On the Implementation of Dynamic Evaluation

P. A. Broadbery
The Numerical Algorithms Group, Ltd
Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR, UK
peterb@nag.co.uk

T. Gómez-Díaz *
Laboratoire d'arithmétique, calcul formel et optimisation (URA 1586)
123, Av. Albert Thomas. University of Limoges, France
tgomez@marie.polytechnique.fr

S. M. Watt
IBM T.J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598 USA
smwatt@watson.ibm.com

Abstract

Dynamic evaluation is a technique for producing multiple results according to a decision tree which evolves with program execution. Sometimes it is desired to produce results for all possible branches in the decision tree, while on other occasions it may be sufficient to compute a single result which satisfies certain properties. This technique finds use in computer algebra where computing the correct result depends on recognising and properly handling special cases of parameters. In previous work, programs using dynamic evaluation have explored all branches of decision trees by repeating the computations prior to decision points.

This paper presents two new implementations of dynamic evaluation which avoid recomputing intermediate results. The first approach uses Scheme "continuations" to record state for resuming program execution. The second implementation uses the Unix "fork" operation to form new processes to explore alternative branches in parallel.

These implementations are based on modifications to Lisp- and C-based run-time systems for the Axiom Version 2 extension language (previously known as $A^\sharp$). This allows the same high-level source code to be compared using the "re-evaluation," the "continuation," and the "fork" implementations.

*Partially supported by IBM Research contract N. 40140052 and PoSSo Esprit/Bra 6846

Reprinted from pp. 77–84, *Proc. 1995 International Symposium on Symbolic and Algebraic Computation (ISSAC 95)*, A. H. M. Levelt editor, ACM Press 1995.

1 INTRODUCTION

Consider the following example:

$$rank \begin{pmatrix} 1 & 1 \\ 1 & a \end{pmatrix}$$

When examining this example it is clear that the entries are of two different kinds: some are numbers and one is a *parameter*. The value of this expression is:

$$\begin{array}{l} 1 \;\; \textit{if} \;\; a = 1 \\ 2 \;\; \textit{if} \;\; a \neq 1 \end{array}$$

and this can be obtained easily by *hand*. When computing the rank of the matrix, the question $a = 1$ appears. Obviously, if one does not know whether a is equal to 1, *both* results are possible, and if one computes using both possibilities, one gets the right answer.

This, however, is *not* what one gets using a computer algebra system; there, the usual answer is simply "2". Computer algebra systems will normally provide only *one* result, possibly querying the user or selecting the "more general" branch during the course of the computation. This can lead to problems when the conditions under which the branch is valid are not recorded. For example, related questions can arise during the course of a computation, and there is no guarantee that the results of querying the user or automatically selecting branches will be consistent. Even when the choices are consistent, the conditions for validity are not presented as part of the answer in available algebra systems.

A proposal has existed for some time in the Maple community to return this list of conditions via a variable, the "proviso" [CJ], and an experimental version of Maple's "`solve`" command has been created to test these ideas [La].

The second difficulty is to obtain a complete solution covering all cases of a general mathematical problem. There do exist packages for particular algorithms which return results covering different cases (as for example [Si]). The next step is to understand how to use such packages together, composing the results of sub-problems. This is not straight-forward, since the course of an algorithm will likely vary depending on the conditions placed on the different cases. One does not wish to obscure the intent of every function in an algebra system with back-tracking logic. Even if one were willing to accept this, it would be necessary to modify all of the code in the entire system.

Dynamic evaluation addresses this problem and gives a method of proceeding when there are many possibilities for the answer [DR]. It has been implemented on the computer algebra system Axiom Version 1 [JS] and has been applied first in computations involving algebraic numbers [DDD, DD, Du], which are considered as a special kind of parameter. Dynamic evaluation is intrinsically parallel, but in earlier implementations the parallelism is only simulated, and some parts of the computation are performed many times. This redundant computation was unavoidable in the Axiom system, as it provides no mechanism for restarting a computation at a point in the middle of a program.

A possible way of avoiding redundant computation is to use continuations [R4R]. Continuations provide a mechanism for marking a point in a computation (in the middle of an equality in our example) and to return to this point as many times as desired. This allows the back-tracking primitive that dynamic evaluation needs. The question now is how to use continuations from within a computer algebra system. This has been made possible by the new Axiom compiler, called $A^\sharp$.

