

BUILDING LOGIC CONSTRUCTS INTO PROCEDURAL PROGRAMMING LANGUAGES

Y. LIU

School of Applied Science, Nanyang Technological University

Singapore 2264

E-mail: asyliu@ntu.ac.sg

and

J. STAPLES

SVRC, University of Queensland

Brisbane, Queensland 4072, Australia

E-mail: staples@cs.uq.oz.au

ABSTRACT

Because Prolog's abstract machine is quite different from Von Neumann machines, its execution is generally slower. Building logic constructs into procedural languages is a way to increase efficiency. Earlier attempts had problems in expressiveness or efficiency. Through comparison of major logic related aspects of existing backtracking languages, a new backtracking procedural language is presented which has improved declarative expressiveness.

1. Introduction

Originating as a confluence of natural language processing, logic and automatic theorem proving in early 1970's, Prolog is a language based on predicate calculus. Its statements are sentences of a logic. The consequence of using logic to represent knowledge (programming) is that the knowledge can be understood declaratively. This simplifies programming. Prolog has drawn the attention of a fairly large number of computer scientists, both from universities and industry, refining the language and developing new applications. It has gained wide application in AI, theorem proving and natural language processing, for example.

Considered as a general-purpose programming language however, Prolog has limited data structures and slow memory access through a form of associative memory emulation. Because Prolog's abstract machine is very different from conventional machines, the standard implementation of Prolog compilers is the Warren Abstract Machine (WAM). Though the WAM implementation has many carefully engineered optimizations, aimed at reducing both processing time and memory consumption, Prolog execution is still generally slow compared to execution of a conventional procedural language program with equivalent functionality. For access to the code of procedures, Prolog's calling mechanism relies on emulation of associative memory access.

In Prolog, a program is expressed by means of Horn clauses and deduction is performed by backwards reasoning embedded in resolution. Backtracking is important for achieving declarative expressiveness and deduction. Thus, one of the approaches to achieve efficiency while still keep a declarative programming style is to build logic constructs into a procedural programming language. Procedural programming languages use conventional direct memory access which is more efficient than the associative memory access used in WAM implementations. By logic constructs, we mean those constructs which contribute to the creation and control of choice points and management of memory backtracking.

There have many attempts at procedural backtracking since 1960's. These languages or language proposals have constructs to express choice points, success and failure. That is they have built-in support for generation and search through a set of possible solutions. Typically their semantics recognize success or failure of execution, and support programming of choice points. Traditionally these languages are called *Backtracking Procedural languages*, as they are procedural programming languages and these built-in logic

constructs support backtracking algorithms. Representatives are: backtracking Pascal ⁴, ND-Fortran ¹, backtracking LISP ⁷, and Icon ³. However, most of them have not had wide application. The main reason is that none of the existing backtracking procedural languages is satisfactory both in efficiency and in expressiveness of choice points. The former ensures they are not used in practical applications, and the latter limits them to fewer applications and makes them uncompetitive with other declarative programming languages. In existing backtracking procedural languages, logic constructs support limited data types and limited styles of choice point, and the strategies they use to trail old values are relatively primitive and inefficient. For example, ND-Fortran and Icon record the old value of a backtracking variable whenever that variable is updated. That means much unnecessary trailing is done, so much computation time and space is wasted. Backtracking BCPL ⁹ spends a long time comparing two memory states to find out their differences, as part of trailing.

The integration of logic constructs into procedural languages could be improved. It is possible to preserve the computational structure and efficiency of procedural programming languages, and improve their expressiveness in programming choice points. At the same time, we can exploit and extend Prolog backtracking implementation techniques, which have made many of the mechanism used in existing procedural backtracking languages obsolete. Practically, this is demonstrated by the design of btC, an extension of C with logic constructs.

2. Stronger Backtracking Procedural Languages

A stronger backtracking procedural language should have the following properties.

