y := x + c;
END
```

In this example, two choice points are set by the two nested `ORELSE` statements, and consequently two logs are created. When the control reaches the assignment `x := b` the old value of `x`, namely `a`, is stored in the active log, which is associated with the innermost `ORELSE` statement.

After the assignment to `x`, due to the `COMMIT` statement, the second choice point is erased. However, the associated log cannot be discarded at the same time because it is the only depositary of the old value for `x`. Therefore, it is kept in memory even though it pertains to a choice point which has been already erased.

When the failure occurs, upon reaching the statement `FALSE`, the two logs are replayed in succession by the `LP := replay_logs(ORELSE_FRAME, BP)` pseudocode instruction, and the correct value of `x` is restored. In the end, the program succeeds with `x` equal to `a` and `y` equal to `a+c`.

This complication does not arise in WAM because in that architecture the value can be assigned to a variable only once. Therefore the old value of a variable does not exist, and a fortiori does not need to be remembered.

10. CONCLUSIONS

In this article we presented the programming language Alma-0. In our opinion Alma-0 makes clear that many useful aspects of the logic programming paradigm, and more generally of declarative programming, can be amalgamated in a natural way with the imperative programming paradigm. Also, it shows that some algorithmic problems can be solved in a simpler way when drawing on both programming paradigms.

10.1 Related Work

A departure point for our considerations was the work of Cohen [1979], who surveys some simple primitives for nondeterministic programming within the imperative programming framework.

These primitives involve a nondeterministic choice, here adopted as an `ORELSE` statement, a parameterized nondeterministic choice, here adopted as a `SOME` statement, and the *failure* and *success* statements with the expected meaning. The *failure* and *success* statements are present in many imperative languages that support automatic backtracking, the most known of them being Icon (see Griswold and Griswold [1983]) and SETL (see Schwartz et al. [1986]).

The language Icon allows for nondeterministic constructors similar to our `ORELSE` and `SOME` statements. In order to explore the full set of branches of a nondeterministic construction the user can use the `every` statement, which resembles our `FORALL` statement. However, in Icon all the choice points created inside the body of a procedure are erased as soon as the procedure is left. To maintain choice points through procedure calls, the user must resort to the explicit `suspend` expression. Unfortunately, the *suspension* mechanism of Icon, differently from our proposal, does not have a clear counterpart in declarative semantics.

In the language SETL, nondeterminism is implemented by means of the built-in function `ok` which returns both `true` and `false` in two different branches. Therefore the Alma-0 statement `EITHER S ORELSE T END` can be implemented in SETL by `if ok then S else T end`. However in SETL, differently from Alma-0, only those variables explicitly marked as "backtracking" ones have their values restored upon backtracking. SETL also provides the `succeed` primitive which resembles the `COMMIT` statement in Alma-0. In particular, the invocation of `succeed` erases the most recent choice point left open by a previous `ok` invocation.

In Alma-0 we follow the approach taken in the 2LP language of McAloon and Tretkoff [1995] and identify boolean expressions and statements. As a result *failure* and *success* statements come for free—they are simply booleans expressions used as statements and that evaluate to `FALSE`, respectively `TRUE`. This makes the resulting programs conceptually simpler. Of all existing languages, 2LP (which stands for "logic programming and linear programming") is closest to the spirit of Alma-0. The language supports the extensions discussed in Sections 2 and 3. The `FORALL` statement is available in 2LP in a limited way by means of the `find_all` construct that corresponds to `FORALL S DO TRUE END`. This language uses C syntax and has been designed for constraint programming in the area of optimization. We shall return to it in in the next subsection.

In the realm of functional programming automatic backtracking is supported by the language MICRO-PLANNER of Sussman et al. [1970], which is an implemented fragment of its theoretical version PLANNER of Hewitt [1971]. In addition to backtracking, MICRO-PLANNER supports explicit manipulation of program states and provides some deductive and pattern–matching mechanisms. Program manipulations are dealt with by the `FRAME` command that allows the user to store the program state and with the `CONTINUE` command that restarts the execution from a stored state.

