THE BRANCHING-TIME LOGIC PROGRAMMING LANGUAGE *Cactus* AND ITS APPLICATIONS

P. Rondogiannis, M. Gergatsoulis and T. Panayiotopoulos

3-97

Preprint no. 3-97/1997

Department of Computer Science
University of Ioannina
45 110 Ioannina, Greece

The Branching-Time Logic Programming Language *Cactus* and its Applications*

P. Rondogiannis[1], **M. Gergatsoulis**[2], **T. Panayiotopoulos**[3]

[1] Dept. of Computer Science, University of Ioannina,
P.O. BOX 1186, 45110 Ioannina, Greece,
e_mail: prondo@zeus.cs.uoi.gr

[2] Inst. of Informatics & Telecom., N.C.S.R. 'Demokritos',
153 10 A. Paraskevi Attikis, Greece
e_mail: manolis@iit.nrcps.ariadne-t.gr

[3] Dept. of Informatics, University of Piraeus
80 Karaoli & Dimitriou Str., 18534 Piraeus, Greece
e_mail : themisp@unipi.gr

*This work has been funded by the Greek General Secretariat of Research and Technology under the project "TimeLogic" of ΠΕΝΕΔ'95, contract no 1134.

Abstract

The notion of *tree* is a very common and useful one in computer science. Trees are used in all kinds of applications, ranging from simple sorting algorithms to sophisticated heuristics for solving intractable problems. It is therefore reasonable to investigate programming language paradigms in which the notion of tree is the main design criterion.

In this paper we introduce the new logic programming language **Cactus**, in which tree-related concepts can be described in a clear and elegant way. Cactus is in fact a temporal logic programming language, one in which the notion of time has a branching (and therefore tree-like) structure. As a result, Cactus appears to be especially appropriate for expressing non-deterministic computations or generally algorithms that involve the manipulation of tree data structures.

Keywords: Logic Programming, Temporal Logic Programming, Branching Time.

Contents

1 Introduction

Temporal programming languages[OM94, Org91] are recognized as natural and expressive formalisms for describing *dynamic* systems. For example, consider the following Chronolog [Wad88] program simulating the operation of the traffic lights:

```
first light(green).
next light(amber)  ←  light(green).
next light(red)  ←  light(amber).
next light(green)  ←  light(red).
```

However, Cronolog as well as most temporal languages[OM94, Hry93, OWD93, Bau93, Brz91, Brz93, GRP96] are based on linear flow of time, a fact that makes them unsuitable for certain types of applications. For example, as M. Ben-Ari, A. Pnueli and Z. Manna show in [BAPM83], branching time logics are necessary in order to express certain properties of non-deterministic programs.

In this paper we present the new temporal logic programming language **Cactus** which is based on a tree-like notion of time; that is, every moment in time may have more than one next moments. The new formalism is appropriate for describing non-deterministic computations or more generally computations that involve the manipulation of trees.

Cactus supports two main operators: the temporal operator `first` refers to the beginning of time (or alternatively to the root of the tree). The temporal operator $\texttt{next}_i$ refers to the i-th child of the current moment (or alternatively, the i-th branch of the current node in the tree). Notice that we actually have a family $\{\texttt{next}_i \mid i \in N\}$ of `next` operators, each one of them representing the different next moments that immediately follow the present one.

As an example, consider the following program:

```
first nat(0).
next0 nat(Y)  ←  nat(X),Y is 2*X+1.
next1 nat(Y)  ←  nat(X),Y is 2*X+2.
```

The idea behind the above program is that the set of natural numbers can be mapped on a binary tree of the form shown in figure 1. More specifically, one can think of `nat` as a time-varying predicate. At the beginning of time (at the root of the tree) `nat` is true of the natural number `0`. At the left child of the root of the tree, `nat` is true of the value `1`, while at the right child it is true of the value `2`. In general, if `nat` is true of the value `X` at some node in the tree, then at the left child of that node `nat` will be true of `2*X+1` while at the right child of the node it will be true of `2*X+2`. One can easily verify that the tree created contains all the natural numbers.