In this paper we describe dynamic evaluation and its utilisation in some computations involving parameters. We describe how continuations may be used to avoid redundant computation in the dynamic evaluation of a program, and how continuations may be used from within the Axiom Version 2 system. Section 2 describes the form of the domains over which dynamic evaluation can be made. This section then goes on to describe the splitting trees which can be used to represent a particular computation over these domains. This section then gives a more extended example of such a tree. Section 3 describes the implementations of dynamic evaluation, and describes the interface to $A^\sharp$.

2 DYNAMIC EVALUATION AND COMPUTING WITH PARAMETERS

Dynamic evaluation is a process of calculation that allows the execution of a program even where several answers are possible for some questions appearing in the program. It is described rigorously using sketch theory [DR].

Dynamic evaluation was first applied to computations with algebraic numbers, implemented as the *dynamic algebraic closure* of a field, first written in Reduce (known as the D5 system), and later in Axiom [DDD, DD, Du]. In this program the parameters are algebraic numbers, represented by symbols under algebraic constraints. This domain handles questions of equality over itself, using an algorithm based on the gcd.

The Axiom version of the dynamic algebraic closure is composed of several categories, domains and packages [Du][1]. In particular there is the control package with the function `allCases`. This function manages the progress of a dynamic evaluation. It is done in a *general way that is independent of the particular application under consideration.* This was possible because dynamic evaluation is a general principle, and also thanks to the polymorphism of Axiom.

[1]this version is referred to later as the old implementation of dynamic evaluation

Other applications of dynamic evaluation implemented in Axiom are, for example, the *dynamic algebraic real closure* [DGV] and the *dynamic constructible closure* [Go]. They use (without modification) the same control package. In the dynamic algebraic real closure parameters are also algebraic numbers, but additionally allow sign tests ($>$). The sign test implies the use of algorithms from real closed fields which makes the implementation more difficult (in fact, this work is still in progress).

The dynamic constructible closure of a field allows only the equality test, but here one deals with parameters in a more general sense than the algebraic one. In this domain one can get the full answer for the example in the introduction. As in the algebraic case, the gcd is the tool used in answering equality tests. We explore this domain further in the next section.

2.1 The Dynamic constructible closure

Dynamic constructible closure is an Axiom constructor (i.e. a domain-producing function) that provides parameters. It needs a ground field $\mathbf{K}$, which can be *any* field and produces a new Field with additional operations. If $\mathbf{K}$ is a subfield of complex numbers, the domain deals with complex parameters, that is parameters that take a complex number as their value.

In Axiom, one builds the dynamic constructible closure of a field $\mathbf{K}$ in the following way:

```
CL:= DynamicConstructibleClosure(K)
```

The result is a Field which also provides a function to introduce parameters:

```
newElement: Symbol -> CL
```

In addition, there are two functions to impose or forbid values for the introduced parameters, i.e. imposing constraints over the parameters:

```
areEqual: (CL,CL) -> Boolean
areDifferent: (CL,CL) -> Boolean
```

The boolean result of these operators says whether a new constraint is compatible with the previous ones. Constraints over parameters express the possible values for a parameter. At the moment of their introduction, they are reduced into a standard form in a recursive way, which reduces its presentation to the case of a parameter a over the ground field $\mathbf{K}$. The constraints on a are in one of the following forms:

- `anyElement`: there is no constraint on a (this means that a can take any value)
- `algebraic`: there is a constraint of type $P(a) = 0$ with P a monic univariate polynomial of positive degree with coefficients in $\mathbf{K}$ (this means that a can take as value any zero of P).
- `exception`: there is a constraint of type

  $$P_1(a) \neq 0 \textit{ and } \ldots \textit{ and } P_k(a) \neq 0$$

  with $P_1, \ldots, P_k$ monic univariate polynomials of positive degree with coefficients in $\mathbf{K}$ and pairwise coprime (this means that a can take any value different from any zero of any of the polynomials $P_1, \ldots, P_k$)

