Improving the Efficiency of Non-Deterministic Computations in Curry

A Thesis Submitted in Partial Satisfacation of the Requirements For An Honors Baccalaureate Degree in Computer Science

Jason Wilcox

May 14, 2004

Contents

List of Figures

List of Tables

1 Introduction

Declarative languages[7] provide powerful techniques that allow programmers to work at a higher-level of abstraction. This leads to a focus on algorithms and design rather than on low-level details by pushing the details into the compiler. The claim is that this abstraction leads to more elegant solutions developed in a shorter amount of time than solutions in imperative languages. A recent development in declarative programming has been the emergence of functional logic languages[7], which hope to increase the level of abstraction by merging the power of the two most popular declarative paradigms.

Hanus et al.[9] have been relatively successful in developing a compiler, called PAKCS, for the functional-logic language Curry[8]; unfortunately, the depth-first evaluation strategy of their Prolog[4]-based back-end leads to operational incompleteness. Current efforts are therefore focused on a virtual machine (hereafter VM) that uses both[7] narrowing and residuation to conduct functional-logic computations and provides operational completeness[1] by exploiting concurrent programming techniques.

In this paper, we present a technique for improving the efficiency of non-deterministic and narrowing[2] computations within the current VM[1]. These improvements will assist the current VM in closing the performance gap between itself and PAKCS[9]. The technique that we have developed allows the cost of computations to be proportional to their longevity rather than only proportional to their size.

The overall organization of our paper is fairly straightforward. Section 2 begins by presenting a more in-depth overview of functional, logic, and functional logic languages so that the reader has a better understanding of the context of our research. Section 3 then discusses the organization of the current VM, and its shortcomings in handling non-deterministic and narrowing steps in computations. A detailed examination of technique to improve these computations is then presented in section 4. This examination explores both the theoretical idea behind our research and its practical implementation. Section 5 then provides some benchmarking data comparing our implementation against the current one. This will demonstrate that our technique really does improve the efficiency of computations. Finally, Section 6 presents our final conclusions and some direction for future work. Many of the examples used in this program come from a paper by Antoy[3], and we thank him for allowing us to use them.

2 Functional Logic Programming and Curry

Functional logic languages[7], such as Curry[8]and Toy[13], attempt to combine the best features of two much older language paradigms: functional programming[11], and logic programming[4]. The goal of merging these features together is to create a class of languages that is as expressive as possible without sacrificing too much efficiency. This section will describe the important features of both these paradigms before describing some of the unique features of functional logic languages. This will establish the context in which our research occurred.

2.1 Functional Languages

Functional languages, such as Haskell[12] and ML, arose from a desire to concentrate on what a program computes rather than focusing on the specific instructions that must be executed by the underlying machine. As we will see, Curry is really just an attempt to further pursue this ideal. As its name suggests, this community decided the best building block for solving problems in this manner would be functions.

In a functional language everything is treated as a function, although some are degenerate like the number three, or the letter a. These functions are interesting because they have a closer relationship to functions in a mathematical sense than "functions" represented in other languages. Namely, there is only one result computed by a function for a given input. Functions exhibit this property because they are, for the most part, side-effect free. This means they do not update or destroy variables, consume input, or perform similar operations. Hopefully, this makes reasoning about programs easier because if something goes wrong within a function it is apparantly because of that function and not a peculiar interaction with its surrounding context.

Another interesting trait of functional languages is that they support the concept of higher-order functions. A higher-order function is one that can take other functions as arguments; a function that takes only degenerate values as arguments is first-order, hence one that takes first-order functions as arguments must be second-order, etc. This allows programmers to build very abstract, powerful, functions that can be reused in a variety of situations. For example, it is common for Haskell programmers to approach problems by thinking in terms of the higher-order function *foldr*.