One could claim that branching time logic programming (or temporal logic programming in general) does not add much to logic programming, because *time* can always be added as an extra parameter to predicates. However, from a theoretical viewpoint this does not appear to be straightforward (see for example [Gab87, GHR94] for a good discussion on this subject). Moreover, temporal languages are very expressive for many problem

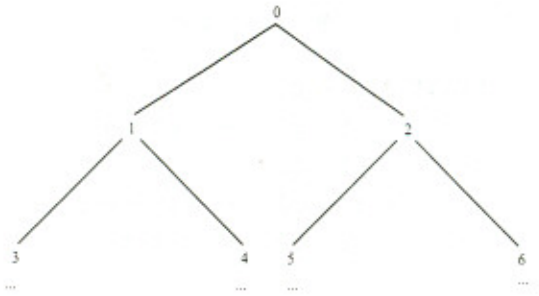

Figure 1: A mapping of the natural numbers on a binary tree

domains. As it will become apparent in the next sections, one can use the branching time concept in order to represent in a natural way time-dependent data as well as to reason in a lucid manner about these data.

2 The syntax of Cactus programs

The syntax of Cactus programs is an extension of the syntax of Prolog programs. In the following we assume familiarity with the basic notions of logic programming [Llo87].

A *temporal atom* is an atomic formula with a number (possibly 0) of applications of temporal operators. The sequence of temporal operators applied to an atom is called the *temporal reference* of that atom. A *temporal clause* is a formula of the form:

$$H \leftarrow B_1, ..., B_m$$

where $H, B_1, ..., B_m$ are temporal atoms, $m \geq 0$. If $m = 0$ then the clause is said to be a *unit temporal clause.* A *Cactus program* is a finite set of *temporal clauses.*

A *goal clause* in Cactus is a formula of the form $\leftarrow A_1, ..., A_n$ where A_i, $i = 1, ..., n$ are temporal atoms.

3 Cactus Applications

In this section we present various applications showing the expressive power of branching time logic programming.

3.1 Expressing non-deterministic behaviour

Consider the non-deterministic finite automaton shown in figure 2 (taken from [LP81] page 55) which accepts the regular language $L = (01 \cup 010)^*$. We can describe the behaviour of

this automaton in Cactus with the following program:

```
first state(q0).
next0 state(q1)  ← state(q0).
next1 state(q2)  ← state(q1).
next1 state(q0)  ← state(q1).
next0 state(q0)  ← state(q2).
```

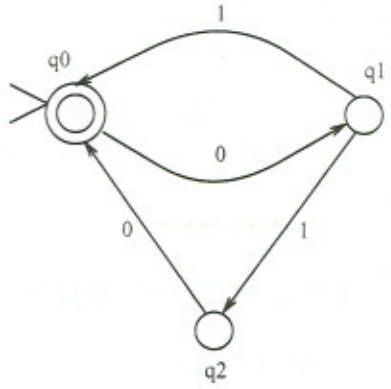

Figure 2: A non-deterministic finite automaton

Notice that, in this automaton $q0$ is both the initial and the final state. Posing the goal clause:

```
← first next0 next1 next0 state(q0).
```

will return the answer **yes** which indicates that the string 010 is an acceptable string of the language L.

3.2 Generating sequences

One can write a simple Cactus program for producing the set of all binary sequences. The set of such sequences may be thought of as a tree, which can be described by the following program:

```
first binseq([ ]).
next0 binseq([0|X])  ← binseq(X).
next1 binseq([1|X])  ← binseq(X).
```

The goal clause:

```
← binseq(S).
```

will trigger an infinite computation which will generate all possible sequences.

One can combine the program `binseq` with the program for the nondeterministic automaton given in subsection 3.1. In this way we can produce the language recognized by the automaton. More specifically, the goal clause:

$$\leftarrow \texttt{state(q0), binseq(S).}$$

produces the infinite set of all the binary sequences recognized by the automaton. The above goal clause (assuming a left to right computation rule) is not the classical generate-and-test procedure (not all binary sequences are generated but only those for which the automaton reaches the final state `q0`). Each succesful evaluation of the goal `state(q0)` conducts the corresponding evaluation of `binseq`.

It is worthwhile noting here that in order to generate another language one only needs to change the definition of the automaton and not the definition of `binseq`.