However, MICRO-PLANNER (and its successor CONNIVER of Sussman and McDermott [1972]) is a Lisp-based language, and, differently from our proposal, it lacks the full capability of imperative programming languages. In particular, it supports neither strong type checking nor powerful control structures.

On the logic programming side we would like to mention here the work that dealt with addition of arrays and bounded quantifiers (that correspond to the `FOR` and `SOME` loops) to the logic programming paradigm. Arrays in logic programming were introduced by Eriksson and Rayner [1984].

Bounded quantifiers and arrays were used in logic programming in Kluźniak and Miłkowska [1997] in which a specification language Spill was introduced that allows us to write executable, typed, specifications in the logic programming style. (The original work on this language dates from 1991.) For related references see Voronkov [1992], Barklund and Bevemyr [1993], and more recently Apt [1996].

Finally, let us mention that the initial work on the design of Alma-0 was reported in Apt and Schaerf [1997].

10.2 Towards Imperative Constraint Programming

The language Alma-0 was not a goal in itself but rather an intermediate stage on the road toward a realization of a strongly typed constraint programming language that combines the advantages of logic and imperative programming.

As already mentioned in Section 5.1, our generalized use of equality treats (some forms of) equality as a constraint. In fact, in our approach we wish to perceive constraints as boolean expressions that do not appear inside a condition. Depending on the type and syntax of their operators and operands we have then equality constraints, boolean constraints, linear integer equality constraints, linear real inequality constraints, etc.

The use of types should allow us to extend the advantages of strong typing to constraint programming: their use should lead to a simple "compartmentalization" of the constraint store and should allow us to catch simple errors at compile time and report other obvious errors at run-time. These benefits are difficult to realize within the logic programming framework.

To clarify why we feel that we remained upward compatible with the future extensions to constraint programming in the imperative programming style, let us return to the 2LP language of McAloon and Tretkoff [1995]. In 2LP there are two types of variables: the "customary," programming, variables and the *continuous* variables (the name derives from their use in mathematics). The continuous variables vary over the real interval $[0, +\infty)$ and can be either simple ones or arrays. The only way these variables can be modified is by imposing linear constraints on them. In the most extreme case these variables can be assigned a specific value by means of an equality constraint. Whenever a constraint is added, its feasibility with respect to the old constraints is tested by means of an internal simplex-based algorithm.

Even though at first sight the programming examples discussed in this article seem to have nothing to do with constraints, it turns out that many of the presented programs can be directly executed by the 2LP system (after appropriate syntactic modifications that have to do with the C-based syntax of 2LP).

The reason is that our generalized use of equality and the use of `VAR` and `MIX` parameter mechanism can be modeled in 2LP by means equality constraints and continuous variables passed as actual parameters. Consequently, our solutions to the *Remarkable Sequence Revisited* problem (Problem 7), the *Eight Queens* problem (Problem 8), and most of the multiple uses of them discussed in Section 5 can be reproduced in 2LP once the relevant arrays are declared as continuous.

It is useful to mention here that in 2LP the assignments are not "undone" upon backtracking, in contrast to the constraints imposed on continuous variables. Consequently, our solution to the *Knapsack* problem (Problem 6) cannot be reproduced within 2LP because it relies upon backtracking over assignment.

The above analysis shows that Alma-0 indeed realizes some simple uses of constraints without introducing them explicitly and seems to support our view about the upward compatibility of Alma-0 with imperative constraint programming. In fact, in Alma-0 there is no concept of a constraint store, and consequently all constraints have to be immediately processed. Currently we are working on means of incorporating a constraint store into the language. In our future work we plan to focus on the use of constraint propagation in presence of the features here introduced, a mechanism that is absent in 2LP.

We conclude by mentioning two recent alternative approaches to constraint programming that lie outside the realm of logic programming. The first is the ILOG system of Puget [1994] in which constraint programming (on finite domains) is realized in the form of a C++ class. So in ILOG constraint programming is not integrated into the underlying imperative language, C++, but rather "imported" in the form of a library.

