Improving the Efficiency of Non-Deterministic Computations

Sergio Antoy[1] Pascual Julián Iranzo[2,3] Bart Massey[1]

Computer Science Department
Portland State University
Portland, Oregon
{antoy,pjulian,bart}@cs.pdx.edu

Abstract. Non-deterministic computations greatly enhance the expressive power of functional logic programs, but are often computationally expensive. We analyze two programming techniques that improve the time and memory efficiency of some non-deterministic computations. These techniques rely on the introduction of a new symbol into the signature of a program. In one technique this symbol is a polymorphic defined operation, in the other an overloaded constructor. Our programming techniques may save execution time by reducing the number of steps of a computation, as well as memory occupation, by reducing the number of terms constructed by a computation. We show how to apply our techniques using some examples, and informally reason about their effects.

1 Introduction

Functional logic programming studies the design and implementation of programming languages that integrate both functional programming and logic programming into a homogeneous paradigm. In recent years, it has become increasingly evident that non-determinism is an essential feature of these integrated languages. Non-determinism is a cornerstone of logic programming. It allows problem solving using programs that are textually shorter, easier to understand and maintain, and more declarative than their deterministic counterparts.

In a functional logic programming language, non-deterministic computations are modeled by the defined operations of a constructor-based left linear conditional rewrite system. With respect to logic computations, which are based on resolution, functional logic computations are nested and therefore can be lazily executed. The combination of these features makes functional logic languages both more expressive than functional languages and more efficient than traditional logic languages.

[1] Supported in part by the NSF grant INT-9981317.
[2] Supported by the Spanish Knowledge Society Foundation.
[3] Permanent Address: Univ. Castilla La-Mancha, Spain, pjulian@inf-cr.uclm.es.

A typical approach to the definition of non-deterministic computations is by means of the defined operations of a constructor based non-confluent rewrite system. The following emblematic example [8, Ex. 2] defines an operation, `coin`, that non-deterministically returns either zero or one. Natural numbers, represented in Peano notation, are defined by the datatype (or sort) `nat`.

```
datatype nat = 0 | s nat
coin = 0
coin = s 0
```

Rewrite systems with operations such as `coin` are non-confluent. A computation in these rewrite systems may have distinct normal forms and/or non terminate. To understand non-determinism in the context of a computation, consider the following operations:

```
add 0 Y = Y
add (s X) Y = s (add X Y)
positive 0 = false
positive (s _) = true
```

The evaluation of a term such as `positive (add coin 0)` requires the evaluation of subterm `coin`. This subterm has two replacements, i.e., `0` and `s 0`. Each replacement leads to a different final result. The choice between these two replacements is non-deterministic. Assuming that non-determinism is appropriately used in the program where the evaluation occurs, there is no feasible means of deciding which replacement should be chosen at the time `coin` is evaluated. Therefore, evaluation under both replacements must be considered.

To ensure operational completeness, all the possible replacements of a non-deterministic computation must be executed fairly. In fact, if one replacement is executed only after the computation of another replacement is completed, the second replacement will never be executed if the computation of the first replacement does not terminate. Thus, continuing with our example, to compute `positive (add coin 0)` one must compute fairly and independently both `positive (add 0 0)` and `positive (add (s 0) 0)`.

This approach, which we refer to as *fair independent computations*, captures the intended semantics, but clearly it is computationally costly. In some situations the cost of fair independent computations might be avoided. For example, define a "bigger" variant of `coin`:

```
bigger = s 0
bigger = s (s 0)
```

and consider again the previous example, but invoking `bigger` instead of `coin`. The evaluation of `positive (add bigger 0)`, which will be shown in its entirety later, may be carried on, as in the previous example, using fair independent computations. However, this is not necessary. The computation has a single result that may be obtained using only deterministic choices. Avoiding fair independent computations saves execution time, memory occupation, and the duplication of the result.

