Electronic Notes in Theoretical Computer Science 64 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume64.html 22 pages

Improving the Efficiency of Non-Deterministic Computations [1]

Sergio Antoy [a] Pascual Julián Iranzo [b] Bart Massey [a]

[a] *Computer Science Department*
Portland State University
Portland, Oregon
antoy@cs.pdx.edu, bart@cs.pdx.edu

[b] *Escuela Superior de Informática*
Universidad de Castilla–La Mancha
Ciudad Real, Spain
pjulian@inf-cr.uclm.es

Abstract

Non-deterministic computations greatly enhance the expressive power of functional logic programs, but are often computationally expensive. We analyze a programming technique that improves the time and memory efficiency of some non-deterministic computations. This technique relies on the introduction of a new symbol into the signature of a program. This symbol may be treated either as a polymorphic defined operation or as an overloaded constructor. Our programming technique may save execution time, by reducing the number of steps of a computation. The technique may also save memory, by reducing the number of terms constructed by a computation. We give some examples of the application of our technique, address its soundness and completeness, and informally reason about its impact on the efficiency of computations.

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.

[1] Supported in part by NSF grants INT-9981317 and CCR-0110496 and by the Spanish Knowledge Society Foundation, the Spanish Research Funding Agency (CICYT) TIC 2001-2705-C03-01, by Acción Integrada Hispano-Italiana HI2000-0161, and the Valencian Research Council under grant GV01-424.

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. The narrowing-based logic computations of functional logic programs 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.

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 [9, Ex. 2] defines an operation, `coin`, that non-deterministically returns either zero or one. All of our examples conform to the syntax of Curry [12] and were compiled and executed by Pakcs [11], a popular Curry compiler/interpreter. Natural numbers, represented in Peano notation, are defined by the datatype (or sort) `nat`.

```
datatype nat = 0 | s nat
coin = 0                                    (1)
coin = s 0
```

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

```
add 0 Y = Y
add (s X) Y = s (add X Y)
positive 0 = false                          (2)
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. 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 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)
```
(3)

and consider the previous example with `bigger` substituted for `coin`. The evaluation of `positive` (`add bigger 0`) (which will be shown in its entirety later) may be performed using fair independent computations as in the previous example. 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 usage, and the duplication of the result.

In this paper, we discuss a programming technique that has been evaluated in the context of a project aiming at the implementation of a back-end for a wide class of functional logic languages [7]. In some cases, this technique has the potential to offer substantial improvements. In other cases, it tends to consume slightly more memory, but without a substantial slowdown.

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 technique that is the focus of our work. In some cases, this technique reduces the computing time and/or the memory consumption attributed to non-deterministic computations. Section 5 discusses a transformation that introduces our technique into a program. The correctness of this transformation is further analyzed informally. Section 6 both theoretically and experimentally evaluates our technique using some examples. Section 7 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 functional logic programming languages proposed early on neglect non-determinism. Programs in these early languages [8,16] are 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 be severely impaired [4, Ex. 3].

More recently [4,9], 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 minimal effort. For example, the following operation creates a regular expression over an alphabet of symbols. To simplify the example, we exclude the empty string ϵ from our regular expression language. Any such regular expression may be obtained by appropriate non-deterministic choices during computation.

```
regexp X = X
regexp X = "(" ++ regexp X ++ ")"
regexp X = regexp X ++ regexp X                              (4)
regexp X = regexp X ++ "*"
regexp X = regexp X ++ "|" ++ regexp X
```

The defined operation `regexp` closely resembles the formal definition of *regular expression* found in standard sources [1, p. 94]. This transparency in semantics can be quite convenient for the programmer. For example, to recognize whether a string s denotes a well-formed regular expression over some alphabet such as

```
alphabet = "0"                                               (5)
alphabet = "1"
```

it suffices to evaluate `regexp alphabet` $=:= s$.

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 computation are quite useful—in particular, techniques that limit the number of fair independent computations that originate from a non-deterministic choice. The overall goal of this paper is the study of a technique for this purpose.

3 Cost Analysis

The most common approach to analyzing the efficiency of a program is measurement of its execution time and memory usage. We measure the execution time of benchmark programs using 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 adds an interesting twist to the situation.

