Search and Imperative Programming

Krzysztof R. Apt
apt@cwi.nl
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
and
Dept. of Mathematics, Computer Science, Physics & Astronomy
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands

Andrea Schaerf
aschaerf@dis.uniroma1.it
Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza"
via Salaria 113, 00198 Roma, Italy

Abstract

We augment the expressive power of imperative programming in order to make it a more attractive vehicle for problems that involve search. The proposed additions are limited yet powerful and are inspired by the logic programming paradigm. We illustrate their use by presenting solutions to a number of classical problems, including the straight search problem, the knapsack problem, and the 8 queens problem. These solutions are substantially simpler than their counterparts written in the conventional way and can be used for different purposes without any modification.

The proposed language is an intermediate stage on the road towards a realization of a strongly typed constraint programming language that combines the advantages of the logic programming and imperative programming.

1 Introduction

1.1 Motivation

In this paper we try to combine advantages of logic and imperative programming in order to deal in a natural way with algorithmic problems that involve search. To this end we extend imperative programming with some features that are inspired by the logic programming paradigm. They involve:

- 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. In particular, we can use a rich variety of data types, including arrays and records, *in presence* of strong type checking.

The assignment, shunned in declarative programming and a fortiori 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 the solutions to 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 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.

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 our language proposal. 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 our language proposal.

Combining two programming styles is always a debatable endeavour 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, without assignment can be viewed as declarative in the sense that they admit an alternative reading as logic formulae. Verification of such programs or program fragments is considerably simplified due to their logical meaning. In some cases programs are equal to their specifications — see e.g. our solutions to Problems 3 and 10, — and are therefore obviously correct.

- All the introduced programming constructs guarantee termination. As a result we can now write programs, like the solutions to the just mentioned two problems or solutions to Problems 8 and 11, 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 the imperative programs.

As we often refer here to programs presented in Wirth (1986), a book about programming in Modula-2, we have used below the syntax of Modula-2. More precisely, as a base language we take a subset of Modula-2 in which, after carrying out the proposed extensions, the example programs can be written.

The 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 "`;`" are already legal statements, the presence of which is ignored.

It should be stressed, however, that the notation is completely inessential in our investigations. The presented programs 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.

1.2 Related Work

A departure point for our considerations is 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 `OR` 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 backtracking, the most known of them being Icon (see Griswold & Griswold (1983)) and SETL (see Schwartz, Dewar, Dubinsky & Schonberg (1986)).

In our language proposal we follow the approach taken in the 2LP language of McAloon & 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 is the closest to the spirit of our proposal. This language uses C syntax and has been designed for constraint programming in the area of optimization. 2LP stands for "logic programming and linear programming".

The features that are present in our proposal and which we believe are new are: The `FORALL` statement, that offers a controlled iteration over backtracking, equality used as an assignment, and a new parameter mechanism that combines call by value and call by reference.

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 & Rayner (1984).

Bounded quantifiers and arrays were introduced in logic programming by Kluźniak (1993) in a specification language SPILL-2 in which executable specifications can be written in the logic programming style. For related references see Voronkov (1992) and Barklund & Bevemyr (1993).

Conceptually, we arrived at our language proposal by encountering difficulties in finding satisfactory solutions to various problems here considered, like the *knapsack problem*, in the logic programming framework of Apt (1996).

In our exposition we proceed in stages and introduce each extension separately.

2 Expressions and Statements

2.1 Boolean Expressions as Statements

We begin by allowing boolean expressions to be used as statements. In what follows we refer to boolean expressions used as statements as *tests*. A specific interpretation of tests is crucial for our purposes. We stipulate the following.

Definition 1

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

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

(iii) If during evaluation of a test an uninstantiated variable is encountered, then a run-time error arises.

(iv) If the computation of a procedure call fails, then the computation upon reaching this procedure call *fails*.

(v) If the computation of a function call fails, then a run-time error arises.

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

Clause (iii) refers to the notion of an uninitialized variable, further elaborated in Section 5.1. We shall also relax there this clause for tests of the form `s = t`.

Clauses (iv) and (v) explain how the failure propagates due to the use of functions and procedures. We stress the fact that failure differs from a run-time error.

For example, consider the following program fragment

```
x < 10;
y := 2*x + 1
```

If the value of `x` is `6` the program succeeds and `y` is `13`; conversely, if the value of `x` is `15` the program fails and no value is given to `y`.

The above extension is hardly of interest in isolation.