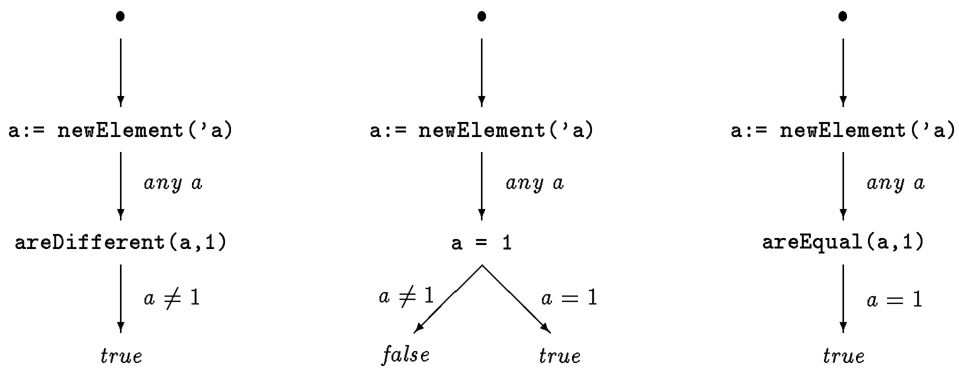

Figure 1: Three basic trees.

In addition, when the characteristic of the ground field is zero, we can suppose that polynomials $P, P_1, \ldots, P_k$ above are square-free.

The main point now is that parameters can take different values, so that it is in general impossible to answer `true` or `false` to an *equality test* over parameters. When both answers are possible, it is essential to distinguish the values of the parameters corresponding to `true` from the values corresponding to `false`. This is called a *splitting*, and it is detected using gcd computations.

One can use this program to solve polynomial systems or in mechanical geometry theorem proving. Also, one can get Jordan canonical form for matrices with parameters in its entries (see [Go]).

To illustrate the use of this program, we show the code (slightly simplified) corresponding to our example in the introduction:

```
RN:= Fraction Integer
CL:= DynamicConstructibleClosure(RN)
M := Matrix(CL)

dynamicRank():NonNegativeInteger ==
  a:CL:= newElement('a)
  m:M:= [[1,1],[1,a]]
  rank m

allCases(dynamicRank)
```

In the first line we say that our ground field is the rational numbers, in the second we build its constructible closure, `CL`. `CL` is a field in Axiom sense, so it makes sense to build matrices over it. Next we write a function with the computation to be done. It starts with the introduction of the necessary parameters and after building the matrix, calls rank over it. Finally we call this function with `allCases` in order to get the full answer:

```
[value is 1 in case a = 1,
 value is 2 in case a /= 1]

Time: 0.12 sec
```

We remark that `rank` is a function from Axiom. This function has no knowledge of dynamic evaluation, and is used without modification.

2.2 Splitting trees

Dynamic evaluation associates a *splitting tree* with a computation. It is built by the `allCases` function as computation proceeds.

We explain here splitting trees related to computations in the dynamic constructible closure domain. They can be defined in the following way:

1. The root node represents the beginning of the computation.
2. The edges represent current information about parameters, that is constraints over parameters in their standard form.
3. The nodes represent the points in the computation where the constraints can change — this can happen with the introduction of a new parameter (with `newElement`), and in the use of `areEqual`, `areDifferent` or the equality test (=).

The third point is the essential one, we will extend it more precisely, first for the case of only one parameter over the ground field, then for the case of many parameters, where recursion appears.

Case of one parameter. Let a be a parameter over the field $\mathbf{K}$, and `x` and `y` two elements in the field $\mathbf{K}(a)$. At every node of the splitting tree, except the root node, there is one of the following possibilities:

- `areEqual(x,y)`. There is only one edge coming down from this node, it corresponds to:
  - `true` if `areEqual(x,y)` implies a constraint on a which is compatible with the old ones,
  - or `false` otherwise. In this case, current constraints on a do not change.
- `areDifferent(x,y)`. There is only one edge coming down from this node, it corresponds to:
  - `true` if `areDifferent(x,y)` implies a constraint on a which is compatible with the old ones,
  - or `false` otherwise. In this case, current constraints on a do not change.
- `x = y`. In this case we have several possibilities:

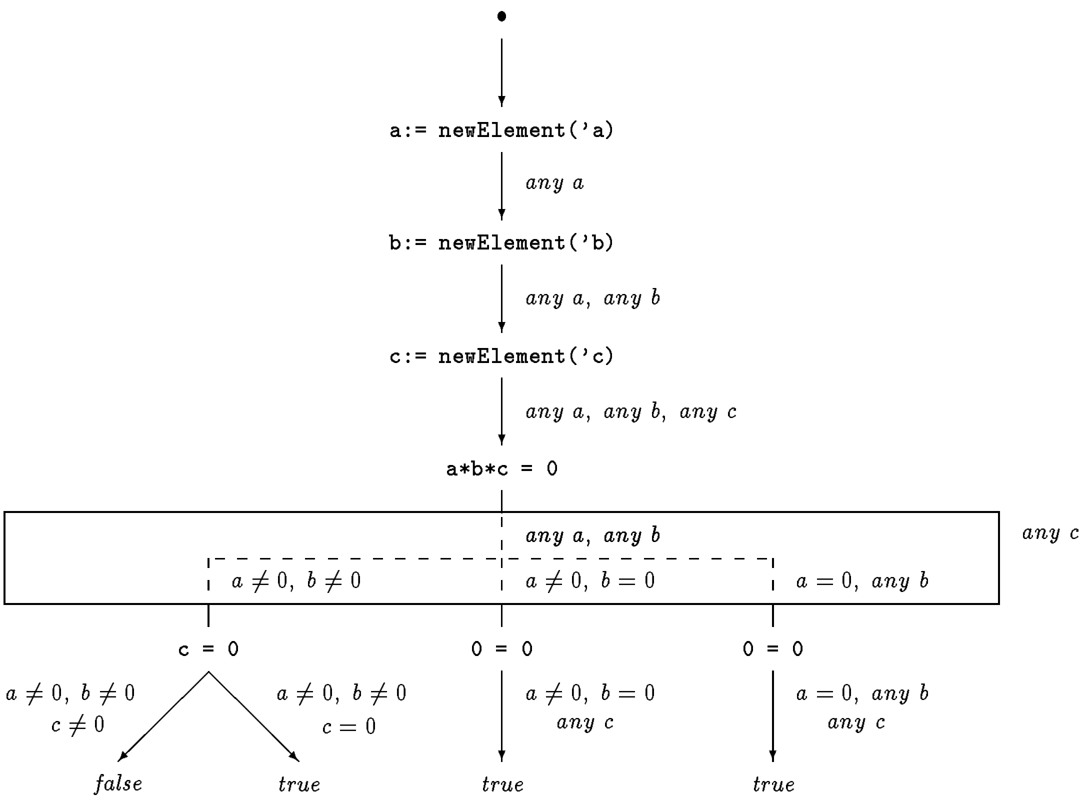

Figure 2: Splitting tree with box.

- the answer is `true` for all the possible values of a. This node has only one edge coming down, and current constraints on a do not change.
- the answer is `false` for all the possible values of a. This node has only an edge coming down, and current constraints on a do not change.
- both answers, `true` and `false`, are possible. There are two edges coming down from this node, one for each answer. This is called a *splitting point*.

Figure 1 shows three basic trees for the case of one parameter. The second tree in this figure corresponds to the example of the matrix rank computation.

Case of many parameters. Let $a_1, \ldots, a_n$ be the parameters introduced. The main point is that the program works recursively over parameters and then the use of `=`, `areEqual` or `areDifferent` over expressions of level k ($k \leq n$) can produce equality tests (and then splitting points) in lower levels.

Initially we show the last level only, with possible internal splittings in lower levels represented by a *box*. From this box, there is at least one edge coming down, but possibly many edges. During this process only constraints related to lower levels could change.

In order to show it we will study the following example:

```
a:CL:= newElement('a)
b:CL:= newElement('b)
c:CL:= newElement('c)
(a*b*c = 0)$CL
```

This corresponds to the splitting tree in figure 2. In the root node computation starts, in the second one there is the introduction of the first parameter a. From this node there is an edge coming down with the current information over this parameter that is *any a*. In the next node there is the introduction of the second parameter, and after the introduction of the parameter c we find the equality test `a*b*c = 0`. In order to answer this question the system works recursively over the parameters and some splitting points appear for parameters a and b. This splitting tree hides these splits in a box in which the current constraint on c (*any c*) does not change. From this box there are three edges coming down, with the current information about parameters a and b after split. For each edge we find a node with the question `a*b*c = 0` specialized with respect to these new constraints. In two nodes there is `0 = 0`, which we can answer now without additional split over c. In the other node we find the question `c = 0` that needs a split over c in order to be answer.