The other is CLAIRE, a high-level functional and object-oriented language of Caseau and Laburthe [1996]. CLAIRE was designed to use constraint programming techniques to deal with operations research problems. In CLAIRE constraints are represented as objects and rule processing capabilities can be used to implement constraint propagation. CLAIRE is a complete programming system with several advanced tools available. It has been successfully used to deal with jobshop scheduling and various instances of the travelling salesman problem.

ACKNOWLEDGMENTS

We would like to thank Nissim Francez and Feliks Kluźniak for detailed comments on this article, and Ken McAloon and Carol Tretkoff for useful discussions concerning 2LP and its implementation. All five referees of Apt and Schaerf [1997], Marc Bezem, Mirka Miłkowska, and the two referees of this article provided us with useful suggestions.

This work has been partly carried out while the fourth author was visiting CWI in Amsterdam, as part of the ERCIM Fellowship Programme financed by the Commission of the European Communities.

REFERENCES

Aït-Kaci, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, Cambridge, Mass.

Apt, K. R. 1996. Arrays, bounded quantification and iteration in logic and constraint logic programming. *Sci. Comput. Program. 26,* 1-3, 133–148.

Apt, K. R. 1997. *From Logic Programming to Prolog.* Prentice-Hall, London, U.K.

Apt, K. R. and Bol, R. 1994. Logic programming and negation: A survey. *J. Logic Program. 19-20*, 9–71.

Apt, K. R. and Schaerf, A. 1997. Search and imperative programming. In *Proceedings of the 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97).* ACM Press, New York, 67–79.

Barklund, J. and Bevemyr, J. 1993. Prolog with arrays and bounded quantifications. In *Logic Programming and Automated Reasoning—Proceedings of the 4th International Conference*, A. Voronkov, Ed. Lecture Notes in Computer Science, vol. 698. Springer-Verlag, Berlin, 28–39.

BARR, A., FEIGENBAUM, E. A., AND COHEN, P. R. 1981. *The Handbook of Artificial Intelligence* Vol. 1. HeurisTech, Stanford.

BRUNEKREEF, J. 1998. Annotated algebraic specification of the syntax and semantics of the programming language Alma-0. Tech. Rep. P9803, Programming Research Group, University of Amsterdam, The Netherlands. Available online at http://www.cwi.nl/alma.

BYLANDER, T. 1991. Complexity results for planning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-91)*. 274–279.

CASEAU, Y. AND LABURTHE, F. 1996. Introduction to the CLAIRE programming language. Tech. rep., Departement Mathématiques et Informatique, Ecole Normale Supérieure, Paris, France.

COELHO, H. AND COTTA, J. C. 1988. *Prolog by Example.* Springer-Verlag, Berlin.

COHEN, J. 1979. Non-Deterministic algorithms. *ACM Comput. Surv. 11,* 2, 79–94.

COLMERAUER, A. 1990. An introduction to Prolog III. *Commun. ACM 33,* 7, 69–90.

DONNOLY, C. AND STALLMAN, R. 1995. *Bison, the YACC-compatible Parser Generator.* Free Software Foundation, Cambridge, Mass. Available online at http://www.math.utah.edu/docs/info/bison_toc.html.

ELLIS, M. E. AND STROUSTRUP, B. 1990. *The Annotated C++ Reference Manual.* Addison Wesley, Reading, Mass.

ERIKSSON, L.-H. AND RAYNER, M. 1984. Incorporating mutable arrays into logic programming. In *Proceedings of the 2nd International Conference on Logic Programming*, S. Å. Tarnlund, Ed. Uppsala University, 101–114.

FIKES, R. E. AND NILSSON, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell. J. 2*, 189–208.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification, Version 1.0.* Sun Microsystems. Available online at http://java.sun.com/docs/language_specification/index.html.

GRISWOLD, R. E. AND GRISWOLD, M. T. 1983. *The Icon Programming Language.* Prentice-Hall, Englewood Cliffs, N. J.

GROENENDIJK, J. AND STOKHOF, M. 1991. Dynamic predicate logic. *Ling. Phil. 14,* 2, 39–101.