In this paper, we discuss two programming techniques that have been considered within a project aiming at the implementation of a back-end for a wide class of functional logic languages [7]. In some cases, these techniques have the potential to offer substantial improvements. In other cases, they tend to consume slightly more memory, but without a substantial slowdown. We are currently working to assess whether either or both techniques should be deployed in the back-end: this document is a report of our preliminary findings.

Section 2 discusses the usefulness of non-deterministic computations in functional logic programs and how they are related to our work. Section 3 justifies our overall approach to measuring the efficiency of a computation. Section 4 presents the programming techniques that are the focus of our work. In some cases, these techniques reduce the computing time and/or the memory consumption attributed to non-deterministic computations. Section 5 discusses, both theoretically and experimentally, the effects of our techniques on some examples. Section 6 contains our conclusions.

2 Non-Determinism

Non-determinism is an essential feature of logic programming, perhaps the single most important reason for its acceptance and success. Some early proposals of functional logic programming languages neglected this aspect. Programs in these early languages were modeled by weakly orthogonal rewrite systems. In these languages, the results of non-deterministic computations are obtained by instantiating the arguments of a predicate. A serious drawback of this situation is that a non-deterministic computation cannot be functionally nested in another computation. The lazy evaluation of non-deterministic computations becomes impossible and the efficiency of a program may incur severe losses.

More recently [4, 8], non-determinism in functional logic programming has been described using the operations of a non-confluent Term Rewriting System (TRS). These operations are quite expressive, in that they allow a programmer to translate problems into programs with a minimal effort. For example, the following operation computes a non-empty regular expression over an alphabet of symbols. Each non-empty regular expression is obtained by appropriate non-deterministic choices of a computation.

```
regexp X = X
regexp X = "(" ++ regexp X ++ ")"
regexp X = regexp X ++ regexp X
regexp X = regexp X ++ "*"
regexp X = regexp X ++ "|" ++ regexp X
```

The definition of operation `regexp` closely resembles the formal definition of *regular expression*, e.g., as found in [1, p. 94]. This transparency in semantics can be very convenient for the programmer. For example, to recognize whether a string s denotes a well-formed non-empty regular expression over some alphabet a, it suffices to evaluate $\texttt{regexp a} = s$,

Non-deterministic operations support a terse programming style, but may impose a stiff penalty on execution performance. In practice, several computations originating from a non-deterministic choice may have to be executed fairly. Therefore, techniques to improve the efficiency of non-deterministic computations, in particular to limit the number of fair independent computations that originate from a non-deterministic choice, are quite useful. The overall goal of this paper is the study of two techniques for this purpose.

3 Cost Analysis

The most common approach to analyzing the efficiency of a program is measuring its execution time and memory occupation. We measure the execution time of benchmark programs by means of primitives available in our run-time environment. In addition to measuring the amount of memory used during the execution of a program by means of primitives, we compute the amount of memory used by simple benchmarks using a theoretical technique. In this section, we discuss this theoretical approach to memory usage measurement.

Our starting point is the *number of applications* cost criterion defined in earlier work on partial evaluation [2, Def. 2]. This criterion intends to measure the storage that must be allocated for executing a computation. We adapt the criterion to the behavior of our run-time environment. We also address the problems of non-deterministic steps. We show that non-determinism, which is not considered in the earlier definition, adds an interesting twist to the situation.

The following definitions formalize our adaptation of the cost criterion "number of applications."