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.

As a first example of the usefulness of this extension consider the following problem.

Problem 1 Check whether an array `a: ARRAY[1..M] OF INTEGER` is ordered.

The solution is immediate — it suffices to write the following statement:

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

Note that when the array is not ordered, the above statement evaluates to FALSE as soon as the least value of `i` is encountered for which the test `a[i] <= a[i+1]` fails.

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. Consequently, we can now use the above statement as a boolean expression, as in the following program fragment:

```
WHILE NOT FOR i:=1 TO M-1 DO a[i] <= a[i+1] END
DO swap(a[i], a[i+1]) END
```

which implements a naive sorting algorithm.

Problem 2 Count the number of different elements in an array `x: ARRAY[1..M] OF CHAR`.

A natural solution to this problem (although not the most efficient one) uses a statement as a boolean expression and assignment:

```
no := 0;
FOR i := 1 TO M DO
  IF FOR j := 1 TO i-1 DO x[i] <> x[j] END
  THEN no := no+1
  END
END
```

The outcome is computed in the variable `no`.

3 Nondeterministic Statements

We now proceed by introducing choice points and backtracking into the computational process.

3.1 OR Statement

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

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

We refer to the parts of the `OR` statement as *branches*. The computational interpretation is as follows.

Definition 3 The computation of an `OR` statement starts by proceeding through the first branch. If the computation eventually fails, possibly beyond the end of the `OR` statement, *backtracking* takes place and the computation resumes with the next branch in the state in which the previous branch was entered. If the last branch fails the `OR` statement fails. □

Thus the `OR` statement introduces choice points to which the computation can return.

Consider the following program fragment

```
EITHER x := x - 20; x > 0
ORELSE x > 10; y := x
END
```

If the value of `x` is 15 the computation that passes through the first branch fails upon encounter of the test `x > 0`, then backtracking takes place, the value `15` for `x` is restored and the computation eventually succeeds, assigning the value `15` to `y`. Conversely, if the value of `x` is 5, the second branch fails as well and no value is assigned to `y`.

Consider now this other example, assuming the value of `x` is `-1`:

```
EITHER y := x
ORELSE x < 0; y := -x
END
x := x + 10;
y > 5;
```

Here again the computation that passes through the first branch fails upon encounter of the test `y > 5` and backtracking takes place. The second branch of the `OR` statement is entered restoring the value of `x` equal to `-1` and eventually the whole computation fails, with `x` equal to `9` and `y` equal to `1`.

Note that in the second example the failure occurs outside the scope of the `OR` statement;[1] that is, the backtracking takes place here *after* the control has left the `OR` statement. The example shows that upon backtracking the assignments outside the scope of the `OR` statement are also "undone". This interpretation of the meaning of the `OR` statement is crucial to our purposes.

3.2 SOME Statement

Next, we introduce a `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. 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 the empty statement (`skip`).
- `m2 = m1`. Then `S` is equivalent to `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 a `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]

As a simple example consider the following problem that illustrates use of the `SOME-FOR` combination.

[1] This point will be reconsidered in Section 4.

[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.

Problem 3 *Straight string search.* Consider two arrays of characters, `p` (the *pattern*) and `s` (the *string*), declared as

```
p: ARRAY [0..M-1] OF CHAR;
s: ARRAY [0..N-1] OF CHAR;
```

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

The following program is a naive solution to this problem. It is much more straightforward than the customary solution given in Wirth (1986, page 60).

```
SOME i:= 0 TO N-M DO
  FOR j:= 0 TO M-1 DO
    s[i+j] = p[j]
  END
END
```

The result is delivered here in the variable `i`.

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

Problem 4 (See Coelho & Cotta (1988, page 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.

```
TYPE Sequence = ARRAY[1..27] OF [1..9];
PROCEDURE question(VAR a: Sequence);
  VAR i,j: CARDINAL;
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 question
```

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 & Nilsson (1971)).

In propositional STRIPS, *actions* and *goals* are members of two (disjoint) alphabets of propositional letters. A STRIPS *action rule* is composed by 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 whose application 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 (non-deterministically) 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 on a. The concatenation of the sequences $p \circ \langle a \rangle$ for all g in the final state gives the complete plan.