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

Definition 3.1 [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,[2] 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 symbols with positive arity in t, together with 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 with 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.

Most cost analysis techniques proposed in the literature are for the analysis of deterministic computations. Most work on non-deterministic computations in functional logic programming postdates these cost analysis techniques: non-determinism introduces significant theoretical and practical complications.

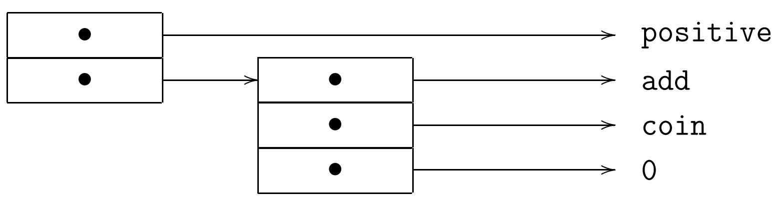
To ensure operational completeness, non-deterministic computations must be executed fairly. The requirement of fairness has an interesting consequence. When a program outputs a result (derived, for example, by using the first alternative in a non-deterministic computation), the measured values of time and resource usage may include resources spent to partially compute other results that have not yet been output. Thus, the time and space resources consumed from the beginning of the execution to the time of the output may be an overestimate of the cost of computing the result. The extent of these computations is generally difficult to estimate. Consequently, a quantification of the resources spent by these computations is difficult to achieve. A potentially better approach would be to measure the resources needed to compute all the results of a non-deterministic computation. However, this is impossible in practice for computations over an infinite search space: Example (4), the

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

`regexp` example presented earlier, provides one illustration of this.

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 variation of our technique (discussed in Section 4.2) is not well-suited to use with builtin types such as binary integers.

We informally reason about the number of steps of a computation and the memory occupied by term representations. 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 specifically allocated to represent any term, but are shared by all terms. In our run-time environment, Prolog, symbols are fully applied. Thus, the arity of an occurrence of a symbol is not a part of the term.



Previous work in analysis [2], like the adaptation introduced in this section, is limited to deterministic computations. Extending this analysis 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 Technique

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. To achieve this, we employ a programming technique that achieves 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 implementations of our technique have been automated in our current system [7].

Our programming technique is based on the introduction of a new symbol, the infix *alternative* operator "!", into the signature of the TRS modeling a functional logic program. The new symbol may be regarded either as a polymorphic defined operation or as an overloaded constructor.

4.1 The *Alternative* Operator

In the first variation of our programming technique, the *alternative* operation is defined by the rules

```
X ! Y = X
X ! Y = Y
```
(6)

In the work of González-Moreno *et al.* [9,10] an operation with these rules is called *alt* and denoted by "//". We can 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 is obviously commutative.

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

```
bigger = s (0 ! s 0)
```
(7)

The significant difference is that `bigger` has been "factored": a common portion of the right-hand sides of the two rewrite rules of the original definition has been combined, and the two rules have been merged. This new definition might be coded explicitly by the programmer, or might be automatically obtained from the original definition by an optimizing compiler or other specialized tool.

The advantage of this definition of `bigger` over the original one is in computational efficiency. If only the factored portions of the two alternative right-hand sides of the rewrite rules of `bigger` are needed by a context, no fair independent computations are created by a *needed* strategy [4]. Instead, a single deterministic computation suffices. 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
```
(8)

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 for fair independent computations, the run-time cost of factoring is a single additional rewrite step. For realistic programs, this cost is negligible. Hence, factoring the 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 little additional memory.

4.2 The *Alternative* Constructor

A second, different 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. Thus, the symbol "!" must be overloaded 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 textually identical to the previous one:

```
bigger = s (0 ! s 0)
```
(9)

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 of the set {`s 0`, `s (s 0)`}.

We believe that extending builtin types (such as the integers or booleans) or well-known types (such as the naturals) is generally 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 extend the sort `nat` to include the alternative constructor here: this achieves a certain immediateness and eases the comparison with Section 4.1.

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 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)
```
(10)

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))
```
(11)

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:

$$\pi[X\,!\,Y]_p \to \pi[X]_p\,!\,\pi[Y]_p \tag{12}$$

The advantages of factoring right-hand sides when the *alternative* symbol is an operation are also preserved by additional rewrite rules of this kind when the *alternative* symbol is a constructor. 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 cases, 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)
```
(13)

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 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 variation for problems where only a small fraction of the potentially computable values are actually needed.