The final trait of functional languages that we wish to mention briefly is

```
factorial 0 = 1
factorial n = foldr (*) 1 [1..n]
```

Figure 1: Haskell Implementation of factorial

that it provides a robust, easy-to-use, mechanism for user-defined data types. Once again, its main strength is its close relationship to mathematics and its expressiveness as compared to other languages. This simplicity allows the programmer to focus on using the data type to solve his specific problem instead of focusing on the minutiae of implementing the data structure correctly. Figure 3 shows a data type definition.

Before completing this quick summary of functional languages we wish to present a brief example of a functional program, which uses higher-order and piecewise functions. The program in figure 1 is a naive Haskell implementation of a function that computes the factorial of a given number.

2.2 Logic Languages

Logic languages, such as Prolog[4], utilize propositional logic and Horn clauses[10] to perform computations. They differ from traditional languages because they compute whether something is true within a given axiomatic system rather than a value. The techniques needed to compute these answers include predicates, logic variables, and non-determinism.

Predicates are at the core of logic languages. Predicates specify a truth-valued function that denotes a relationship between its arguments. If the arguments to the predicate satisfy the relationship then the result is *yes* (or success), and the result is *no* (or failure) if they do not. Figure 2 shows the two predicates necessary to express the factorial relationship. The first predicate states that 1 is the factorial of 0. The second predicate states that N is the factorial of M if the conditions on the right-hand side are all true.

Logic variables are a more interesting feature of these languages. Logic variables start off free, but become bound in an attempt to satisfy the predicate as execution proceeds; if an appropriate binding cannot be found then that instance of the predicate is false. For example, the variable J is an unbound logic variable in the goal factorial(3,J), and it will be instantiated to the value 6 in order to satisfy the predicate. Even though this does compute that 6 is the factorial of 3, it is important to remember that the result of the computation is the fact that the predicate can be satisfied, and not the value

```
factorial(0,1).
factorial(M,N) :- M ¿ 0, M1 is M-1, factorial(M1,N1), N is M*N1.

dice(1).
dice(2).
...
dice(6).
```

Figure 2: Prolog Implementation of Factorial and Dice

6.

Logic variables also enable another powerful feature of logic languages: non-determinism. Non-determinism is possible in logic languages because there may exist several bindings of a logic variable that satisfy the goal. The dice predicates in figure 2 demonstrates this property because the goal dice(M) can be satisfied by any integer between one and six. In logic languages, this feature may not be that interesting because we are only interested in a truth value, but as we will see below this can provide a lot of power when we are interested in calculating values.

2.3 Curry

Curry takes the powerful features of both these language paradigms to create an efficient, highly expressive, programming language. Its primary strength is that it allows programmers to develop programs by specifying the equations a solution must satisfy, rather than the specific algorithm necessary to generate the solution. Unlike logic languages, its primary purpose in satisfying these equations is to produce a value. We need to revisit non-determinism and discover narrowing to understand how Curry does this.

In Curry, non-determinism allows functions to have multiple definitions that may return different values for the same input arguments. Furthermore, the use of unbound variables is not necessary to induce non-deterministic results. This mechanism allows programmers to make an arbitrary choice of values when no simple algorithm exists to make a better one. The expectation is that later on an equation will constrain the choice to the appropriate one(s) for that solution.

This brings us to an important point, which is that functions can fail rather than producing a value. Unlike functional languages, this is even desirable because it filters out the unproductive choices and continues on with the (possibly) productive ones. This notion that failure is acceptable, and will probably occur often, is at the core of this thesis. Our work centers around reducing the performance penalties that each failure incurs.

Returning to how Curry produces values that satisfy equations, the concept of narrowing[2] needs to be introduced. Narrowing is built on top of non-determinism, but the values that can be chosen are patterns from a function's left-hand side. These patterns are non-deterministically chosen as instantiation values for an unbound (or logic) variable. Once again, the incorrect values should be filtered out via equational constraints. Narrowing may need to be recursively applied many times to substructures of the initial instantiation before a correct value is found.