- General-purpose – supporting general data structures and types. Contrasted to Prolog’s single data structure (ground term) that is a benefit of using procedural languages.
- Procedural – choosing the imperative as the basic, and supporting the declarative style too, because the former can be more efficiently implemented on conventional computers.
- Efficient backtracking – especially by designing new techniques, and using existing Prolog techniques where possible, to implement backtracking.
- Expressive – in programming and in controlling choice points.

btC is an extension of the C programming language with added logic constructs. The choice of C as the base language of btC meets the general-purpose and procedural requirements. New constructs have been added to: define choice points; enable backtracking control; and define program memory as either backtracking or non-backtracking. The basic addition to C syntax concern backtracking control and choice points creation. The basic change in semantics is the introduction of the concept of failure of computation and resumption of computation at an earlier stage. These additions are essential to a backtracking procedural language.

This paper will focus on the syntax and semantics of improved integration of logic constructs into procedural languages, with the practical discussion of the btC language. For detailed discussion of btC, see related publications ^{5,6}. Firstly, the creation of choice points is discussed. Then, the control of choice points is presented, followed by a study of memory backtracking.

3. Choice Point Codes

In procedural backtracking languages, we call an expression or a statement a *choice-point code* if it can create choice point(s), that is, cause multiple executions. It can cause execution to resume at the entry state and proceed along an alternative execution. A choice-point code whose alternatives are expressions may be called an *expression-based* choice-point code.

The expressiveness of choice points decides the expressiveness of declarative programming, and affects the extent to which backtracking languages can liberate programmers from details of backtracking.

3.1. Prolog

In Prolog, a procedure is a sequence of clauses in the database, comprising all clauses in the database whose head has a given predicate symbol and a given number of arguments. A procedure having more than one clause has the potential to create a choice point. Each clause defines one alternative of the choice point. In addition, the internal structure of the body of each clause may define additional choice points.

The expressiveness of Prolog in defining choice points is very powerful. Its alternative are clauses, which in turn can be any generic operations a programmer may want.

3.2. Existing Procedural Backtracking Languages

Most backtracking procedural language proposals allow choice points to be created only through the mechanism of multiple-valued functions. More flexible alternatives are however available.

Multiple-valued Functions. Floyd proposed a multiple-valued function `choice(X)`², which returns the positive integer `X` the first time it is called, then `X - 1, ..., 1` on successive returns to this choice point by backtracking. In Floyd's flowchart language this is the only means to establish choice points. Many other backtracking procedural languages follow this approach, though names differ and the order of the alternatives is sometimes reversed.

More Flexible Choice Points. There appear to be only two backtracking procedural language proposals which support a more flexible notion of choice points: one given by Prenner et al.⁸ and one presented in Icon³.

In Prenner et al's proposal, each choice point is implemented by a tag point which can be repeatedly resumed with the same forward execution code. A tag point in a program's execution sequence which is associated with the integer value `X` is created by execution of the primitive `TAG(X)`. The primitive `FAIL(X)` backtracks to the immediately preceding tag point with value `X`. The primitive `UNTAG(X)` will remove the tag with value `X` and all tag points created later.

Icon provides the user with generators to program choice points. Generators are multiple-valued expressions, and are classified as either built-in generators or programmer-defined generators.

The most commonly used built-in generators are: `i to j by k`, `expr1 | expr2`, `find(s1, s2)`, and `upto(c, s)` which are multiple-valued functions.

Programmer-defined generators are programmer-defined functions or procedures which can produce more than one result. During the execution of these generators, execution of the statement

`suspend expression`

returns the value of *expression* as the result of the execution of the procedure, then leaves the call in suspension with the values of local variables intact and transfers control back to the calling level. The suspended procedure can be resumed by backtracking, and execution continues from the point of suspension.

3.3. btC

The methods available in previous backtracking procedural languages for programming choice points have limitations. Only expression-based choice-point codes are available in previous backtracking procedural languages, except Icon. Also, alternative values must be integers.

These expression-based choice points have limitations in practice: they are best suited to combinatorial searches, and may be inconvenient when the alternatives at a choice point can be comprehensive actions.

Icon allows construction of choice-point codes using randomly ordered expressions. Icon's generator mechanism can execute different codes before returning each alternative value, and so provides a form of

backtracking for statements, but not a convenient general syntax.