5 Transformation

There is a substantial difference between the *alternative* operation and the *alternative* constructor. Extending a rewrite system P with the former does not change the computations of P. By contrast, extending a sort of a rewrite system P with the latter substantially changes the sort, and consequently the meaning, of P. As we informally discussed in Section 4, if a rewrite system P defines the *alternative* operation, in some cases some rules can be modified to obtain a new system Q with the same meaning as P but with potentially greater execution efficiency. Q is the result of a transformation of P, called *factoring*, that is precisely defined as follows:

Definition 5.1 [Term factoring] Let t, u and v be terms. We say that t is a *product* of u and v if and only if one of the following conditions holds:

(i) $t = u \,!\, v$.

(ii) t is of the form $f(t_1, \ldots, t_n)$, where f is a symbol of arity n and $t_1, \ldots, t_n$ are arbitrary terms. Similarly, u and v are of the form $f(u_1, \ldots, u_n)$ and $f(v_1, \ldots, v_n)$ respectively. Finally, there exists an $i \in \{1, \ldots, n\}$ such that the following two subconditions hold: (1) t_i is a product of u_i and v_i, and (2) $t_j = u_j = v_j$ for all $j \in \{1, \ldots, n\}$ such that $j \neq i$.

For example, both `s 0 ! s (s 0)` and `s(0 ! s 0)` are products of the terms `s 0` and `s (s 0)`.

In the following, we consider rewrite systems defining the alternative operation "!". This assumption is harmless, since the operation can be added to any program that does not already define it without changing the meaning of existing operations. The following definition considers rules whose left-hand sides are variants, i.e., they can be made identical by renaming variables. Rules of this kind are not unusual in practical programs. This is the only kind of overlapping allowed in *overlapping inductively sequential* rewrite systems, a class that supports both non-deterministic computations and optimal lazy evaluation [4].

Definition 5.2 [Program factoring] Let P be a rewrite system defining the alternative operation "!", and $l_1 \to r_1$ and $l_2 \to r_2$ rules of P such that l_1 and l_2 can be made identical by renaming of variables. Without loss of generality we assume that $l_1 = l_2 = l$ (since renaming the variables of a rule does not change the rewrite relation). A rewrite system Q *factors* P if and only if at least one of the following two conditions holds:

(i) $Q = (P \backslash \{l \to r_1, l \to r_2\}) \cup \{l \to r\}$, where r is a product of r_1 and r_2.

(ii) Q factors some program P' and P' factors P.

Informally, to factor a program P we proceed as follows. We arbitrarily select two rules whose left-hand sides are equal after renaming their variables. We merge the two rules into a new single rule whose right-hand side is the alternative of the right-hand sides of the selected rules. Furthermore, if possible, we push the root alternative operator down the term by factoring some common part. It should be intuitively clear from the discussion in preceding sections that the deeper the alternative operator can be pushed down the right-hand side, the more likely it will be to replace two fair independent computations with a single computation.

To address the correctness of a program transformation when programs are modeled by constructor-based rewrite systems, terms built only from constructor symbols are considered data or literal *values*, whereas terms containing operations are considered *computations*. We are interested only in computations that rewrite to values.

Definition 5.3 Let $\Im$ be a mapping from constructor-based rewrite systems to constructor-based rewrite systems. Let P and Q be constructor-based rewrite systems with the same signature such that $Q = \Im(P)$. $\Im$ is *complete* if and only if for every term t and value v, $t \xrightarrow{*} v$ in P implies $t \xrightarrow{*} v$ in Q. $\Im$ is *sound* if and only if for every term t and value v, $t \xrightarrow{*} v$ in Q implies $t \xrightarrow{*} v$ in P.

Our concern is the correctness of the factoring transformation given by Def. 5.2. Unfortunately, in some cases, factoring a program does not preserve its meaning. Consider the following program, which uses the operation `add` defined in Example (2):

```
double X = add X X
f = double 0
f = double (s 0)
```
(14)

Factoring (14) produces

```
double X = add X X
f = double (0 ! s 0)
```
(15)

It is simple to verify that `s 0` is one of several values of `f` in (15), but it is not a value of `f` in (14). The problem with unrestricted factoring is soundness, not completeness.