Definition 1 (number of applications). *We denote by $\mathcal{A}$ an overloaded function, called the* number of applications*, as follows:*

- *If t is a term, $\mathcal{A}(t) = \Sigma_{p \in \mathcal{P}(t)}\ (arity(root(t|_p)) + 1)$, where $\mathcal{P}(u)$ is the set of positions of non variable symbols of arity greater than zero in any term u, $root(u)$ is the root symbol of any term u, and $arity(f)$ is the arity of any symbol f.*
- *If $R \equiv l \to r$ is a rewrite rule[1], we define $\mathcal{A}(R) = \mathcal{A}(r)$.*
- *If $C \equiv t \to_{R_1} t_1 \to_{R_2} \cdots \to_{R_n} t_n$ is a computation of a term t to a constructor term t_n, we define $\mathcal{A}(C) = \mathcal{A}(t_n) + \Sigma_{i=1}^{n} \mathcal{A}(R_i)$.*

The number of applications of a term t is the total number of occurrences of n-ary symbols, with $n > 0$, in t, plus their arities. In our run-time environment (and, we believe, in many lazy language implementations) it is appropriate to consider both defined operation and constructor symbols occurring in the term. The number of applications of a computation accounts for the number of applications of each step *and* the number of applications of the result. In a run-time environment that supports in-place updates, it would *not* be necessary to account for the number of applications of the result. We use the implementation of narrowing in Prolog described in [6]. We have verified on several simple programs that this implementation allocates memory in accordance to our definition.

Earlier work [2] shows that the number of reduction steps of a computation is weakly correlated to its execution time. Nevertheless, we count the *number of steps* [2, Def. 1] of a computation, since the computation of all cost criteria in this work is based on steps.

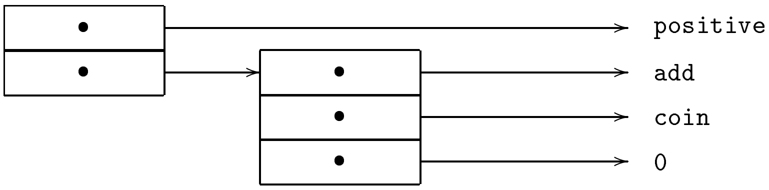
[1] Without loss of generality, we consider only unconditional rules [5].

Most cost analysis techniques in the literature are proposed for deterministic computations. Non-deterministic computations in functional logic programming are a relatively newer concept, and introduce significant theoretical and practical complications.

To ensure operational completeness, non-deterministic computations must be executed fairly. A consequence of this condition is that when a program outputs a result (derived, for example, by using the first alternative in a non-deterministic computation) the time and space resources consumed from the beginning of the execution to the time of the output may not be a correct indication of the cost of computing that result. The reason is that the measured values may include resources spent to partially compute other results that have not yet been output. The extent of these computations, and consequently a quantification of the resources spent by these computations, are generally difficult to estimate. A better approach would be to measure the resources needed to compute all the results of a non-deterministic computation, but this is impossible in practice for computations over an infinite search space, such as the computation of the `regexp` operation presented earlier.

To deal with these difficulties, which to date have no universally accepted solution, we consider only simple examples. In particular, we reason with natural numbers in Peano notation. This decision is quite convenient for explanation purposes. In practice, one technique that we will discuss in the next section may not be well suited for builtin types, such as binary integers.

We informally reason about the number of steps of a computation and the memory occupied to represent terms. In typical implementations of rewrite systems and functional logic programs, terms are represented by dynamic (linked) data structures. In these structures, each occurrence of a symbol of arity n greater than zero takes $n+1$ units of dynamic (heap) memory. Nullary symbols are allocated in global (static) memory. Variables are local to rules or clauses and are allocated in local (stack) memory. The following picture informally shows the 5 units of memory allocated to represent the term `positive` (`add coin 0`). The symbols in the right column are allocated in global memory. They are not not specifically allocated to represent any term, but are shared by all terms. The number of arguments of an occurrence of a symbol is not a part of the term because in our run-time environment, Prolog, symbols are fully applied.



The previous analysis [2] and the adaptation introduced in this section are limited to deterministic computations. The extension to non-deterministic computations would be a non-trivial task. We believe that our less formal discussion is appropriate for our goals and easier to grasp than a more rigorous approach.