Now that we have briefly seen how Curry produces values that satisfy equations, we wish to present a concluding example that will be used throughout the rest of this paper to illustrate concepts. Figure 3 is a Curry program that implements Dijkstra's[6] Dutch National Flag problem. The problem is to rearrange a given list of n objects, which are colored red, white, or blue, so that objects of the same color are adjacent. An important feature of this algorithm is that the items in the list are actually swapped rather than simply creating a new list with the same properties as the first one except the colors are already adjacent. Additionally, the colors must be sorted so that all of the red objects appear before the white ones, and all of the white objects appear before the blue ones.

The example is interesting because it uses many of the features we have discussed above. The functions mono and solve are both non-deterministic, which is exploited to keep the program simple. Solve also uses narrowing to instantiate the free variables x, y, and z such that the values on either side of the *=:=* are unifiable. If the equation can be satisfied then the body of that particular rule of *solve* is executed. If not, an incorrect non-deterministic choice was made so another choice must be tried. This continues with recursive calls to *solve* until the conditions of a solution, as expressed by the fourth rule of *solve*, are satisfied.

3 The Current Implementation

The implementation of Curry our research is based off of is a virtual machine (VM) developed by Prof. Sergio Antoy and his research group[1], which is

Figure 3: Curry Implementation of the Dutch National Flag Problem

```
data Color = red | white | blue

mono _ = []
mono c = c : mono c

solve flag | flag =:= x ++ white:y ++ red:z
           = solve (x ++ red:y ++ white:z)
           where x,y,z free
solve flag | flag =:= x ++ blue:y ++ red:z
           = solve (x ++ red:y ++ blue:z)
           where x,y,z free
solve flag | flag =:= x ++ blue:y ++ white:z
           = solve (x ++ white:y ++ blue:z)
           where x,y,z free
solve flag | flag =:= mono red ++ mono white ++ mono blue
           = flag
```

written in Java. The VM is an attempt to provide an operationally-complete and sound implementation of the Curry language. It leverages the theory of term-rewriting systems[5] and the power of concurrent programming to accomplish this. To understand our research better, an overview of the VM and its shortcomings needs to be presented.

3.1 Terms

Terms are the primary unit of evaluation in the VM. This is not surprising since it is based off of term-rewriting systems. Terms are composed of variables, operations over other terms, and constructors over other terms. As evaluation within the VM progresses the terms are replaced by new terms via rewrite rules until no rewrite rules can be applied. At this point we say the term is in normal form. The VM imposes the additional requirement that normal forms must be constructor terms, meaning that the term and all of its subterms must consist of only constructors.

A constructor-rooted term is one in which the root of the outermost term is a constructor, such as *:*, which is the non-empty list constructor. Similarly, a term is operation-rooted if the outermost term is an operator, such as ++. Constructor- and operation-rooted terms have subterms associated with them, which are the terms that the constructor or operation is applied to. In turn, these subterms may contain other subterms, and therefore they may need to be rewritten before anything useful can be accomplished on the top-level term.

During the process of rewriting it may also be necessary to apply a substitution to a term. Substitutions express the mapping of variables to terms, or inotherwords the binding of a variable to a term. When a substitution $\{x \mapsto t_2\}$ is applied to a term t_1, it states that every occurrence of x within t_1 should be replaced by t_2. This substitution is carried through every subterm of t_1. The goal of a substitution is to allow new rewrite rules to be applied that will hopefully move t_1 closer to its normal form.

Two important definitions related to terms are still needed: redex and narrex. Allterms that can be rewritten are known as redexes, meaning reducible expression. A narrex is a term that can take a narrowing step, which includes rewriting. Therefore, every redex is a narrex, but not every narrex is a redex. The narrowing strategy employed by the VM requires that all narrexes be operation-rooted.

3.2 Computations

The management of this complex rewriting system is not left to the terms themselves to manage. Instead, a computation abstraction is added onto the virtual machine. Each computation is given one term that it is tasked with rewriting, in most cases to normal form. Over the course of its life each computation may actually evaluate many terms; each one a subterm of the original top-level term.