Prolog's expressiveness goes well beyond existing backtracking procedural languages. It is based not on alternative expressions, but on alternative clauses in the definition of predicates.

For stronger expressiveness, we follow Prolog's support for choice-point codes with generic operations as alternatives, expressed in general syntax.

On the other hand, expression-based choice-point codes have their own advantages. In procedural languages, most common operations are expression based. The function `choice()` concisely expresses multiple-valuedness, is standard in previous backtracking procedural languages, and has good applications in combinatorial searching. Choice-point codes based on random combinations of expressions using `or` provide additional flexibility as the data type is no longer a limitation.

Expression-based choice-point codes can in principle be replaced by statement-based choice-point codes, but the reverse is not easily done. That underlines the importance of statement-based choice-point codes in procedural backtracking languages.

The btC language supports both statement-based and expression-based choice-point codes. Its statement-based choice-point codes related production is:

statement:

```
{ statement } or { statement }
```

Thus one alternative can perform a comprehensive action as required. The introduction of statement-based choice-point codes allows more concise and natural code. Choice-point code statements can allow the alternatives of a choice point to be any valid statement. That is to say, an alternative of a choice point could be, for example, a `for`-statement, an input/output function call, or a compound statement which is a sequence of statements. Thus one alternative can perform a comprehensive action as required.

The relevant productions for expression-based choice-point codes are:

expression:

```
expression or expression  
choice( expression )
```

The semantics of `or`-expressions are similar to those of `or`-statements. Syntactically, sub-expressions of an `or`-expression can be any valid btC expressions. The precedence of the `or` operator, which is left associative, is higher than `<<` and `>>` but lower than `*` and `/`.

The function `choice(expr)` is semantically equivalent to:

```
if (expr < 1)  
    fail;  
1 or 2 or ... or N
```

where `N` is equivalent to $\lceil expr \rceil$, that is, the largest integer which is less than or equal to expression `expr`. It is straightforward to declare such a function `choice`, but the predefined function evaluates faster.

4. Control of Choice Point

When computation proceeds after selecting one of the possible ways at a choice point, it will continue until it succeeds (finds a solution) or fails (finds the sequence of choices made so far is inconsistent with solving the stated problem). On failure, the algorithm backtracks to its state immediately before the most recent choice point was executed, selects one of the untried choices at this point and resumes execution. If all possible choices have already been attempted, the algorithm backtracks to the previous choice point, if any, or else fails. In case of success, that is if a solution is found, computation may stop or may backtrack for more solutions, depending on the program or the user. This depth-first execution order is the most commonly used backtracking strategy. Typically, logic constructs related to control of choice points are dealing with

success, failure of execution, and the selection of alternatives.

4.1. Prolog

Control backtracking in Prolog goes from a failure point directly to the latest choice point. Two built-in (meta) predicates can modify backtracking: `cut(!)` and `once`. A cut discards all alternatives of choice points created since entering the procedure in which the cut occurs. In other words, it commits the system to every choice it has made since the procedure call began execution. On the other hand, `once(sequence_of_predicate)` forces `sequence_of_predicate` to execute once only. That is, when computation exits from `once`, all choice points created during execution of `sequence_of_predicate` are discarded.

These backtracking control predicates increase the efficiency of Prolog programs by: (1) eliminating the time needed to execute alternatives that the programmer can tell beforehand will never contribute to a solution; (2) saving space which would otherwise be needed to record the choice points.

4.2. Existing Procedural Backtracking Languages

Choice point control is the most basic part of any procedural backtracking language. We classify existing types of control either as full or restricted.

Full Control Backtracking. In backtracking procedural languages with this feature, all alternative choices in a created choice point will be tried one by one in successive backtracking. The `fail` statement is the only way to trigger backtracking. ND-Fortran's control backtracking is of this type. Following Floyd's flowchart language, it is implemented in a step-by-step fashion as follows. Each command is compiled into a forward part and a backward part. The forward part executes the normal semantics and trailing. The backward part undoes the effect of the corresponding forward part. When backtracking starts by failure of a command, it reverses the execution sequence to return to the latest choice point, by executing the backward part of each command.