To understand why the treatment of non-deterministic computations is more complicated, consider the evaluation of $t =$ `s coin`. This term has two normal forms, `s 0` and `s (s 0)`. The root symbol of each normal form can be traced back to the root symbol t. This shows that fair independent computations may have to duplicate the portion of a term above a redex with distinct reducts. Hence, even in run-time environments that support in-place updates, the cost of a step may depend on its context. This consideration further supports the appropriateness of including the number of applications of the result of a computation in the number of applications of the computation itself.

4 Programming Techniques

We attempt to improve the efficiency of non-deterministic computations by avoiding the duplication of both reduction steps and term representations that occur within fair independent computations. We use two programming techniques that achieve some improvements in some cases. In other words, our approach is a guideline for the programmer, i.e., a suggestion on how to code certain problems into programs. However, we envision that an optimizing compiler or a similar specialized tool could automatically transform a program in the same way. In fact, several experimental variations of the second technique have been automatically implemented in our current system [7].

Both of our programming techniques are based on the introduction of a new symbol into the signature of the TRS modeling a functional logic program. One technique regards the new symbol as a polymorphic defined operation, the other as an overloaded constructor. The first approach is not new [8]: our contribution in this case is limited to recognition that there are potential benefits of this technique in the context of modern FLP implementation, and the quantification of these benefits. The new symbol that we introduce, is denoted by the infix operator "`!`", and read as *alternative*.

4.1 The *Alternative* Operator

In the first programming technique, the *alternative* operation is defined by the rules:

```
X ! Y = X
X ! Y = Y
```

An operation with these rules is called *alt* and denoted by "//" in the work of González-Moreno *et. al.* [8,9]. We could regard this symbol as left associative or overload it for arbitrarily long argument lists: in our examples the symbol is always binary, so the difference is irrelevant.

The *alternative* operation allows us to give a different, though equivalent, definition of the operation `bigger` presented earlier.

```
bigger = s (0 ! s 0)
```

The significant difference is that a common portion of the right-hand sides of the two rewrite rules of the original definition of `bigger` has been "factored". This new definition can be directly coded by the programmer or it could be automatically obtained from the original definition by an optimizing compiler or other specialized tool.

The advantage of this definition of `bigger` with respect to the original one is that if only the factored portion of the two alternative right-hand sides of the rewrite rules of `bigger` is needed by a context, no fair independent computations are created by a *needed* strategy [4]. A single deterministic computation suffices in this case. This is exactly what the composition of `positive` and `add` requires, as shown by the following derivation:

```
positive (add bigger 0)
  → positive (add (s (0 ! s 0)) 0)
  → positive (s (add (0 ! s 0)) 0)
  → true
```

Two computations have been replaced by a single computation of the same length. In cases where factoring the right-hand sides of two rewrite rules does not eliminate the need of fair independent computations, the run-time cost of the factorization is a single additional rewrite step. For realistic programs, this cost is negligible. Hence, the factorization of right-hand sides is a worthwhile potential improvement. In the best case, it saves computing time and/or storage for representing terms. In the worst case, it costs one extra step and very little additional memory.

4.2 The *Alternative* Constructor

Our second approach is to consider the *alternative* symbol as a constructor. Since functional logic programs are generally strongly typed, they are modeled by many-sorted rewrite systems. This condition requires overloading the symbol "`!`" for each sort in which it is introduced.

The consequences of introducing such an overloaded constructor are interesting. For example, the new definition of `bigger` is like the previous one

```
bigger = s (0 ! s 0)
```

except that the right-hand side is an irreducible (constructor) term. In this example, new constructor terms should be interpreted as non-deterministic choices in sets of terms. The right-hand side of the definition of `bigger` is interpreted as an element in the set {`s 0`, `s (s 0)`}. In general, we think that extending builtin types, (such as the integers or booleans) or well-known types (such as the naturals) is inappropriate. Extending a sort with new constructor symbols radically changes the nature of that sort. The interpretation of the new terms of an extended sort may be difficult. Nevertheless, we do it here for the sort `nat` for its immediateness and to ease the comparison with the examples presented for the first technique.