HENNESSY, M. C. B. AND PLOTKIN, G. D. 1979. Full abstraction for a simple programming language. In *Proceedings of Mathematical Foundations of Computer Science.* Lecture Notes in Computer Science, vol. 74, Springer-Verlag, New York, 108–120.

HEWITT, C. 1971. Procedural embedding of knowledge in PLANNER. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence (IJCAI-71).* 167–182.

KLINT, P. 1993. A meta–environment for generating programming environments. *ACM Trans. Softw. Eng. Methodol. 2,* 2, 176–201.

KLUŹNIAK, F. AND MIŁKOWSKA, M. 1997. Spill: A logic language for writing testable requirements specifications. *Sci. Comput. Program. 28,* 2 & 3, 193–223.

MARCUS, L. 1996. Syntactic and semantic dependence of array-arithmetic sentences, with an application to program verification. *Fundamenta Informaticae 27,* 1, 77–100.

MCALOON, K. AND TRETKOFF, C. 1995. 2LP: Linear programming and logic programming. In *Principles and Practice of Constraint Programming*, P. Van Hentenryck and V. Saraswat, Eds. MIT Press, Cambridge, Mass. 101–116.

PARTINGTON, V. 1997. Implementation of an imperative programming language with backtracking. Tech. Rep. P9712, Department of Mathematics, Computer Science, Physics & Astronomy, University of Amsterdam, The Netherlands. Available online at http://www.wins.uva.nl/research/prog/reports/reports.html.

PAXSON, V. 1995. *Flex, version 2.5, A fast scanner generator.* The Regents of the University of California. Available online at http://www.math.utah.edu/docs/info/flex_toc.html.

PUGET, J.-F. 1994. A C++ implementation of CLP. In *Proceedings of the 2nd Singapore International Conference on Intelligent Systems.* Singapore.

SCHWARTZ, J. T., DEWAR, R. B. K., DUBINSKY, E., AND SCHONBERG, E. 1986. *Programming with Sets—An Introduction to SETL.* Springer-Verlag, New York.

SCOTT, D. S. AND DE BAKKER, J. W. 1969. A theory of programs. Unpublished seminar notes, IBM, Vienna.

SEO, K. AND YOKOTA, T. 1988. Pegasus: A RISC processor for high-performance execution of Prolog programs. In *VLSI '87*, C. H. Sequin, Ed. IFIP, North-Holland, Amsterdam, 261–274.

SHOHAM, Y. 1994. *Artificial Intelligence Techniques in Prolog.* Morgan Kaufmann, San Francisco, Calif.

STERLING, L. AND SHAPIRO, E. 1994. *The Art of Prolog*, 2nd ed. The MIT Press, Cambridge, Mass.

SUSSMAN, G. J. AND MCDERMOTT, D. V. 1972. CONNIVER reference manual. AI Memo no. 259, MIT Project MAC.

SUSSMAN, G. J., WINOGRAD, T., AND CHARNIAK, E. 1970. MICRO-PLANNER reference manual. AI Memo no. 203, MIT Project MAC.

VAN DEURSEN, A., HEERING, J., AND KLINT, P., Eds. 1996. *Language Prototyping—An Algebraic Specification Approach.* AMAST Series in Computing, vol. 5. World Scientific Publishing Co, Singapore.

VORONKOV, A. 1992. Logic programming with bounded quantifiers. In *Logic Programming and Automated Reasoning—Proceedings of the 2nd Russian Conference on Logic Programming*, A. Voronkov, Ed. Lecture Notes in Computer Science, vol. 592. Springer-Verlag, Berlin, 486–514.

WIRTH, N. 1985. *Programming in Modula-2*, 3rd ed. Springer-Verlag, New York.

WIRTH, N. 1986. *Algorithms and Data Structures.* Prentice-Hall, Englewood Cliffs, N. J.

WIRTH, N. 1996. *Compiler Construction.* Addison Wesley, Reading, Mass.

Received September 1997; revised March 1998; accepted May 1998