The STRIPS algorithm involves guessing (realized by backtracking) and consequently it is natural to implement it in Prolog. A Prolog implementation of the STRIPS algorithm is provided 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 `OR` 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 assume the availability of a type `List` (whose elements are characters), with various predefined functions (with their usual intuitive meaning): `member`, `head`, `tail`, `subset`, `union`, `difference`, `insert`. We also assume that `head` and `tail` fail if the argument is an empty list.

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

PROCEDURE Strips(VAR State: List; Goals: List;
                 ForbActions: List;
                 VAR Plan: List; Lib: ActionLib);
VAR Goal: CHAR;
```

```
BEGIN
  IF NOT subset(Goals,State)
  THEN
    ChooseGoal(Goal,Goals,State);
    AchieveGoal(Goal,Lib,ForbActions,State,Plan);
    Strips(State,Goals,ForbActions,Plan,Lib)
  END
END Strips;

PROCEDURE ChooseGoal(VAR Goal: CHAR;
                     Goals: List; State: List);
BEGIN
  EITHER Goal := Head(Goals); NOT member(Goal,State)
  ORELSE ChooseGoal(Goal, Tail(Goals))
  END
END ChooseGoal;

PROCEDURE AchieveGoal(Goal: CHAR; Lib: ActionLib;
                      ForbActions: List;
                      VAR State: List; VAR Plan: List);
VAR i: CARDINAL;
BEGIN
  SOME i := 1 TO NumActions DO
    NOT member(Lib[i].Name, ForbActions);
    member(Goal,Lib[i].AddList);
    Strips(State,Lib[i].PreList,
           insert(Lib[i].Name,ForbActions),Plan,Lib);
    ApplyRule(Lib[i],State,Plan)
  END
END AchieveGoal;

PROCEDURE ApplyRule(Action: ActionType;
                    VAR State: List; VAR Plan: List);
BEGIN
  State := union(difference(State,Action.DelList)),
                 Action.AddList);
  Plan := insert(Action.Name,Plan)
END ApplyRule;
```

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 `ForbActions` and for `Plan`, and the given action library (which is not modified) as `Lib`.

Notice that the guess of the goal, which in Prolog is typically obtained by the call `member(Goal,Goals)` with `Goals` instantiated and `Goal` a variable, is implemented here by means of the `OR` statement.

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 would be 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 behaviour that could result from existence of active choice points far away in the program.

To this aim, we introduce a `COMMIT` statement, with the following syntax:

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

The statement `COMMIT S END` is executed in the same way as `S`, except that when the computation of `S` ends successfully, all choice points created by the execution of `S` are removed. The choice points previously created are left unchanged.

For example, the following program fragment

```
COMMIT
  EITHER x > 6; y := x
  ORELSE y := 6
  END;
  y < 10;
END
y < 8;
```

fails if the value of `x` is `9`: When the control leaves the `COMMIT` statement the value of `y` is `9` and the choice point created by the `OR` statement is erased. Therefore, backtracking to the second branch does not takes place once the test `y < 8` fails. On the other hand, if the value of `x` is `11`, the whole computation succeeds with value `6` for `y`: The test `y < 10` (i.e. the one inside `COMMIT`) fails, and the second choice is performed.

We now show an example of the use of the `COMMIT` statement by presenting a naive solution to a classical problem.

Problem 6 Check whether a propositional formula is satisfiable.

We implement an enumeration procedure that uses the `OR` statement to assign values to an array of propositional letters. We assume to have a representation of the formula, by means of the type `Formula`, and a function `SatisfyFormula` that checks if a given interpretation is a model of the formula. Then the following program fragment succeeds (and certifies an interpretation as a model) if and only if the formula is satisfiable.

```
VAR a: ARRAY [1..N] OF BOOLEAN;
  f: Formula;
...
COMMIT
  FOR i := 1 TO N DO
    EITHER a[i] := TRUE
    ORELSE a[i] := FALSE
    END
  END;
  SatisfyFormula(a,f)
END
```

The COMMIT statement prevents the program from looking for a different model in case a later failure occurs (we want to check the satisfiability of the formula, and not to generate all its models upon backtracking).

4.2 FORALL Statement

Suppose now that we want to compute not just one model, but *all* the models of a propositional formula. In this case we need to explore the whole search space, and not only the part of it up to the first successful node.

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

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

The statement FORALL S DO T END is processed in the following way: S and T are executed in sequence, thereafter, if there is a choice point left within S, control returns to the successive branch of the choice (as if a failure were encountered). This process continues as long as there are still choice points in S. Thereafter, the computation succeeds, even if S fails (i.e. S succeeds 0 times), and no choice points are created.

Consequently, FORALL COMMIT S END DO T END is not equivalent to S; T as the latter statement fails if S does.

Statements within S are undone upon backtracking, whereas those in T are not, i.e. they have a *permanent* effect within the execution of the FORALL statement. The choice points created during each execution of T are removed as soon as the control returns to the successive choice point left within S. So, in effect, there is an implicit COMMIT statement surrounding T. If at certain stage the execution of T fails, then the execution of the whole statement FORALL S DO T END fails. For example, the program fragment