Claim 5.4 *The factoring transformation is complete.*

To prove this claim, let P and Q be programs and $R_1 : l \to r$ and $R_2 : l \to s$ rewrite rules of P and Q respectively. Suppose that Q factors P and r factors s. If t rewrites to u with rule R_1, then t also rewrites to u using only R_2 and the rules of "!".

As we have seen, factoring is generally unsound, but soundness can be recovered in some cases of practical interest. In the following discussion, the notions of *descendant* of a redex is informal, as this notion has been rigorously defined only for orthogonal rewrite systems [13]. The unsoundness of factoring originates from computations in which some redex r generates several descendants, and distinct descendants are reduced to different terms. Thus, two simple independent solutions to the unsoundness problem are available: avoiding distinct descendants of a redex is sufficient, as is ensuring that all the descendants of a redex are contracted using the same rewrite rule.

The first solution can be achieved by restricting ourselves to right-linear rewrite systems. This restriction, however, seems too onerous for whole-program transformation. Unfortunately, it is not difficult to see that restricting factoring to expressions containing only operations defined by right-linear rules is not a strong enough condition to ensure soundness.

The second solution can be achieved by the use of *call-time choice* semantics [14]. Informally, this approach consists of "committing" to the value of an argument of a symbol (either operation or constructor) at the time of the symbol's application. The value of the argument does not need to be computed at application time: thus, the laziness of computations is not jeopardized. This semantics is the most appropriate for some computations, but it fails to capture the intuitive meaning of others, such as the recognizer of regular expressions discussed in Example (4). Nevertheless, there are formalisms [10] and languages [15] based on these formalisms that adopt call-time choice as the only semantics of non-deterministic computations. Factoring is sound in this case. The programming language Curry also adopts call-time choice: thus, all of our examples have sound Curry semantics.

At this time, we do not have a proof of soundness for either right-linear rewrite systems or call-time choice semantics.

6 Examples

In this section we benchmark some examples of our technique using two distinct strategies. The "theoretical" approach applies a cost analysis to the examples. The "experimental" approach actually implements the examples and measures their performance. This two-tiered strategy is useful both for validation and for the insight it provides into performance issues.

6.1 Theoretical Benchmarks

In order to reason about the advantages and disadvantages of our technique, 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:

We consider a complete independent computation for each non-deterministic step. Two independent computations differ for at least 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 actually is.

We consider the computations of `positive` (`add bigger 0`) with and without using our technique. In the 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, 2, and 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 technique cuts the number of steps in half and reduces the memory consumption by 25% when the alternative symbol

Step	*Rule*	$\mathcal{A}$
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*	$\mathcal{A}$
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*	$\mathcal{A}$
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.

is defined as an operation. These exact savings are also obtained when the alternative symbol is defined as a constructor.

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, 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. When "!" is an operation, the computation uses 6 steps and 13 units of memory, an increase of 8%. This is an example where our technique is not effective, since it is impossible to "merge" the two fair independent computations arising from the computation of `half bigger`.

It is noteworthy that, in these examples, an existing implementation [6] of Curry allocates memory according to our theoretical model.

6.2 Experimental Benchmarks

The examples of Section 6.1 are too small and artificial for understanding the effects of our programming technique in practice. Also, our theoretical analysis is difficult to apply to programs that make more than a few steps. For this reason, we have benchmarked both the memory consumption and the execution times of some larger programs. Our programming language is Curry [12]. The compiler is Pakcs [11], which transforms Curry source code into Prolog for execution.

The first program that we have benchmarked 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

	regular program			our technique			
Problem	Runtm	G. stck	L. stck	Runtm	G. stck	L. stck	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%

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

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 of value 24 in which each digit occurs exactly once. 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), such as $3*(2+8)-6$ 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 dataset is for a version of the program that does not use our factoring technique. The second dataset is for a program that uses our technique: the alternative symbol is a defined operation. Defining the alternative symbol as a constructor does not seen appropriate for this problem. The runtime measures are nearly identical over several executions. The memory measures are constant over all executions.

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.