Finally the complete answer is:

```
[value is true in case any c and any b and a = 0,
 value is true in case any c and b = 0 and a /= 0,
 value is true in case c = 0 and b /= 0 and a /= 0,
 value is false in case c /= 0 and b /= 0 and a /= 0]

Time: 1.46 sec
```

Let us now open the box (see figure 3). In order to answer the question `a*b*c = 0` the system needs to answer `a*b = 0` and for that `a = 0`. One split appears at the level of a, that is inside the inner box. There are two edges coming down from this box, for each of them there is a reduction step to specialize the question `a*b = 0` to the new constraints on

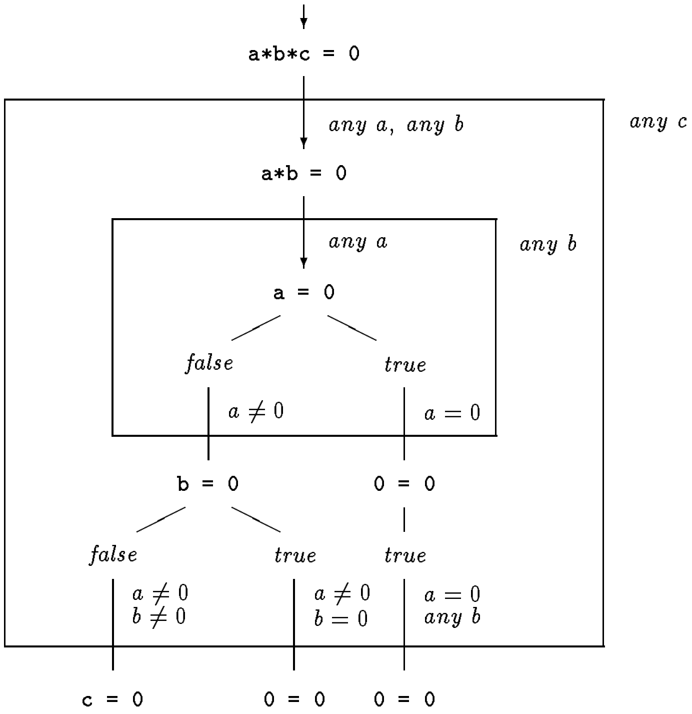

Figure 3: Splitting tree with open box.

a. In one of the branches this question can be answered without more splits, in the other it is needed another split over b.

Note that boxes are used to show the change of level in the computation, where recursion acts. Also there are some boolean values that appear, they only clarify the meaning of the branch (they can be seen as a flag for the branch).

3 IMPLEMENTING DYNAMIC EVALUATION

3.1 Strategies

The splitting trees show what kind of parallel problems that dynamic evaluation generates: from a given splitting point, each case is independent of the others, and consequently may be evaluated by a separate thread of execution.

Most common parallel libraries (including RPC, Linda, and PVM) assume that a parallel task can be described by a function, plus some arguments, and the parallel execution simply obtains a processor, branches to the function and when the function returns, marks itself as finished in some way. Unfortunately, dynamic evaluation is not suited to these kinds of model, as the parallel work is the remainder of the computation (ie. a continuation) rather than a well defined subpart of it.

In the previous implementation of dynamic evaluation, one starts the computation (for example the function `dynamicRank`) with a call to the function `allCases`. This initiates the computation. The process may then reach a point where a split is needed (the question $a = 1$ in the rank algorithm). At this point it picks one branch ($a \neq 1$) saving the other possibility onto a list ($[a = 1]$). Execution continues until the process is complete (and we have the result 2 for the case $a \neq 1$). The next stage takes the first condition on the list ($a = 1$) and re-evaluates the function (`dynamicRank`) using those conditions, possibly generating more list entries. When this list has been exhausted, a complete set of solutions has been found. This mechanism walks every possible path in the splitting tree, from the root node down to each leaf. This mechanism therefore evaluates the first part (up to a splitting point) of the function redundantly (in fact, if n answers are produced, the execution up to the first splitting point will have been repeated n times).