```
y := 0;
FORALL
  EITHER x := 5
  ORELSE x := 2
  ORELSE x := 7
  END
DO
  write(x);
  y := y + x
END;
```

prints the values 5, 2, and 7, and assigns the value 14 to y. The computation succeeds and leaves no choice points after its execution.

Note that the effect of T is permanent only within the execution of the FORALL statement, whereas it will be undone if it is included in another nondeterministic statement. For example, if a FORALL statement is inside a branch of an OR statement, the state of the variables before entering a new branch is restored, thus removing the effects of the DO part of the FORALL statement.

Although we do not impose any syntactic constraint on the form of the FORALL statement, its correct use imposes some common-sense limitations. Namely, no variable can 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 non-permanent, 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.

Problem 7 Compute all models of a propositional formula and return them in a list.

The following program fragment does the job:

```
VAR a: ARRAY [1..N] OF BOOLEAN;
  f: Formula;
  m: ListOfModels;
...
m := EmptyList;
FORALL
  FOR i := 1 TO N DO
    EITHER  a[i] := TRUE
    ORELSE a[i] := FALSE
    END
  END;
  SatisfyFormula(a,f);
DO
  m := insert(a,m);
END
```

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 (see e.g. Sterling & Shapiro (1986)) 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. This allows us to include in T all the permanent operations that need to be completed after each solution. Such operations always exist (otherwise everything that was computed would be lost) and in logic programming they are generally implemented by means of input/output operations or assert and retract.

Problem 8 *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 (i) it violates the capacity constraint or (ii) it can't 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` cycle with an `OR` statement inside.

Calling `volume` the total volume of the objects for which we have set x_i to 1, condition (i) can be tested 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, condition (ii) can be tested by checking if the`waste` in the given partial solution is larger than the waste in the current (complete) best solution. Therefore, the use of tests allows us to enforce conditions (i) and (ii) in a very simple way by means of the statements `volume <= capacity` and `waste < TotalValue - CurrentBest`.

```
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 CurrentBest, TotalValue, volume, waste: REAL;
BEGIN
  CurrentBest := 0;
  TotalValue := 0;
  FOR i := 1 TO N DO
    TotalValue := TotalValue + Value[i]
  END;
  volume := 0;
  waste := 0;
  FORALL
    FOR i := 1 TO N DO
      EITHER
        Solution[i] := 1;
        volume := volume + Volume[i];
        volume <= capacity
      ORELSE
        Solution[i] := 0;
        waste := waste + Value[i];
        waste < TotalValue - CurrentBest
      END
    END
  DO CurrentBest := TotalValue - waste
  END
END knapsack
```

The assignment to the variable `CurrentBest` 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 Problem 4. Suppose that we would like now to find an array of 27 elements that is a remarkable sequence. To obtain a single solution to both problems we proceed in two steps. As a first step 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 crucial definition.

Definition 4 Consider a test `s = t`.

- Suppose both sides are expressions with known values. Then we treat it as in Definition 1.
- Suppose that
  - one side, say `s`, is an uninitialized variable,
  - the other side, `t`, is an expression with known value,
  - their types are compatible.

Then we treat it as an *assignment*, which means that the value of `t` is assigned to `s`.

- The remaining cases yield a run-time error. □

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

We can now return to the issue raised at the beginning of this subsection. Thanks to the generalized use of equality the original program is now a solution to both problems!

From the computational point of view the equalities in the `question` procedure serve now both to assign a value to an (uninitialized) subscripted variable and to test a value of an (initialized) subscripted variable. So if the actual array parameter is completely uninitialized, the equalities are used as assignments, and if the actual array parameter is completely initialized, these equalities are used as tests.

5.2 New Parameter Mechanism

In this subsection we take a closer look at the interplay between the generalized use of equality and the parameter-passing mechanisms. We just noticed that the procedure `question` could be used either for testing or for computing. To this end it was crucial that its parameter was declared as a call by reference parameter.

In general, this double use of a single procedure is not possible. For example, in the case of simple types (say `INTEGER`) also non-variable expressions (like 7) can be passed as actuals. In this case only call by value is legal.

We now introduce a parameter-passing mechanism that permits such a double use of procedures — for testing and for computing — for a larger class of programs. We call this parameter mechanism *call by mixed form* and denote its use by the keyword `MIX`. First, we introduce it for parameters of a simple type.

Definition 5 Suppose that the formal `MIX` parameter is a variable of simple type.

- If the actual parameter is an *uninitialized variable*, then `MIX` becomes call by reference.
- If the actual parameter is an *expression with known value*, then `MIX` becomes call by value.
- The remaining cases yield a run-time error. □

So the call by mixed form is in effect a "late binding" parameter mechanism — the decision whether a specific parameter is to be called by reference or by value is delayed to the run-time.

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

Problem 9 Check if an integer `e` is present in an array `a: ARRAY[0..N] OF INTEGER`.

We write the solution as a procedure.