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 extremely simple 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
```

	regular program			our technique			
Input	Runtm	G. stck	L. stck	Runtm	G. stck	L. stck	Speedup
"1+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%

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

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

$$\mathit{Bin}\ \texttt{'+'}\ (\mathit{Num}\ \texttt{"1"})\ (\mathit{Bin}\ \texttt{'-'}\ (\mathit{Num}\ \texttt{"2"})\ (\mathit{Num}\ \texttt{"3"})).$$

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

$$\begin{array}{rcl} \mathit{expression} & ::= & \mathit{term}\ \texttt{'+'}\ \mathit{expression} \\ & \mid & \mathit{term}\ \texttt{'-'}\ \mathit{expression} \\ & \mid & \mathit{term} \\ \mathit{term} & ::= & \texttt{'('}\ \mathit{expression}\ \texttt{')'} \\ & \mid & \mathit{digits} \end{array}$$

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]` $\rightarrow$ `[Char]` $\rightarrow$ `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, the *expression* operation 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, also, our 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.

6.3 Applicability Analysis

Obviously, the effectiveness of our technique depends on the specific program in which it is applied and the nature of the computation. The more a computation is non-deterministic, the more opportunity it provides for the application of our technique. Conversely, our technique is useless in a totally deterministic computation. Libraries that define non-deterministic operations offer an opportunity to exploit our technique. Programs with non-deterministic computations are likely to use operations defined in these libraries. Thus, an analysis of these operations somewhat epithomizes the applicability of our technique and an improvement of these operations indirectly improves the execution of a potentially large number of programs.

The Pakcs distribution of Curry includes a set of libraries for frequently occurring computations such as character and string manipulation, operations on sets and graphs, utilities for reading and parsing, etc. One of these libraries, *Combinatorial.curry*, provides a handful of operations for combinatorial algorithms such as permuting a list or partitioning a set. As one should expect, some of these algorithms are directly or indirectly based on non-deterministic operations. An analysis of the non-deterministic operations in this library allows us to better understand the applicability of our technique and to improve the efficiency of some computations.

The following operation computes a permutation of a list.

```
permute [] = []
permute (x:xs) = ndinsert (permute xs)
    where ndinsert ys = x:ys                            (16)
          ndinsert (y:ys) = y:(ndinsert ys)
```

The non-determinism of operation `ndinsert` originates from rules that are overlapping, but cannot be factored since their left-hand sides differ. Hence, our technique is not applicable to this operation.

A second operation of the library computes the set of subsets of a set. This operation is based on an auxiliary operation, `subset`, which computes a sublist of a list. This operation is non-deterministic in the sense that `subset` l returns any sublist of the list l. The exact meaning of "sublist" is easily inferred from the code of operation `subset`:

```
subset [] = []
subset (x:xs) = x:subset xs                              (17)
subset (_:xs) = subset xs
```

To properly use lists to represent sets, some suitable assumptions are enforced. For example, lists representing sets do not contain duplicate elements, and the order of the elements in these lists cannot be observed. Given this representation, the set of all the subsets of a set is easily computed using Curry's primitive `findall` as follows:

```
allSubsets list = findall \x -> subset list =:= x        (18)
```

Our technique can be applied to operation `subset` by factorizing its second and third rule. However, this factorization produces no benefits in the Pakcs implementation of Curry. Actually, our technique marginally slows down computations involving `subset`. The factorized code:

```
subset [] = []
subset (x:xs) = x:z ! z where z = subset xs              (19)
```

prompts some interesting considerations, though. Here, the two non-deterministic alternatives do have a common part, but this part cannot be factorized since it consists of an inner subterm rather than an outer one. According to the semantics of Curry, variable `z` in (19) is shared. Sharing occurrences of the same subterm in different arguments of an application of operation "!" is semantically useless, since at most one argument contributes to the computation. Furthermore, in the Pakcs's implementation of Curry, which maps to Prolog and is based on backtracking, sharing adds a further overhead without benefits. If backtracking takes place, the expression bound to `z` must be re-computed.

However, the implementation proposed in [7] is based on fair independent computations rather than backtracking. This strategy ensures that shared subterms are computed at most once. In computations that explore both branches of an alternative, our technique could be quite beneficial since the expression (`subset xs`) would be computed only once.

A third non-deterministic computation of the *Combinatorial* library is implemented by an operation, `sizedSubset`, that takes a non-negative integer c and a list l and returns a sublist of l of length c. This operation is non-deterministic in the same sense as `subset`. Coding this operation is somewhat tricky. We will discuss its design which shows that factorization is a natural and useful technique regardless of our analysis.

The simples design condition is when `c` $= 0$. In this case, the result is the empty list. Therefore, one would be tempted to code:

```
sizedSubset 0 _ = []
...                                                        (20)
```

Unfortunately, since the integers are a builtin type, it is impossible to pattern match the condition `c` > 0. This suggests replacing the above attempt with:

```
sizedSubset c l = if c==0 then [] else ...                 (21)
```

The `else` clause operates on the head and/or tail of the second argument, `l`. It entails a non-deterministic choice: either to exclude the head of `l` from the result and recur on the tail of `l`, or to include the head of `l` in the result and complete the result with a sublist of size $c-1$. However, a new problem arises. The decomposition of `l` into head and tail parts cannot be done by pattern matching, since this would break the semantics of the `then` clause when `l` is the empty list. Thus, one could naively code:

```
sizedSubset c l = if c==0 then []
    else let (x:xs) = l in x:sizedSubset (c-1) xs
sizedSubset c l = if c==0 then []                          (22)
    else let (_:xs) = l in sizedSubset c xs
```

Unfortunately, the above code is flawed: each result is computed twice due to the overlapping of the two rules' left-hand sides. Using our technique leads to the following code:

```
sizedSubset c l = if c==0 then []
    else let (x:xs) = l                                          (23)
         in x:sizedSubset (c-1) xs ! sizedSubset c xs
```

which computes the intended function. An equivalent and marginally more efficient formulation of the same computation uses an auxiliary operation:

```
sizedSubset c l = if c==0 then [] else aux c l
    where aux c (x:xs) = x:sizedSubset (c-1) xs                  (24)
          aux c (_:xs) = sizedSubset c xs
```

This example shows that factorization, whether explicitly obtained with the alternative operator or implicitly encoded in the structure of a program, is a natural, useful and sometimes necessary programming technique.

7 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 a simple programming technique intended to improve the efficiency of certain non-deterministic computations. Our technique is based on the introduction of a new symbol, called *alternative*, into the signature of a program. This symbol allows a program to factor a common portion of the non-deterministic replacements of a redex. We have considered two variations of our technique: in one of them, the *alternative* symbol is a polymorphic defined operation; in the other, the *alternative* symbol is an overloaded constructor.

Our 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 variations of our technique, savings are obtained when fair independent computations are avoided, because only the factored portion of non-deterministic replacements is needed. For the second variation, 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 variation is small. There are cases in which the application of the second variation may actually result in computations that consume more memory.

Our technique can be easily adopted by a programmer. One variation can be introduced into a program by a transformation that is easy to automate. The transformation is complete and it is sound under suitable assumptions. We have quantified the effects of the application of our technique or transformation in simple examples. Our technique is applicable to programs coded in several existing or proposed functional logic programming languages, and in some practical cases provide substantial efficiency improvements.

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
```

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)*, pages 199–206. ACM Press, 2001.

[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)*, pages 207–217. ACM Press, 2001.

[8] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: a logic plus functional language. *Journal of Computer and System Sciences*, 42:139–185, 1991.

[9] 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.

[10] 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. *Journal of Logic Programming*, 40:47–87, 1999.

[11] M. Hanus, S. Antoy, J. Koj, R. Sadre, and F. Steiner. PAKCS 1.3: The Portland Aachen Kiel Curry System User Manual. Technical report, University of Kiel, Germany, 2000. Web document. URL `http://www.informatik.uni-kiel.de/~pakcs` accessed April 15, 2002 08:35 UTC.

[12] M. Hanus (ed.). Curry: An Integrated Functional Logic Language, 2000. Web document. URL `http://www.informatik.uni-kiel.de/~mh/curry/report.html` accessed April 15, 2002 08:34 UTC.

[13] G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*. MIT Press, Cambridge, MA, 1991.

[14] H. Hussmann. Nondeterministic algebraic specifications and nonconfluent rewriting. *Journal of Logic Programming*, 12:237–255, 1992.

[15] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proceedings of RTA '99*, pages 244–247. Springer LNCS 1631, 1999.

[16] J. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.