There are two possible solutions to this problem:

- to save the initial part preceding the split, to choose one branch, and re-instantiate the initial part in order to get the other branch when the first one is complete.
- to start two independent threads of computation at a splitting point,

The first of these may be implemented as a continuation save and restart, and the second by using a primitive similar to fork. Note that for the program to be useful, any independent threads must be re-synchronised and the results combined. This makes the two roughly equivalent, and gives us a way of describing fork as a continuation saving device.

Naturally, for any mechanism to be judged useful, the expense involved in either taking a continuation or forking a process should be less than the expense of re-evaluating the program up to the point at which the split occurs.

3.2 A tool: continuations

The language Scheme [R4R] provides continuations as first class objects. Continuations are useful for implementing a wide variety of advanced control constructs, including non-local exits, backtracking, and coroutines.

Whenever an expression is evaluated there is a *continuation* which is waiting for the result of the expression. The continuation represents an entire (default) future for the computation and may be represented as a snapshot of the current state of the program. A language which provides continuations as first-class objects allows a continuation to be saved, and later re-invoked. This re-invocation puts the computation back to the point where the continuation was saved, and additionally passes a result back to the continuation, in place of the result of the original expression.

For dynamic evaluation, the points we would mark are equalities in splitting points. In this case there are two possible ways to continue the computation, one for each branch of the tree. We therefore save the continuation of the equality test in a global table, returning false as the result of the equality test. After execution of this side of the splitting tree has completed, we can invoke this continuation with a true argument to force execution along the other path of the splitting tree. This serialises the computation as a sequence of disjoint paths in the splitting tree — there is no redundant execution of code when continuations are used.

The Scheme primitive which provides this snapshot is `call-with-current-continuation`, abbreviated to `call/cc`. This is a Scheme primitive which passes an object representing the continuation of the `call/cc` expression to the argument of the `call/cc`. The argument should be a function that takes one parameter. The following illustrates the use of `call/cc`:

```
(define *the-cont* nil)      ;; continuation holder
(define *the-val* nil)       ;; value for continuation

;; Equality. If either argument is a symbol, we save
;; the current continuation, and return false. If
;; neither is a symbol, we call =  as usual. In Scheme
;; false is #f, true is #t.

(define (my-equal a b)
   (if (or (symbol? a) (symbol? b))
       (call/cc
          (lambda (cont)
             ;; save the continuation,
             (set! *the-cont* cont)
             ;; and a value for it
             (set! *the-val*  #t)
             #f))
       (= a b)))

;; if x is zero, then return good, bad otherwise.
(define (test x)
   (if (my-equal x 0) "good" "bad"))


> (test 3)                ;; simple case, 3 <> 0
value is: "bad"

> (test 'x)               ;; saves a continuation
value is: "bad"           ;; and returns as if false

;; restart the computation from my-equal with the
;; value it saved (#t in this case)
> (*the-cont* *the-val*)
value is: "good"          ;; we get this result
```

The above example shows the basic idea behind continuations, as used in dynamic evaluation — an equality test can (by definition) only return true or false, but we use a continuation in order to be able to back-track to the point of the test, so that we can follow both possible execution paths. However, we must identify cases which are provably true or provably false, which in the case of a constructible closure of a field is a significant task.

Note that a continuation saves dynamic information, but not global variables — clearly some state has to be left alone when restoring a continuation, because without that a program would have no way of discovering whether it was continuing from a re-invoked continuation, or from normal execution.

For the purposes of dynamic evaluation, it is possible to simulate continuations in C using the `fork` function. This is available under UNIX-style operating systems — in this case a continuation is really a whole separate process which is allowed to compute up to the end of the `allCases` function, and then return its result to its caller.

3.3 Making continuations available within Axiom

A$^\sharp$ [W1] is a part of version 2 of the Axiom symbolic algebra package. The language provides a large number of relatively novel constructs, including first-class types, functions and iterators plus backward compatibility with the previous Axiom library language [JS]. As well as producing code for the Axiom system, it may also generate stand-alone programs in either Lisp or C. The dialect of Lisp that the compiler produces is heavily abstracted using macros, and has been designed in such a way that the macros may be modified for a Scheme system.