This computation abstraction manages the actual task of evaluating a term by performing a series of rewrite "steps." At each step the computation determines if the current term needs to be rewritten further. This is accomplished by querying the term to determine if it is in either normal or head-normal form. If further rewriting is necessary, the computation executes a sequence of instructions associated with the current term. This series of "steps" continues until either the computation fails or further rewriting is no longer possible.

The computation maintains a pre-narrex stack to keep track of the subterms that it must rewrite. A subterm is pushed onto this stack once the computation determines it needs to be rewritten, and a subterm cannot be put onto the stack unless its parent term is already on it. Evaluation proceeds by popping the next term off of the stack once the current one can not be rewritten any further; the previous term should be a subterm of this new term. This method of evaluation is very top-down in nature because the parent term always invokes the rewriting of its subterm as well as providing the subterms their execution context.

3.3 Non-determinism, Narrowing, and Efficiency

The efficiency problems with this implementation arise when non-deterministic steps are taken within a computation. This is likely to occur often since non-determinism is one of the primary advantages of Curry. Two attributes of the current implementation contribute this inefficiency: first, the use of a top-down execution strategy that can lead to redundancy; second, this strategy must be repeated for each of the n distinct choices presented whenever we narrow or make a non-deterministic step. An examination of how the machine behaves in these situations will be presented so that the reader can understand the magnitude of the inefficiencies.

When a non-deterministic step is encountered, the machine must prepare to evaluate the top-level term with each distinct choice replacing the current term. For example, if the current, non-deterministic, term presented us with four distinct choices then the machine must evaluate the top-level term four times: once for each distinct choice; shared subterms that are not dependent on the non-deterministic choice will be shared amongst the evaluations. This helps mitigate the cost of non-determinism to a certain extent. The machine executes the terms for each choice in parallel to ensure no term will block the others from executing even if it continues rewriting forever; this also makes the computation more non-deterministic than simply choosing the first choice that satisfied all constraints and allowed the top-level term to evaluate to a normal form. However, this also means that the steps described in the following paragraphs will be performed for each choice simultaneously, which will compound the inefficiencies they discuss.

The VM begins by selecting the right-hand of a particular instance of the non-deterministic term. For example, the term *mono* from Figure 3 may be the non-deterministic term and the right-hand side *[]* maybe selected. Every instance of the non-deterministic term (*mono*) within the current top-level term is then replaced by its right-hand side (*[]*). At this point, all ancestors of the newly-replaced terms are copied and updated to reflect their new subterm. This copying propagates all the way to the top-level term. This is necessary because the top-level term needs to reflect the distinct choice that was chosen throughout all of its subterms.

A new computation is then created to manage the new top-level term just created, and the current computation is disregarded. This means that evaluation of the term has to begin at the top-level once again. Terms that had already been rewritten will not be reevaluated, but it still computationally-expensive to push and pop subterms off of the stack and ensure that they are in the proper form already. This process will continue until we reach the term where the last computation became abandoned. At that point, evaluation resumes as normal, but clearly a heavy price was paid to get to this point.

Instantiation of free variables and narrowing proceed via a similar process; obviously, instantiation will only do the process once and not n times. The primary difference is that we are substituting a term for all occurrences of a variable rather than replacing one term with another. The relative expense of these operations is probably equivalent.

Narrowing is much more expensive than plain non-determinism in the long run, however, because narrowing is generally a recursive process. This recursion occurs when a narrowing operation introduces another free variable, and it often does. Figure 4 provides a good example of how this can occur. The VM will try both *[]* and *(x:xs)* as possible instantiations whenever we attempt to narrow on the first argument of *++*. Notice that when *(x:xs)* is chosen as the instantiation, *x* and *xs* are introduced as two new uninstantiated variables. Furthermore, *xs* may be narrowed again because the term *(xs++ys)* is introduced. The machine may then repeat the process of narrowing the first argument of *++* all over again.