In order to code the same problem in ordinary logic programming (e.g. in Prolog), the notion of the *sequence* has to be added as an extra argument to the `state` predicate. The corresponding program is shown below:

```
state(q0, [ ]).
state(q1, [0|X])  ←  state(q0, X).
state(q2, [1|X])  ←  state(q1, X).
state(q0, [1|X])  ←  state(q1, X).
state(q0, [0|X])  ←  state(q2, X).
```

Given the goal clause:

$$\leftarrow \texttt{state(q0, S).}$$

Prolog's underlying execution engine would generate the sequences recognized by the automaton.

The Prolog version of the program is not as natural as the Cactus one. In Prolog, the sequences handled by the automaton have to be made explicit and "passed around" by the program, while in Cactus this is avoided with the use of the temporal operators.

3.3 Representing and manipulating trees

Branching time logic programming is a powerful tool for representing and manipulating trees. A tree can be represented in Cactus as a set of temporal unit clauses. The structure of the tree is expressed through the temporal references of the unit clauses. Moreover, the well known tree manipulation algorithms are easily and naturally expressed through Cactus programs. For example, consider the binary tree of figure 3.

A possible representation of the information included in this tree is given by the following set of Cactus unit clauses:

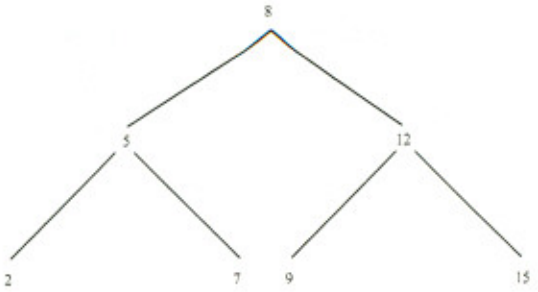

Figure 3: An (ordered) binary tree containing numeric data

```
first data(8).
first next0 data(5).
first next1 data(12).
first next0 next0 data(2).
first next1 next0 data(9).
first next0 next1 data(7).
first next1 next1 data(15).
```

The following program defines the predicate `descendant(X)`. A temporal atom of the form ⟨*Temporal reference*⟩ `descendant(X)` is true if `data(X)` is true in the time represented by ⟨*Temporal reference*⟩ or in a future moment of this time point.

```
descendant(X) ← data(X).
descendant(X) ← data(Y), next0 descendant(X).
descendant(X) ← data(Y), next1 descendant(X).
```

A more efficient definition of the predicate `descendant` which takes into account the fact that the binary tree is ordered (binary search) is shown in the following program.

```
descendant(X) ← data(X).
descendant(X) ← data(Y), X < Y, next0 descendant(X).
descendant(X) ← data(Y), X > Y, next1 descendant(X).
```

By posing the goal clause:

```
← first next0 descendant(7).
```

we will get the answer `yes`, because the value 7 is in a node which represents a moment in the future of `first next0`.

Using the definition of the predicate `descendant` we can define the predicate `search` which tests if a specific numeric value is in a node of the data tree. The definition of `search` is given by the clause:

```
search(X) ← first descendant(X).
```

Let us now define a predicate `flattree` which collects the values of the tree nodes into a list. This definition corresponds to the preorder traversal of the tree.

```
flattree([ ]) ← data(void).
flattree([X|L]) ← data(X),
        next0 flattree(L1),
        next1 flattree(L2),
        append(L1, L2, L).
```

Notice that the above program recognizes the tips of the tree when it encounters a `data(void)` unit clause. For this, we have to add the following unit clauses to the program:

```
first next0 next0 next0 data(void).
first next0 next0 next1 data(void).
first next0 next1 next0 data(void).
first next0 next1 next1 data(void).
first next1 next0 next0 data(void).
first next1 next0 next1 data(void).
first next1 next1 next0 data(void).
first next1 next1 next1 data(void).
```

A more compact representation of the above tree (that avoids the use of void nodes) would be to distinguish the (inner) nodes from the leafs of the tree by using two different predicate names e.g. `node` and `leaf` instead of the single predicate `data`. In that case we have to change slightly the definition of `flattree`. The new representation of the data tree becomes:

```
first node(8).
first next0 node(5).
first next1 node(12).
first next0 next0 leaf(2).
first next1 next0 leaf(9).
first next0 next1 leaf(7).
first next1 next1 leaf(15).
```

The new definition of `flattree` becomes:

```
flattree([X]) ← leaf(X).
flattree([X|L]) ← node(X),
        next0 flattree(L1),
        next1 flattree(L2),
        append(L1, L2, L).
```

Notice that the definition of the predicate **append** used in **flattree** is the usual one:

```
append([ ], L, L).
append([X|Xs], L, [X|R]) ← append(Xs, L, R).
```

The predicate **append** is independent of time and it performs the concatenation of its first two arguments.