The introduction of a new constructor symbol makes some formerly well-defined operations incompletely defined. It is relatively easy to correct this problem in the class of the overlapping inductively sequential TRSs [4]. Every operation in this class has a definitional tree [3]. The necessary additional rules may be determined from this tree. For example, consider the operation that halves a natural:

```
half 0 = 0
half (s 0) = 0
```

```
half (s (s X)) = s (half X)
```

If the type natural is extended by an *alternative* constructor, the following additional rewrite rules complete the definition of `half`:

```
half (X ! Y) = (half X) ! (half Y)
half (s (X ! Y)) = (half (s X)) ! (half (s Y))
```

In general, a new rewrite rule is needed for each branch of the tree. If π is the pattern of a branch and p is the inductive position of π, then the required rewrite rule is:

$$\pi[X \ ! \ Y]_p \ \rightarrow \ \pi[X]_p \ ! \ \pi[Y]_p$$

The advantages of factoring right-hand sides when the *alternative* symbol is an operation are preserved by additional rewrite rules of this kind when the *alternative* symbol is a constructor as well. However, when one of the new rewrite rules is applied, additional storage is required for the representation of terms. Referring to the example under discussion, the representation of `half (s X) ! half (s Y)` takes more storage—exactly three units for the top occurrence of the *alternative* constructor—than the representations of `half (s X)` and `half (s Y)` combined.

In general, it is not possible to say whether defining the *alternative* symbol as a constructor will increase or decrease the storage used to represent the terms of a computation. In some case, the *alternative* symbol allows a more compact representation of some results of a computation. For example, consider the evaluation of:

```
add (s 0) coin
  → s (add 0 coin)
  → s (coin)
  → s (0 ! s 0)
```

If the *alternative* symbol were not a constructor, the last term of the above computation would create two fair independent computations. To complete these computations both additional steps would be executed and additional storage would be needed for the execution of these steps.

A consequence of defining the *alternative* symbol as a constructor is that several alternative normal forms are represented by a single term. Therefore, it is likely inappropriate to adopt this programming technique to code problems where only a small fraction of the potentially computed values of a computation are actually needed.

5 Examples

In order to reason about the advantages and disadvantages of our techniques, we analyze a few computations using the cost criterion discussed in Section 3. As noted there, the theory that we use has previously been studied only for deterministic computations. In our simple examples, where the computation space is finite, we adapt it to non-deterministic computations as follows.

Consider a complete independent computation for each non-deterministic step, and two independent computations that differ for a non-deterministic replacement. In our implementation [7], some fair independent computations may share steps and terms. In these cases, our theory would predict that the storage allocated for all the computations of a term is higher than it is actually is.

We consider the computations of `positive (add bigger 0)` with and without using our first technique. In the following tables, each line represents a step of a computation. We measure both the number of steps and the number of applications of a computation. The columns of a table respectively show the step counter, the rewrite rule applied in the step, and the number of applications of the step. The result does not contribute to the number of applications of the computation because it is a constant (a nullary symbol).

Tables 1–3 show that when `bigger` is defined by two rewrite rules, the resources spent to compute `positive (add bigger 0)` are 6 steps and 16 units of memory. By contrast, our first technique cuts the number of steps in half and reduces the memory consumption by 25%. These exact savings are also obtained with our second technique.

Step	*Rule*	A
1	`bigger → s 0`	2
2	`add (s X) Y → s (add X Y)`	5
3	`positive (s _) → true`	0

Table 1. Computation when `bigger` non-deterministically rewrites to `s 0`. Total resources: steps 3, memory units 7.

Step	*Rule*	A
1	`bigger → s (s 0)`	4
2	`add (s X) Y → s (add X Y)`	5
3	`positive (s _) → true`	0

Table 2. Computation when `bigger` non-deterministically rewrites to `s (s 0)`. Total resources: steps 3, memory units 9.