Figure 4: Append Operation in Curry

```
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs++ys)
```

4 Improving the Implementation

The above description of the current system shows why the focus of our research has been on improving the efficiency of computations involving narrowing and non-determinism. Their relative expense and frequency in computations means that any improvement in them should provide an appreciable gain in efficiency. However, we also had to be careful in approaching this problem because we did not want our techniques to reduce the efficiency of "normal" computations.

We used an insight about the unique behavior of non-deterministic and narrowing to achieve this goal. Specifically, our insight was that most non-deterministic choices fail quickly. Many do not even satisfy the constraints placed upon them by their parent term. This means that all of the effort placed in carefully copying their ancestor terms, and pushing them onto the pre-narrex stack, is rarely productive, and instead it is usually pure overhead. Clearly, eliminating unproductive computational steps should provide us with the efficiency gains we seek.

The first fruit of our research is a technique that solves this problem. We do this by only performing the copying and substitution on the term that precipitated the non-deterministic step. We then save a small portion of the computation's state inside the new computation. This saved state allows us to rebuild the term's ancestors as they are needed. The new computation is then setup to treat this new term as its top-level term.

Computation proceeds normally from this point until the computation reaches a point where it needs the parent of this term to continue. At this point, the parent term, and all of its subterms, are copied and have the current substitution applied. The subterm the machine has been rewriting up to this point then replaces the newly copied subterm; if we did not do this then our previous rewriting steps on that subterm would have to be repeated. The evaluation of the parent term then continues on until it needs its parent, and this pattern is repeated until the original top-level term is reached.

This approach may sound reasonable, but it doesn't sound like it will gain us any efficiency. Instead, it appears that all we have done is allowed terms to be copied incrementally rather than all at once. The efficiency gain comes in when we determine we need the parent of a term, however. Before we make the copy of the parent, we check to see if the current term has failed, and if so rewriting stops with the entire computation failing. This is perfectly reasonable because a term cannot possibly succeed if any of its subterms have failed.

This model of computation certainly solves the problem, but it makes the structure of the computation quite interesting (and worth a brief tangent). The term begins by evaluating in a top-down manner as in the previous implementation, but after a non-deterministic step it will begin computing in a bottom-up fashion instead. This bottom-up computation is then interrupted by further top-down computations on subterms after each step upwards. This creates a computation that is constantly reaching upwards only to delve by back down into other subterms.

4.1 Implementation

We needed to implement the strategy outlined above in the Curry VM to truly test its validity and efficiency. This implementation presented some interesting challenges and minor deviations from our theoretical strategy. As one might expect, the majority of the work had to take place inside the computation abstraction, but the implementation of terms and some internal instructions also had to change. This section will outline the implementation of our technique within these three areas.

4.1.1 Computation

The first decision that had to be made within computation was how to capture the state of the computation. We decided that a copy of the pre-narrex stack would make the most logical choice because a term is pushed onto it right before being rewritten. This not only captures every term, but also the hierarchy of the terms. Subterms always appear above their parent terms on the stack. Another advantage of the pre-narrex stack is that it contains no frivolous state because once a term no longer needs to be rewritten it is removed from the stack.

At first glance, this approach seems ideal. Computations can simply pre-fill their pre-narrex stack with the stack from the previous computation whenever a non-deterministic step is made. New terms will be added onto the stack as computation progresses, and when only the copied terms remain we begin computing upwards.

Remember that the structure of a computation is not nearly as ideal as this approach assumes, however. Instead, top-down evaluation may interleave with bottom-up evaluation frequently for non-deterministic terms. The approach we chose to take keeps the old and the new pre-narrex stacks separate. The old stack manages the bottom-up state of the program while the new stack manages the top-down state.

Managing this new state is therefore the primary new task the computation abstraction has to manage. This task is localized within a single method, which is shown in Figure 5. Localizing this management into one place allows the rest of the computation to believe that it only computes in a top-down manner, which means less likelihood of undesirable side-effects. The only change that had to be propagated was replacing each attempt to pop the pre-narrex stack into a call to the *updatePreNarrex* function.