In order for A$^\sharp$ to produce both C and Lisp code, it first produces an intermediate representation called FOAM (for more details of the compiler's design, see [W2]). FOAM is a very small language which can be mapped to a particular target language. Both C and Lisp host languages have been implemented, but in practice the majority of imperative computer languages contain a FOAM-like subset. This makes the compiler retargetable to other host languages, which may provide other control mechanisms.

A$^\sharp$ also includes a foreign function interface which can call functions from the hosting language, and allows one to build abstractions over the imported functions. For example, the `call/cc` function and the continuations it produces may be turned into a type by importing `call/cc` into the compiler namespace, and wrapping a type around it. Thus we declare a type with the following signature:

```
Continuation(T: Type): with {
    callWithCC: (% -> T) -> T;
    apply:      (%, T)   -> Exit;
}
```

This states that `Continuation` is a function, which when applied to a type (called `T`), returns a new type which has two operations: `apply` and `callWithCC`. The `apply` operation invokes a continuation. The type `Exit` indicates that the call never returns — in this case execution continues at the point of the corresponding `callWithCC` call. Calling this function `apply` indicates that using a continuation in a function position will call this function — this is in order to mimic the Scheme syntax. In order to implement `callWithCC`, we have to write a small piece of Scheme which can call back the function passed to it. The function is an Axiom-level function, not a Scheme function, so we have to call it via a macro (`CCall`) from the FOAM package.

```
(define (schemeCallWithCC clos)
      (call/cc (lambda (c)
               (CCall clos c))))
```

We then incorporate this code into the definition of Continuation as follows (continuing from the declaration above):

```
Continuation(T: Type): with {
    callWithCC: (% -> T) -> T;
    apply:      (%, T)   -> Exit;
}
== add {
    import { schemeCallWithCC: (% -> T) -> T }
           from Foreign Lisp;

    callWithCC(fn: % -> T): T ==
           schemeCallWithCC fn;

    ...
}
```

The import statement asserts that the function schemeCallWithCC exists, has a particular type, and should be called as a Lisp function. Note that from the compiler's point of view the Scheme language is identical to Lisp.

These definitions now allow us to use continuations as first class objects within $A^\sharp$.

We also implement a slightly modified version of continuations in terms of fork — in this case, we import functions from C, and have to do more wrapping in order to achieve the desired semantics, but the declaration above is identical, and so the code using the Continuation type does not need to be changed.

The relative costs of the two implementations are hard to compare — in the Scheme case, the act of taking a continuation is not too high unless there is a large amount of dynamic data. However, the cost of running any Scheme code is often much greater than that of running the same code under C. This is partially due to the implementation of loops in Scheme, which had to be hand crafted, and also because Scheme is a dynamically typed language, and therefore spends much time checking types. C, however runs code very fast, but a call to fork has to be carefully guarded, as this may double the amount of memory taken by the program. This doubling is worst case, as many operating systems only replicate information after a process has written to a page, which implies that we only need copy the working set of pages in the program. The overhead can also be reduced by ensuring that no more than a fixed number of processes are active at a time — this ensures that the amount of space needed is proportional to the depth of the splitting tree, rather than the maximum width, at the worst case.

3.4 Using continuations with an algebra library

As we have shown above, continuations form a useful basic mechanism for implementing dynamic evaluation. Dynamic evaluation itself can be applied to many problems, which typically need a large amount of non-trivial algebra code. This code is typically written in a high level language, whose hosting platform is not able to handle the creation of continuations. Conversely, languages that provide continuations do not have a rich enough algebra library or type system to implement some of the applications. Naturally, these could be added, but this is a significant amount of work. The $A^\sharp$ compiler is used to bridge this gap.

As described above, there is the old implementation of dynamic evaluation [Du], which has been translated into $A^\sharp$ language. (not a major change, as the languages are very similar). The principal user of this package, the dynamic constructible closure domain, has also been translated.

These programs are strongly structured into categories, domains and packages, and so it has been very easy to adapt them to the use of the new tools. In particular there are only two major changes:

- the function allCases in the control package,
- the equality in the dynamic constructible closure domain (and that only for the case where a split appears).