Step	*Rule*	A
1	`bigger → s (0 ! s 0)`	7
2	`add (s X) Y → s (add X Y)`	5
3	`positive (s _) → true`	0

Table 3. Computation when `bigger` rewrites to `s (0 ! s 0)` and "!" is an operation. Total resources: steps 3, memory units 12.

A similar analysis for the computation of `half bigger` shows that when `bigger` is defined by two rules, the resources spent by all the computations are 5 steps and 12 units of memory. By contrast, using our second technique (i.e., when `bigger` rewrites to `s (0 ! s 0)` and "`!`" is a constructor) the resources used are 5 steps and 27 units of memory: there is a 108% increase in memory consumption. The first technique uses 6 steps and 19 units of memory, an increase of 58%.

On these examples, the implementation of [6] allocates memory according to our theoretical model. However, the above examples are too small and artificial for understanding the effects of our programming techniques in practice. Also, our theoretical analysis is difficult to apply to programs that make more than a few steps. For this reason, we benchmark both the memory consumption and the execution times of larger programs. Our programming language is Curry [10]. The compiler is PAKCS, which transforms Curry source code into Prolog for execution.

The first program that we benchmark is an implementation of the game of 24. Some of us first encountered this problem at a meeting of the Portland Extreme

Programming User Group on June 5, 2001: it is inspired by a commercial game intended to develop mathematical skills in middle school students. The game is played as follows: given four 1-digit positive integers find an arithmetic expression in which each digit occurs exactly once and that evaluates to 24. A number can be divided only by its factors. For example, a solution for the instance $[2, 3, 6, 8]$ is $(2+8)*3-6$. There are 25 other distinct solutions of this instance (including commutatively and associatively equivalent solutions), including $3*(2+8)-6$ and $6*3+(8-2)$.

The program for this problem, shown in its entirety in the Appendix, proceeds via straightforward generate-and-test. Table 4 shows the CPU time (on a Sun SPARCStation running Solaris) spent for computing all the solutions of a few problems, and the global and local stack allocations for the computation reported by the Curry primitive `evalSpace`. The first group of data is for a version of the program that does not use our techniques. The second group is for a program that uses our first technique, i.e., the alternative symbol is a defined operation. The second technique is not appropriate for this problem. The runtime measures are nearly identical over several executions. The memory measures are constant for every execution.

The data shows that our technique consumes slightly more memory, but speeds the execution of the program by 44% on average. The speedups for various problems range from 27% to 58%. (The speedup achieved by our technique is computed by $(t_1 - t_2)/t_1$ where t_1 and t_2 are the averages of the execution times of the programs not using and using the technique respectively. This speedup indicates the percentage of execution time saved by the technique.)

Table 4. Runtime (msec.) and memory usage (bytes) for "24" instances.

	regular program			first technique			
`problem`	Runtime	G. stack	L. stack	Runtime	G. stack	L. stack	Speedup
[2,3,6,8]	66	2596	932	48	2800	1100	27%
[2,3,4,9]	94	2632	860	52	2836	972	45%
[3,4,5,8]	65	2476	868	27	2680	1036	58%
[1,2,6,8]	64	2812	868	36	2816	980	44%
[4,6,8,9]	38	2416	832	19	2620	1000	50%
`Average`	65	2586	872	36	2750	1017	44%

Our second example is a parser that takes a string representing a parenthesized arithmetic expression and returns a parse tree of the input. Our implementation is simplified to the extreme and serves only as a proof of concept. The abstract syntax generated by the parser is defined by the type:

```
data AST = Num String | Bin Char AST AST
```

For example, on input `"1+(2-3)"` the parser generates

Bin`'+'`(*Num*`"1"`)(*Bin*`'-'`(*Num*`"2"`)(*Num*`"3"`)).