UpdatePreNarrex itself works by first examining the current pre-narrex stack to determine if it is empty or not. If it is not empty, or if the old stack contains nothing, then none of the new code is executed and the machine behaves just as it did before. This means that traditional deterministic computations behave exactly as they did before, and the only extra overhead they incur is one additional method invocation. It also makes sure that non-deterministic computations compute in a top-down style until an upward step is absolutely necessary.

The interesting work of the code happens when an upward step does become necessary because the current top-down pre-narrex stack is empty.

First, the current term is examined to verify that it has not failed, and if it has failed we simply signal the failure and quit computing at that moment. Otherwise, the top term from the old stack is popped off. Since this term was the last one placed on the stack, it must be the bottom most term. Ignoring the &-operator for a moment, the old term is then cloned (copied) with the current substitution, and has the appropriate subterm replaced by the current term; we will discuss how this replacement occurs later. The computation then reinitializes itself to evaluate our new term to a normal form if possible.

The &-operator throws an interesting wrinkle into our system, however. It is the only process in the machine that can fire off subsequent computations without abandoning itself. Therefore, the machine could easily compute many steps redundantly, or worse, if both computations tried to compute beyond the & in a bottom-up fashion. Instead, the computation indicates to its parent computation that it has successfully evaluated and returns control to it. The only deviation from this is when the computation has no parent computation. It has to continue evaluating beyond the & to ensure that the computation does not terminate prematurely in this case.

4.1.2 Term

Our term implementation's new responsibility lies in replacing the appropriate subterm of the newly copied term. This subterm needs to be replaced by our previously evaluated term so that all the effort that went into rewriting it is not duplicated. Determining which subterm to replace is not trivial process. Problems arise because the evaluated version of the subterm may not share any characteristics with the unevaluated, newly copied, version. This is especially true because the evaluated form may be the result of several non-deterministic choices, and therefore several copies of the term removed from the original.

Terms now keep a reference to their previous incarnation whenever they are copied to avoid this problem; sadly, this reduces the efficiency of the garbage collector. The point of this reference is to allow us to determine if two terms originated from the same original term. Therefore, subsequent copying of a term that had an original parent of its own preserves the reference to the original parent rather than to the term being currently copied. The actual replacement then becomes a fairly trivial task. The goal of this replacement is to plug the previously evaluated term into the proper position within its

```
protected void updatePreNarrex() {
    Term repl = (Term) stack.peek();
    pop();
    if(stack.isEmpty() && !oldStack.isEmpty()) {
        if(repl.getRoot() == SuccessModule.termFail.getRoot()) {
            result.update(repl);
            selfSetState(FAILED);
            return;
        }
        Term oldTerm = (Term) oldStack.pop();
        if(oldTerm.getRootSymbol().equals("&")) {
            if(client instanceof Computation) {
                selfSetState(SUCCESS);
                return;
            }
        }
        Term newTerm = (Term) oldTerm.cloneWithSubst(subst);
        newTerm.replArgument(repl);
        result = newTerm;
        taskCase[NORMAL] = new NFTask(result);
        task = taskCase[NORMAL];
    }
}
```

Figure 5: Managing the Pre-Narrex Stack

newly copied parent term. To determine the correct position we compare the memory location of the evaluated term against the memory location of every subterm in the newly copied parent term. If both the evaluated term and the subterm occupy the same memory location, or if the references to the terms they diverged from are the same, then we have found the correct position. The previously evaluated term therefore replaces the subterm at this position.

Every subterm of the current term is compared against our evaluated term. If the two terms are the same term, or if the original term both progressed from is the same, we replace the subterm with the evaluated term. The equality test is based off of where the term's reside in memory.

4.1.3 Instructions

The behavior of the built-in VM instructions remained remarkably similar. The code that previously dealt with cloning terms and starting new computations was removed and abstracted away into the computation abstraction itself. Instructions now only need to calculate the new substitutions in most places. The only place this did not happen was in the code for the &-operator because it does not want to discard the current computation when it creates new ones.