```
PROCEDURE find(MIX e: INTEGER; a: ARRAY OF INTEGER);
  VAR i: CARDINAL;
BEGIN
  SOME i := 0 TO HIGH(a) DO e = a[i]
  END
END find
```

To allow the use of this procedure for arrays of different sizes, we declared here the array parameter as an open array parameter (see Wirth (1985, page 53)). Recall, that if the actual parameter `b`, declared as

```
b: ARRAY[m..n] of INTEGER
```

is used, then `a[i]` denotes `b[m+i]` for `i` $\in$ `[0..HIGH(a)]`, where `HIGH(a) = n-m`.

Suppose now that `x` is an uninitialized integer variable and `a` and `b` are arrays of integers. 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 then it suceeds and otherwise it fails;
- the program fragment

  ```
  FORALL
    find(x,a);
    find(x,b)
  DO
    write(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 reference and the second by value.

So, thanks to the fact that we declared the first parameter as a `MIX` parameter, 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 behaviour 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`. Finally, if we used call by reference, the program fragment `find(x,a); find(x,b)` would exhibit the same behaviour as for call by mixed form, but the call `find(7,a)` would yield a compile time error.

To complete the presentation, the call by mixed form is extended to parameters of compound types: it is determined per position whether it is to be called by value or by reference.

As an example consider the following simple solution to the eight queens problem.

Problem 10 *The Eight Queens Problem.* Place 8 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. The program is purely declarative in the sense that it can be dually read as a logic formula.

```
CONST N = 8;
TYPE board = ARRAY[1..N] OF [1..N];
PROCEDURE queens(MIX 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 `MIX` parameter and the assignments to its elements take place by means of equalities. As a result 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 become interpreted as tests.

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

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

So the procedure `queens` can be used to compute, to test and 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 need to be changed. This is in contrast to the customary solution (see e.g. Wirth (1986, pages 153-157)) which in each case has to be modified.

5.3 Testing the Status of an Expression

The additions discussed in the preceding two subsections relied in a crucial way on the distinction between initialized and uninitialized variables, and more generally between expressions with known and with unknown value. In this subsection we go one step further and add to the language a relation that allows us to perform an explicit test whether an expression has a known value.

More specifically, we introduce a unary relation `KNOWN` such that for an expression `s` the test `KNOWN(s)` succeeds if and only if `s` is an expression with a known value. In particular, for a variable `x` the test `KNOWN(x)` succeeds if `x` is initialized and fails if `x` is uninitialized.

To illustrate a natural use of the `KNOWN` relation consider the following variant of a problem from Colmerauer (1990).

Problem 11 *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 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.

Passing `PosX` and `PosY` as `MIX` parameters (instead of `VAR`) allows us also to use the program to check a given solution or to complete a partial solution.

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

PROCEDURE Squares(Sizes: SquaresVector,
                  MIX PosX, PosY: SquaresVector);
  VAR i,j,k: INTEGER;
BEGIN
  COMMIT
    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
END Squares;

PROCEDURE AlreadyCovered(i,j: INTEGER;
                         Sizes: SquaresVector;
                         MIX PosX, PosY: SquaresVector);
  VAR h: INTEGER;
BEGIN
  SOME h := 1 TO M DO
    KNOWN(PosX[h]);
    KNOWN(PosY[h]);
    PosX[h] <= i;
    PosX[h] + Sizes[h] > i;
    PosY[h] <= j;
    PosY[h] + Sizes[h] > j
  END
END AlreadyCovered;
```

Note that this program does not use any assignment.