Replacing the argument of *Num* with an integer and the *Bin Char* combination with a token would be more appropriate, but it would add to the program details that are irrelevant to our analysis. The language recognized by the parser is generated by the following grammar:

$$
\begin{aligned}
expression ::= &\ term\ \texttt{'+'}\ expression \\
| &\ term\ \texttt{'-'}\ expression \\
| &\ term \\
term ::= &\ \texttt{'('}\ expression\ \texttt{')'} \\
| &\ digits
\end{aligned}
$$

Sequences of *digits* are recognized by a scanner.

The parser is implemented using two defined operations: *expression* and *term*. The type of both operations is `[Char]` $\to$ `[Char]` $\to$ `AST`. For all strings s and r, *expression* $s\ r$ evaluates to a if and only if there exists a string u such that $s = u\,r$ and a is the parse tree of u. Operation *term* is analogous. For example, *term* `"1+(2-3)"` `"+(2-3)"` evaluates to *Num*`"1"`. To parse an entire string, operation *expression* is initially called with its second argument equal to the empty string. In recursive calls, the second argument is a free variable.

Table 5 shows execution time and memory usage on a 233MHz Pentium PC running Linux. In this program, too, the data show that our first technique consumes more memory, but substantially cuts the execution time to parse certain strings. The speedup is highly dependent on the structure of the input string.

Table 5. Runtime (msec.) and memory usage (bytes) while parsing.

	regular program			first technique			
`input`	Runtm	G. stck	L. stck	Runtm	G. stck	L. stck	Speedup
"1+1+1+1+1+1+1+1"	10	12528	736	10	16152	1012	0%
"((((((0))))))"	1440	2676	8	5	6992	568	99%
"5-((2+1)+3+(5-4))"	60	10468	528	10	15240	1144	83%
`Average`	500	8557	424	8	12795	908	98%

6 Conclusions

Non-deterministic computations are an essential feature of functional logic programming languages. Often, a non-deterministic computation is implemented as a set of fair independent computations whose results are used, and possibly discarded, by a context. A non-deterministic computation can be costly to execute: any reasonable attempt to improve its efficiency is worthwhile.

In this paper, we have proposed two simple programming techniques intended to improve the efficiency of certain non-deterministic computations. Both techniques are based on the introduction of a new symbol, called *alternative*, into the signature of a program. In one technique, the *alternative* symbol is a polymorphic defined operation. In the other technique, the *alternative* symbol is an overloaded constructor. This symbol allows a program to factor a common portion of the non-deterministic replacements of a redex.

Either technique may improve the efficiency of a computation by reducing the number of computation steps or the memory used in representing terms. These savings are obtained in two situations. For both techniques, savings are obtained when fair independent computations are avoided because only the factored portion of non-deterministic replacements is needed. For the second technique, savings are obtained when distinct non-deterministic results are more compactly represented by sharing a common factor. In some cases, the improvements offered by these techniques are substantial. In all cases, the cost of applying the first technique is small. There are cases in which the application of the second technique may actually result in computations that consume more memory.

We have discussed how to apply our techniques, and we have quantified the effects of the application of these techniques in simple examples. Our techniques are applicable to programs coded in many existing or proposed functional logic programming languages. Our techniques can be directly adopted by programmers or can be introduced into a program automatically at compile time.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1985.
2. E. Albert, S. Antoy, and G. Vidal. Measuring the effectiveness of partial evaluation in functional logic languages. In *Proc. of 10th Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'2000)*, pages 103–124. Springer LNCS 2042, 2001.
3. S. Antoy. Definitional trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.
4. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.
5. S. Antoy. Constructor-based conditional narrowing. In *Proc. of 3rd Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'01).* Springer LNCS, 2001. To appear.
6. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into prolog. In *Proc. of the 3rd International Workshop on Frontiers of Combining Systems (FroCoS 2000)*, pages 171–185, Nancy, France, March 2000. Springer LNCS 1794.
7. S. Antoy, M. Hanus, B. Massey, and F. Steiner. An implementation of narrowing strategies. In *Proc. of 3rd Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'01).* Springer LNCS, 2001. To appear.
8. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. ESOP'96*, pages 156–172. Springer LNCS 1058, 1996.
9. J.C. González-Moreno, F.J. López-Fraguas, M.T. Hortalá-González, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *The Journal of Logic Programming*, 40:47–87, 1999.
10. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Version 0.71). Web document `http://www.informatik.uni-kiel.de/~mh/curry/report.html` (accessed Jul 10, 2001 04:07 UTC), 2000.