Restricted Control Backtracking. The existing backtracking procedural language with this feature is Icon. Alternative choices remaining at a choice point are removed when exiting certain contexts. Icon supports a form of restricted backtracking in its use of bounded expressions and the limitation control structure. If an expression is bounded and it produces a result, all suspended generators in it are discarded. But, expressions are bounded in specific context, rather than using a defined area. Nor can a user define his/her own bounded contexts.

Both types of control backtracking have limitations. In full control backtracking, the alternatives are attempted one by one from left to right, when failures cause successive backtracking. This is exhaustive search. In Icon, the bounded context is pre-defined in the syntax. If, after checking one alternative of a choice point A, we are sure it is unnecessary to try the remaining untried alternatives of choice point A, what can we do?

4.3. btC

In btC `fail` is a natural way to generate failure and is an important part of backtracking support.

In previous backtracking procedural languages other than Icon, execution of the alternatives of a choice point is controlled only by failure. Prolog has a very efficient way to control backtracking through cuts, and btC takes the same approach.

Since C-style programming is not as highly structured as Prolog programming, there is no obvious default place to which a cut might revert. Neither can we implement the `once` operation as in Prolog. Hence the btC cut is implemented by a `cutpoint` and `cut` pair as follows.

The relevant productions of btC's grammar are:

expression:

`cutpoint()`

statement:

`cut(cutpoint)`

Here *cutpoint* must be a variable of integer type.

A call on the built-in function `cutpoint()` returns an integer value, which is unique to that call. The call has no side effects. The parameter supplied to the built-in function `cut(x)` should hold a value returned by a previously executed `cutpoint()` function. Then, `cut(x)` discards all the choice point(s) created since the latest execution of `x = cutpoint()`. We can say it commits to the choices the execution has made since then. Backtracking to choice points created between the execution of `cutpoint()` and `cut(x)` will not occur.

This is an important tool to enable more efficient use of btC. Programs may: execute faster because no time is wasted trying alternatives which it is known will never contribute to a solution; and use less memory space because removal of choice points also allows removal of some information kept for control backtracking and memory backtracking.

5. Memory Backtracking

Alternative executions defined by a choice-point code should all start from the same computation state, as defined by the backtracking principle. Thus memory backtracking in a backtracking language should come along with control backtracking, when that is triggered by failure. As we will see, memory backtracking is not fully present in existing procedural backtracking languages.

5.1. Prolog

All logical variables in Prolog are backtracked during control backtracking. However, backtracking does not affect the database contents (which are changed by executing built-in Prolog predicates). Input and output operations are also excluded from backtracking.

'Pure' Prolog, a minimal Prolog subset without non-logical features, is a full memory backtracking language. The database update facility of real Prologs makes them partial memory backtracking.

5.2. Existing Procedural Backtracking Languages

In existing procedural backtracking languages, memory backtracking can be defined as either full memory backtracking or partial memory backtracking. The main characteristics of each type are as follows.

Full Memory Backtracking. Languages with this property reset the whole memory state during backtracking. All changes to variable values are undone, to restore the same state of memory as existed at the state of the previous execution of the choice point. ND-Fortran ¹ and backtracking BCPL ⁹ are of this type.

Partial Memory Backtracking. In partial memory backtracking some variables of a program may be not backtracked: a variable is backtracked only if that is specified in the program. In backtracking SIMULA, only variables in the BACKTRACK block are backtracked. In Icon ³ only reversible assignments, that is assignments with the special assign operator '<->' are backtracked. The same is true of backtracking ECL ⁸, where the operator '<=>' is used.

5.3. btC

For a stronger procedural backtracking language, to support full memory backtracking or not will be decided by programming convenience and execution efficiency. It is easy to support full memory backtracking,

as programmer need not to bother specify which part need (not) be backtracked. From the point of execution efficiency, the more memory to be backtracked, the lower the efficiency. That suggests we support partial memory backtracking.

In btC, partial memory backtracking is more suitable for our objectives, because: (1) program memory used by base language programs should be non-backtracking, so as to avoid degrading their performance; (2) backtracking may be needed for only part of the algorithm, while non-backtracking variables ('database variables') record the progress of the rest of the algorithm; and (3) efficiency can be improved and memory consumption reduced by not trailing 'scratch' variables.