6 Conclusions and Future Work

In this paper we extended the imperative programming paradigm to deal in a more natural way with the algorithmic problems involving search. The realized extension was not a goal in itself but rather an intermediate stage on the road towards a realization of a strongly typed constraint programming language that combines logic and imperative programming.

In our approach primitive constraints are simply primary boolean expressions. Depending on the type and syntax of their operands we have boolean constraints, linear integer constraints, linear real 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.

In such a constraint programming language some problems can be solved by a program that consists of two parts: the generation of the relevant constraints and the constraint solving part. For instance, in the case of the 8 queens problem the generation part literally coincides with the problem description in the Modula-2 syntax. The constraint solving part depends on the syntax and the type of the constraints. The presence of types ensures security and the correct usage of constraints, while the `MIX` parameter passing mechanism provides a flexible use of the procedures. In the case of programs involving chronological backtracking, like the solutions to the last two problems, the use of constraints should lead to more efficient solutions due to the constraint propagation.

This view of constraint programming is related to though conceptually different from Puget (1994) in which constraint programming (on finite domains) is realized in the form of a C++ class. It is much closer related to 2LP of McAloon & 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 w.r.t. the old constraints is tested by means of an internal simplex-based algorithm.

The language supports the extensions discussed in Sections 2 and 3. Moreover, the `FORALL` statement is available in a limited way by means of the `find_all` construct that corresponds to `FORALL S DO skip END`.

Even though at first sight the programming examples here discussed have nothing to do with constraints, it turns out that most 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). In particular, in absence of assignment, the `MIX` parameter mechanism models *exactly* the computational behaviour of continuous variables passed as actual parameters. As a result, our solutions to Problems 4 and 10 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. This seems to support our view that call by mixed form is a natural parameter mechanism.

In 2LP the assignments are not "undone" upon backtracking, in contrast to the constraints imposed on continuous variables. Consequently, our solution to Problem 8 (the *knapsack problem*) cannot be reproduced within 2LP because it relies upon backtracking over assignment.

The above analysis shows that 2LP supports an alternative style to programming for problems involving search and that our language proposal realizes some simple uses of constraints without introducing them explicitly. In our future work we plan to focus on the use of constraint propagation in presence of the features here introduced, and on the use of constraints as program output, mechanisms that are absent in 2LP.

We think that the intermediate language proposal here discussed is of interest in its own right. First, it makes clear that many useful aspects of the logic programming paradigm can be realized within the imperative programming paradigm. Second, it shows that some algorithmic problems can be solved in a more natural way when drawing on both programming paradigms. We are enhanced in this view by the fact that we have written several other classical programs involving search in this language, like the α-β pruning. They will be appear in the long version of this paper.

So far we left out of consideration three important topics regarding this language proposal: semantics, program verification and implementation.

First, it is worthwhile to mention that most of the constructs of our language admit a declarative interpretation. So "`;`" corresponds to the conjunction, the `OR` statement to the disjunction, the `FOR` statement to the bounded universal quantification and the `SOME` statement to the bounded existential quantification (though the scope of both bounded quantifiers extends to the end of a conjunction). Finally, `FORALL S DO T` corresponds to the restricted quantification: $\forall \mathbf{x}(\phi_{\mathtt{S}} \rightarrow \phi_{\mathtt{T}})$, where $\mathbf{x}$ is the list of all variables of `S` and $\phi_{\mathtt{U}}$ is the declarative interpretation of the statement `U`.

Because of the presence of assignment not all programs admit a declarative interpretation. Therefore we are working now on an operation semantics of the proposed language in the style of Hennessy & Plotkin (1979).

Program verification calls for proof rules in the style of Hoare or for the weakest precondition semantics in the style of Dijkstra. In our case we plan to extend the weakest precondition semantics provided by Bonsangue & Kok (1994) for a simple language involving both don't know and don't care nondeterminism in the form of guarded commands and backtracking, to the primitives of our language.

As to the implementation, which is still in a preliminary phase, there are essentially two options: either to translate the programs into deterministic programs (here Modula-2 programs) — an approach already discussed in Cohen (1979), or to compile them into a WAM like abstract machine — an approach followed by McAloon & Tretkoff (1995).