3.4 Modeling Recursion Using Branching Time

In the following we present an example of how branching-time can be used to model the tree-like structure of recursion. Consider for example the usual way of computing the Fibonacci numbers in Prolog:

```
fib(1, 0).
fib(1, 1).
fib(F, N) ← N1 is N − 1, N2 is N − 2,
            fib(F1, N1), fib(F2, N2),
            F is F1 + F2.
```

Given the above program, a goal clause of the form:

```
← fib(F, 10).
```

will return the 10'th Fibonacci number.

During the execution of the above goal, the value of the parameter N of **fib** changes in a tree-like way (because of the recursive calls in the body of **fib**). We can define in Cactus a predicate **n** which models the change of the parameter N of the Prolog program (when N starts with initial value 10):

```
first n(10).
next_0 n(Y) ← n(N), Y is N − 1.
next_1 n(Y) ← n(N), Y is N − 2.
```

At the beginning of time, **n** is true of the value 10. In general, if **n** is true of the value N at some node in the tree, then at the left child of that node **n** will be true of N-1 while at the right child of the node it will be true of N-2.

We can rewrite the Fibonacci program in a purely branching time way:

```
fib(1) ← n(0).
fib(1) ← n(1).
fib(F) ← next_0 fib(F1), next_1 fib(F2), F is F1 + F2.
```

Given the above definitions for **fib** and **n**, a goal clause of the form:

```
← first fib(F).
```

will return the 10'th Fibonacci number.

The above program is definitely a less intuitive one than the original Prolog program. The reason is that in the Cactus program, while the tree of `n` is constructed in a top-to-bottom way, the tree of `fib` is beeing built bottom-up. Actually, the variables `N` and `F` of the original Prolog program would vary in exactlty this way during execution, a fact that is not however expressed explicitly in the Prolog program. In other words, the Cactus program has a more operational flavour because it expresses explicitly the recursion mechanism for computing the final result.

As we realize from the above example, it is possible for certain logic programs to be transformed into branching time logic programs that contain only unary predicates. A similar transformation exists for functional programs (see for example [Yag84, Ron94, RW97]) and has formed the basis for dataflow implementations of functional languages. An interesting question for further investigation is whether the technique we outlined in the above example, is applicable to wide classes of logic programs.

4 The branching time logic of Cactus

In this section we describe the branching time logic (BTL) on which Cactus is based. In BTL, time has an initial moment and flows towards the future in a tree-like way. The set of moments in time in BTL, can be modelled by the set $List(N)$ of lists of natural numbers. In this case, each node has a countably infinite number of branches (`next` operators). Similarly, we may choose a finite subset S of N and define the logic $BTL(S)$, which has a finite number of `next` operators (branches starting from each node). In any case, the empty list [ ] corresponds to the beginning of time and the list $[i|t]$ (that is, the list with head i and tail t) corresponds to the i-th child of the moment identified by the list t.

BTL uses the temporal operators `first` and $\texttt{next}_i$, $i \in N$. The operator `first` is used to express the first moment in time, while $\texttt{next}_i$ refers to the i-th child of the current moment in time. The syntax of BTL extends the syntax of first-order logic with two formation rules:

- if A is a formula then so is `first` A, and
- if A is a formula then so is $\texttt{next}_i$ A.

BTL is a relatively simple branching time logic. For more on branching time logics one can refer to [BAPM83].

4.1 Semantics of BTL formulas

The semantics of temporal formulas of BTL are given using the notion of *branching temporal interpretation*. Branching temporal interpretations extend the temporal interpretations of the linear time logic of Chronolog[Org91].

Definition 4.1. A *branching temporal interpretation* or simply a *temporal interpretation* I of the temporal logic BTL comprises a non-empty set D, called the domain of the interpretation, over which the variables range, together with an element of D for each variable; for each n-ary function symbol, an element of $[D^n \rightarrow D]$; and for each n-ary predicate symbol, an element of $[List(N) \rightarrow 2^{D^n}]$.

In the following definition, the satisfaction relation $\models$ is defined in terms of temporal interpretations. $\models_{I,t} A$ denotes that a formula A is true at a moment t in some temporal interpretation I.