Accordingly, partial memory backtracking is supported in the btC language. All variables of a btC language program are declared with specified backtracking attributes.

The backtracking attribute of a variable is defined by the declared storage class of the variable. Variables must be declared with one of the ten storage classes (`local` and `auto` are synonyms) listed below.

```
extern local (auto) static register
bt_extern bt_local bt_static
cbt_extern cbt_local cbt_static
```

The storage class `local` is the default, and need not be declared explicitly. The storage classes of btC extend the C language's four storage classes `extern`, `static`, `auto`, and `register`, to cope with backtracking attributes.

The storage class `register` has the same meaning as the corresponding C storage class. The variables so declared are non-backtracking. Because the addresses of variables declared with `register` storage class are unreadable, btC has no backtracking storage classes corresponding to storage class `register`.

The suffixes `extern`, `local` and `static` indicate the memory location of the variable: `extern` and `static` have the same meaning as the corresponding C storage classes, and `local` corresponds to C's `auto` storage class.

The prefix of a storage class name specifies backtracking behaviour. If a variable is declared by a `extern`-postfixed storage class, and later declared by a `local`-postfixed or `static`-postfixed storage class, these storage classes must bear the same prefix. Declarations with unprefix storage classes (we call them default storage classes), declare non-backtracking variables. Storage classes with the prefix `cbt_` comprise compact-backtracking variables. Storage classes with the prefix `bt_` comprise fast-backtracking variables.

The default for storage classes is non-backtracking so that existing C programs, and especially the C system header files, can be used in the btC language without change.

Real procedural languages, and in particular C, use dynamic memory because it can meet memory requirements which are unpredictable at programming time. Support for backtracking in dynamic memory is therefore important to demonstrate efficient backtrackability. The requirements are: dynamic memory support should be able to be defined as either backtracking or non-backtracking, and its lifetime and behaviour should be consistent with its allocation and backtracking attributes.

To meet these requirements, btC can create and release segments of dynamic memory. Dynamic memory segments in btC are either compact-backtracking or non-backtracking; the choice between these alternatives is made by the programmer's choice of allocation system routine.

Non-backtracking dynamic memory system routines are: `malloc(size)`, `calloc(count, size)`, `realloc(ptr, size)`, and `free(ptr)`. The contents of non-backtracking dynamic memory are not affected by backtracking.

The system routines of btC relating to backtracking dynamic memory are: `bt_malloc(size)`, `bt_calloc(count, size)`, and `bt_free(ptr)`. The effect of these backtracking related functions is undone on backtracking. Backtracking over the allocation function for a backtracking dynamic memory deallocates

the memory, for example.

6. An Example

The following example of a btC program solves the problem of colouring a planar map so that no two adjoining regions have the same colour. It is known that four colours are sufficient to colour any planar map. Figure 1 gives a simple map which requires four colours to be coloured correctly.

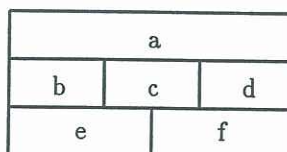


Fig. 1. A map requiring 4 colours

The program below solves this map-colouring problem. It uses a generate-and-test programming technique, modified from the Prolog algorithm in Section 14.1 in Shapiro's book ¹⁰, to exploit the procedural ability of btC.

The map is composed of a list of regions, and is represented by the variable `region`, an array of structure `REGION`. Each region is represented by an element of `region`, which has a name (we use integers as names here to simplify the algorithm), a colour (integer values are used here too), and information about its neighbours in the list `neighbour_list`. The variable `neighbour_list` is an array of integer, and records the addresses of neighbour regions of every region (their indexes in the array `region`). The element `neighbour` and `neighbour_nr` of structure `REGION` record the index of a region's first neighbour in `neighbour_list` and the number of this region's neighbour.