The features defined in Section 5 (e.g. generalized equality) require some special machinery. In particular, to account for the notion of known value it is necessary for each type to put aside a specific bit pattern, which is assigned to all uninitialized variable (resembling the `nil` value for pointers). Alternatively, we might associate with each variable (or variable field) a single bit which tells whether the variable is initialized (has a known value) or not (its value is unknown).

The `MIX` parameter passing mechanism could be dealt with by always storing the address of the actual `MIX` parameter as in call by reference, even though it could be also passed by value. This way we reserve space for values of expressions that are `MIX` parameters in the stack frame of the caller rather than in the stack frame of the callee. The parameter would have then to be accessed indirectly even when it is, effectively, a parameter passed by value.

Acknowledgements

We would like to thank Feliks Kluźniak for detailed comments on a preliminary version of the paper and Ken McAloon and Carol Tretkoff for useful discussions concerning 2LP. All five referees provided us with useful suggestions.

This work has been partly carried out while the second author was visiting CWI in Amsterdam. It is part of the ERCIM fellowship Programme and financed by the Commission of the European Communities.

References

Apt, K. (1996), 'Arrays, bounded quantification and iteration in logic and constraint logic programming', *Science of Computer Programming* **26**(1-3), 133–148.

Barklund, J. & Bevemyr, J. (1993), Prolog with arrays and bounded quantifications, *in* A. Voronkov, ed., 'Logic Programming and Automated Reasoning—Proc. 4th Intl. Conf.', LNCS 698, Springer-Verlag, Berlin, pp. 28–39.

Bonsangue, M. M. & Kok, J. (1994), 'The weakest precondition calculus: recursion and duality', *Formal Aspects of Computing* **6**(6A), 788–800.

Bylander, T. (1991), Complexity results for planning, *in* 'Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)', pp. 274–279.

Coelho, H. & Cotta, J. C. (1988), *Prolog by Example*, Springer-Verlag, Berlin.

Cohen, J. (1979), 'Non-Deterministic algorithms', *ACM Computing Surveys* **11**(2), 79–94.

Colmerauer, A. (1990), 'An introduction to Prolog III', *Communications of ACM* **33**(7), 69–90.

Eriksson, L.-H. & Rayner, M. (1984), Incorporating mutable arrays into logic programming, *in* S. Å. Tarnlund, ed., 'Proc. Second Int'l Conf. on Logic Programming', Uppsala University, pp. 101–114.

Fikes, R. E. & Nilsson, N. J. (1971), 'STRIPS: A new approach to the application of theorem proving to problem solving', *Artificial Intelligence Journal* **2**, 189–208.

Griswold, R. E. & Griswold, M. T. (1983), *The Icon Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, USA.

Hennessy, M. & Plotkin, G. (1979), Full abstraction for a simple programming language, *in* 'Proceedings of Mathematical Foundations of Computer Science', Lecture Notes in Computer Science 74, Springer-Verlag, New York, pp. 108–120.

Kluźniak, F. (1993), SPILL-2: the language, Technical report ZMI Reports No 93-03, Institute of Informatics, Warsaw University. A deliverable for year 1 of the BRA Esprit Project Compulog 2.

McAloon, K. & Tretkoff, C. (1995), 2LP: Linear programming and logic programming, *in* P. V. Hentenryck & V. Saraswat, eds, 'Principles and Practice of Constraint Programming', MIT Press, pp. 101–116.

Puget, J.-F. (1994), A C++ implementation of CLP, *in* 'Proceedings of the Second Singapore International Conference on Intelligent Systems', Singapore.

Schwartz, J. T., Dewar, R. B. K., Dubinsky, E. & Schonberg, E. (1986), *Programming with Sets — An Introduction to SETL*, Springer-Verlag, New York.

Shoham, Y. (1994), *Artificial Intelligence Techniques in Prolog*, Morgan Kaufmann.

Sterling, L. & Shapiro, E. (1986), *The Art of Prolog*, MIT Press.

Voronkov, A. (1992), Logic programming with bounded quantifiers, *in* A. Voronkov, ed., 'Logic Programming and Automated Reasoning—Proc. 2nd Russian Conference on Logic Programming', LNCS 592, Springer-Verlag, Berlin, pp. 486–514.

Wirth, N. (1985), *Programming in Modula-2*, third, corrected edn, Springer-Verlag, New York.

Wirth, N. (1986), *Algorithms and Data Structures*, Prentice-Hall, Englewood Cliffs, New Jersey, USA.