Appendix

The Game of 24

This program solves the game of 24.

```
infixr 5 +++
(+++) eval flex
[] +++ x = x
(x:xs) +++ y = x:xs +++ y

permute [] = []
permute (x:xs) | u+++v =:= permute xs = u++[x]++v where u,v free

data exp = num Int
         | add exp exp
         | mul exp exp
         | sub exp exp
         | dvv exp exp

generate [y] = num y
generate (y:y1:ys)
  | (y:y1:ys) =:= u:us+++v:vs
  = add (generate (u:us)) (generate (v:vs)) where u,us,v,vs free
generate (y:y1:ys)
  | (y:y1:ys) =:= u:us+++v:vs
  = mul (generate (u:us)) (generate (v:vs)) where u,us,v,vs free
generate (y:y1:ys)
  | (y:y1:ys) =:= u:us+++v:vs
  = sub (generate (u:us)) (generate (v:vs)) where u,us,v,vs free
generate (y:y1:ys)
  | (y:y1:ys) =:= u:us+++v:vs
  = dvv (generate (u:us)) (generate (v:vs)) where u,us,v,vs free

test (num y) = y
test (add x y) = test x + test y
test (mul x y) = test x * test y
test (sub x y) = test x - test y
test (dvv x y) = opdvv (test x) (test y)
  where opdvv x y = if y == 0 || not (x `mod` y == 0)
                    then failed else x `div` y

solve p | test x == 24 = x where x = generate (permute p)

-- example: solve [2,3,6,8]
```

The application of our technique calls for the definition of the *alternative* function and the replacement of operation *generate*.

```
infixl 0 !
x ! _ = x
_ ! y = y
```

```
generate [y] = num y
generate (y:y1:ys)
  | (y:y1:ys) =:= u:us+++v:vs
  = (add ! mul ! sub ! dvv) (generate (u:us)) (generate (v:vs))
        where u,us,v,vs free
```

The Parser

This is a parser for parenthesized arithmetic expressions.

```
--import Char

data AST = Num String | Bin Char AST AST

expression X0 X3
  | A1 =:= term X0 X1 &>
    '+':X2 =:= X1 &>
    A2 =:= expression X2 X3
  = Bin '+' A1 A2 where X1, X2, A1, A2 free
expression X0 X3
  | A1 =:= term X0 X1 &>
    '-':X2 =:= X1 &>
    A2 =:= expression X2 X3
  = Bin '-' A1 A2 where X1, X2, A1, A2 free
expression X0 X1 = term X0 X1

term X0 X2
   | X:X1 =:= takeWhile isDigit X0 &> X0 =:= X:X1 ++ X2
   = Num (X:X1)
   | X0 =:= '(':Y0
   = expression Y0 (')':X2)
     where Y0, X, X1 free

-- example: expression "1+(2-3)" ""
```

The application of our technique calls for the definition of the *alternative* function, as in the previous program, and the replacement of operation *expression* with the following code.

```
expression X0 X3
  | A1 =:= term X0 X1 &>
      ( ( OP:X2 =:= X1 &>
          OP =:= ('+'!'-') &>
          A2 =:= expression X2 X3 &>
          TREE =:= Bin OP A1 A2 )
      ! (X3 =:= X1 &> TREE =:= A1)
      )
  = TREE where OP, X1, X2, A1, A2, TREE free
```