The core of the program is the function `colour_map(in_region, in_region_size, colours, colours_nr)`. Here, `in_region` is a list of regions to be coloured, `in_region_size` is the number of regions, `colours` is the head of the list containing the colours used to colour, and `colours_nr` is the number of colours used to colour. For each region in turn a colour is chosen from the colour set (using multiple forward execution) and the function `not_member(match_colour, in_neighbours, in_neighbour_nr)` is used to verify the colour chosen has not been used by any of its neighbours. Otherwise, a failure is created, to try alternative colours.

The data initialization part is omitted to save space.

```
btfn( colour_map, not_member);
typedef char * STR;
typedef int COLOUR;
bt_local int neighbour_list[1000];
struct REGION {
    STR name;
    COLOUR colour;
    int neighbour; /* start position in neighbour_list */
    int neighbour_nr; /* number of neighbours */
};

bt_local struct REGION region[100];

local int colours[4];
local int i;

void
not_member(match_colour, in_neighbours,
```



```

        in_neighbour_nr)
    COLOUR match_colour;
    int in_neighbours;
    int in_neighbour_nr;
    {
        for(i=in_neighbours;i<in_neighbours+in_neighbour_nr;i++)
            if ((region[neighbour_list[i]].colour <> 0) &&
                (region[neighbour_list[i]].colour==match_colour))
                fail;
    }

    void
    colour_map(in_region, in_region_size, colour, colour_nr)
    struct REGION, * in_region;
    int in_region_size, colour_nr;
    COLOUR * colour;
    {
        bt_local int i;

        for(i = 0; i < in_region_size; i++) {
            in_region[i].colour=*(colour+choice(colour_nr)- 1);
            not_member(in_region[i].colour,
                in_region[i].neighbour,
                in_region[i].neighbour_nr);
        }
    }

    void
    main()
    {
        /* Initialization of data. */

        for(i = 0; i < 4; i++)
            colours[i] = i + 1;
        /* Call the function to find the solution. */
        colour_map(region, 6, colours, 4);
        /* Print the result. */
        for(i = 0; i < 6; i++)
            printf("(%s,%i)",region[i].name,region[i].colour);
        printf("\n");
    }

```

7. Conclusion

Compared with existing backtracking procedural languages, btC's logic constructs have improved expressiveness to create choice points, and a greater ability to manage choice points. It inherits the power of the C language as a systems programming language, including its various advanced data types, flexible control structures, and efficient lower level memory access; provides more powerful choice point facilities than are available in existing backtracking procedural languages; provide more backtracking control mechanisms

than are available in existing backtracking procedural languages, so as to make programs more efficient; and achieves efficient implementation by exploiting recent developments in backtracking control mechanisms, and by creating new mechanisms.

8. References

1. J. Cohen and E. Carton, Non-deterministic FORTRAN, *Journal of ACM*, Vol. 17, No. 1, 44–51, 1974.
2. R. W. Floyd. Non-deterministic Algorithms, *Journal of ACM*, Vol. 14, 636–644, 1967.
3. R. E. Griswold and M. T. Griswold, *The Icon Programming Language, second Edition*, (Prentice Hall 1990).
4. G. Lindstrom, Backtracking in a generalized control setting, *ACM Trans on Programming Languages and Systems*, 1(1), 8–26, 1979.
5. Y. Liu, Stronger Procedural Backtracking: the Language btC, *PhD thesis*, Department of Computer Science, University of Queensland, Australia, 1992.
6. Y. Liu and J. Staples, btC: an Extension of C with Backtracking, *Proceedings of the Sixth International Conference on Symbolic and Logical Computing*, (Madison, South Dakota, USA, 1992) 59–72.
7. C. Montangero, G. Pacini, and F. Turini, Two-Level Control Structure for Nondeterministic Programming, *Comm ACM*, Vol. 20, No. 10, 1977, 725–730.
8. C. J. Prenner, J. M. Spitzen, and B. Wegbreit, An Implementation of Backtracking for Programming Languages, *The ACM National Conference Proceedings*, Vol. 2, 1972, 763–771.
9. J. A. Self, Embedding Non-determinism, *Software - Practice and Experience*, Vol. 5, 1975, 221–227.
10. L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, (MIT Press 1986).