Definition 4.2. The semantics of the elements of the temporal logic BTL are given inductively as follows:

1. If $\mathtt{f}(e_0, \ldots, e_{n-1})$ is a term, then $I(\mathtt{f}(e_0, \ldots, e_{n-1})) = I(\mathtt{f})(I(e_0), \ldots, I(e_{n-1}))$.
2. For any n-ary predicate symbol $\mathtt{p}$ and terms $e_0, \ldots, e_{n-1}$,
   $\models_{I,t} \mathtt{p}(e_0, \ldots, e_{n-1})$ *iff* $\langle I(e_0), \ldots, I(e_{n-1}) \rangle \in I(\mathtt{p})(t)$
3. $\models_{I,t} \neg A$ *iff it is not the case that* $\models_{I,t} A$
4. $\models_{I,t} A \wedge B$ *iff* $\models_{I,t} A$ *and* $\models_{I,t} B$
5. $\models_{I,t} A \vee B$ *iff* $\models_{I,t} A$ *or* $\models_{I,t} B$
6. $\models_{I,t} (\forall x)A$ *iff* $\models_{I[d/x],t} A$ *for all* $d \in D$ *where the interpretation* $I[d/x]$ *is the same as* I *except that the variable* x *is assigned the value* d.
7. $\models_{I,t} \mathtt{first}\ A$ *iff* $\models_{I,[\,]} A$
8. $\models_{I,t} \mathtt{next}_i\ A$ *iff* $\models_{I,[i|t]} A$

If a formula A is true in a temporal interpretation I at all moments in time, it is said to be true in I (we write $\models_I A$) and I is called a *model* of A.

Clearly, Cactus clauses form a subset of BTL formulas. It can be shown that the usual minimal model and fixpoint semantics that apply to logic programs, can be extended to apply to Cactus programs. However, such an investigation is outside the scope of this paper and is reported in a forthcoming one[RGP97].

4.2 Axioms and Rules of Inference

In this section we present some useful axioms and inference rules that hold for the logic BTL, many of which are similar to those adopted for the case of linear time logics [Org91]. In the following, the symbol ∇ stands for any of `first` and $\mathtt{next}_i$.

Temporal operator cancellation rules: The intuition behind these rules is that the operator `first` cancels the effect of any other "outer" operator. Formally:

$$\nabla(\texttt{first}\ A) \leftrightarrow (\texttt{first}\ A)$$

Notice that this is actually a family of rules, one for each different instantiation of the operator ∇.

Temporal operator distribution rules: These rules express the fact that the branching time operators of BTL distribute over the classical operators $\neg$, $\wedge$ and $\vee$. Formally:

$$\begin{array}{l}\nabla(\neg A) \leftrightarrow \neg(\nabla A)\\ \nabla(A \wedge B) \leftrightarrow (\nabla A) \wedge (\nabla B)\\ \nabla(A \vee B) \leftrightarrow (\nabla A) \vee (\nabla B)\end{array}$$

Again, each of the above rules actually represents a family of rules depending on the instantiation of ∇.

¿From the temporal operator distribution rules we see that if we apply a temporal operator to a whole program clause, the operator can be pushed inside until we reach atomic formulas. This is why we did not consider applications of temporal operators to whole program clauses.

Temporal operator non-commutativity rule: Of particular interest is the following rule which concerns the branching time operators:

$$\texttt{next}_i\ \texttt{next}_j\ A \not\leftrightarrow \texttt{next}_j\ \texttt{next}_i\ A,\ \textit{when}\ i \neq j$$

What the above rule states is that in general, two operators $\texttt{next}_i$ and $\texttt{next}_j$ can not be interchanged when i and j are different.

Rigidness of variables: The following rule states that a temporal operator ∇ can "pass inside" $\forall$:

$$\nabla(\forall X)(A) \leftrightarrow (\forall X)(\nabla A)$$

The above rule holds because variables represent data-values composed of function symbols and constants which are independent of time (i.e. they are *rigid*).

Temporal operator introduction rules: The following rule states that if A is a theorem of BTL then ∇A is also a theorem of BTL.

$$\textit{if}\ \vdash A\ \textit{then}\ \vdash \nabla A$$

The validity of the above axioms is easily proved using the semantics of BTL.