These two parts have been rewritten to use the continuation model of evaluation.

The old implementation of dynamic evaluation uses mutable variables in the dynamic constructible closure constructor in order to store the *current case* and the *list of next cases*. These variables are changed at a split point and allCases has access to these variables so as to control execution.

The mutable variable for the list of next cases disappears with the introduction of continuations, as the continuation effectively holds the necessary case information. This list of cases variable is therefore substituted with a variable that saves a list of outstanding continuations.

This rewriting allows one to use programs developed using the compiler to make use of dynamic evaluation — for example a program using the Matrix domain formed over a dynamic field may be used unchanged in both a Scheme and C based environment.

4 CONCLUSIONS AND FURTHER WORK

We have presented here two different solutions for the control problem in dynamic evaluation. In fact this control problem appears in a more general context: the use of parallelism from computer algebra systems.

We have seen how $A^\sharp$ is able to incorporate tools such as continuations and fork. This mechanism is very powerful, and while foreign function interfaces are becoming relatively common in other languages, the compiler allows a much greater degree of control over the definition and use of the imported functions.

The problem of redundant computation has been avoided in these implementations of dynamic evaluation, and the use of continuations is a very elegant mechanism for solving this form of control problem. Naturally, more work is needed in both the C and Scheme versions to ensure that they retain this elegance, and execute without excessive overhead. As stated before, the original code was only lightly modified, but one can imagine some changes to make it better adapted to the continuations model.

The dynamic evaluation code itself can be further improved — the gcd algorithm used as the basis of the equality test is a generic implementation, and could be made more specific to the data structures used in dynamic evaluation.

References

[R4R] W. Clinger, J. Rees, eds. — *Revised[4] Report on the Algorithmic Language Scheme.* ACM LISP Pointers IV, 3 (July-September 1991).

[CJ] R. Corless, D. Jeffrey — *Well It Isn't Quite That Simple!* ACM Sigsam Bulletin, Number 26 p. 2–6 (1992).

[DDD] J. Della Dora, C. Dicrescenzo, D. Duval — *About a New Method for Computing in Algebraic Number Fields.* Eurocal'85, vol.2, Springer Lecture Notes in Computer Science 204, ed. G. Goos, J. Hartmanis, p. 289–290 (1985).

[DD] C. Dicrescenzo, D. Duval — *Algebraic Extensions and Algebraic Closure in Scratchpad.* Symbolic and Algebraic Computation, Springer Lecture Notes in Computer Science 358, ed. P. Gianni, p. 440–446 (1989).

[Du] D. Duval — *Evaluation dynamique et clôture algébrique en Axiom.* J. of Pure and Applied Algebra, to appear.

[DGV] D. Duval, L. González-Vega — *Dynamic Evaluation and Real Closure.* To appear in Mathematics and Computers (transactions of IMACS Conference, June 1993).

[DR] D. Duval, J.-C. Reynaud — *Sketches and Computation (Part I): Basic Definitions and Static Evaluation* and *(Part II): Dynamic Evaluation and Applications.* Mathematical Structures in Computer Science, **4** p. 185-238 and 239-271. Cambridge University Press (1994)

[Go] T. Gómez-Díaz — *Quelques applications de l'évaluation dynamique.* Thesis, Université de Limoges (1994). Available from Atelier National de Reproduction des Thèses, Université de Grenoble 2.

[JS] R. D. Jenks, R. S. Sutor — *Axiom, The Scientific Computation System.* NAG, Springer-Verlag (1992).

[La] J. M. Lang — *Private communication.* Waterloo Maple Software (1995).

[Si] W. Y. Sit — *An algorithm for solving parametric linear systems.* J. Symbolic Computation **13** p. 353–394 (1992).

[W1] S. M. Watt, P. A. Broadbery, S. S. Dooley, P. Iglio, S. C. Morrison, J. M. Steinbach, R. S. Sutor — *Axiom library compiler user guide.* NAG Ltd, 1994.

[W2] S. M. Watt, P. A. Broadbery, S. S. Dooley, P. Iglio, S. C. Morrison, J. M. Steinbach, R. S. Sutor — *A first report on the $A^\sharp$ compiler.* Proceedings ISSAC'94, ACM Press, New York 1994, p. 25–31.