Our new evaluation strategy for non-deterministic steps did introduce some problems that the built-in instructions had to compensate for, however. The biggest of these involved how substitutions behaved. Substitutions involving bindings for multiple layers of uninstantiated variables would not propagate the substitutions correctly. For example, say the variable l became bound to *(x:y)*, and y became bound to *(2:[3])*. When a term involving l had l's binding substituted for it, only the *(x:y)* would replace l rather than *(x:2:[3])*. If not handled properly, this could lead to multiple bindings for y, unnecessary narrowing steps, and even computations failing when they should not.

To make sure these problems did not arise, we checked to see if the variable was in fact already bound. If not, the narrowing proceeded as before. If it was bound, we would use the facilities described above in term to replace the variable with its binding in its parent term. The narrowing step would then be ignored, and the computation would continue on as before. It should be noted that we are not entirely sure why the new system causes this problem, or that the fix presented here is even a good solution.

5 Measuring Our Improvement

We now present some benchmarking results comparing the current implementation with our new implementation as outlined in this paper. This benchmarking allows us to prove that our technique does improve the efficiency of non-deterministic computations, and quantify the degree of improvement. These benchmarking tests measure two different criteria: the number of built-in instructions executed, and the amount of time it takes for the computation to execute.

Our testing methodology was to run each test ten times to provide a more reliable measurement. The numbers that appear in the tables are the average of the results from the ten tests. Each test was run by starting up a fresh VM and executing the benchmark code. Tests run within an already started VM would provide erroneous results because of the Java runtime's dynamic optimization techniques. When that happens, it is difficult to tell how much of the improvement is a result of our work and how much is a result of the Java runtime system. As a final note on our methodology, our benchmarking platform was a PowerMac G4 867MHz with 640MB of RAM using the Java 1.4.2 runtime environment. Example of DNF, reduction in the number of instructions, reduction in execution time.

5.1 Benchmark Programs

Before presenting the actual results, we wish to present the programs used in our benchmarking and remark on why they were chosen. The first benchmark program is a naive implementation of the Fibonacci numbers shown in Figure 6. It was chosen as a test case because it does not use narrowing or non-determinism. Results from these tests will therefore show whether our implementation had any adverse effects on deterministic computations, which it should have left unchanged. All tests calculated the 26th Fibonacci number.

The second sample program in Figure 6 calculates the last element in a list through extensive use of narrowing on the first argument to ++. Calculating the last element of a list may seem too trivial to provide good results, but on large enough inputs it actually provides meaningful differentiations between implementations; all of our tests involved 100-element lists. Furthermore, a simple non-deterministic computation such as this will likely lay at the core of many programs so its performance may be vitally important just as the performance of a simple inner for-loop in C may have a large effect on an entire C program.

We chose a program that permutes a list as the final sample program; it is also displayed in Figure 6. It was chosen because of how much narrowing is involved in the algorithm presented because of the recursive call to *permute*. Non-Determinism also plays a large role in the program because the variables u and v can be bound in many different ways and still satisfy their constraints. Indeed, this non-determinism is what makes the algorithm works. All tests with *permute* were done on eight-element lists.

```
fibo x | x == 0    = 0
       | x == 1    = 1
       | otherwise = fibo (x-1) + fibo (x-2)

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

Figure 6: Programs used in our benchmarking.

5.2 Instruction Benchmark

The first benchmark we analyzed was how many built-in instructions the VM needed to execute in evaluating a given expression. This metric was chosen because it provides a stable indicator of how the machine is performing. The results are not effected by other processes running on the machine or other irregularities that can interfere with time-based benchmarks. In Table 1 the lower the number of instructions the more efficiently the machine behaved, and the numbers in the second column indicate behavior without our improvements while the third column indicates behavior with our improvements.

Clearly, the results from Table 1 show that VM performance has only improved with our modifications. Calculating the 26th Fibonacci number involved the same number of instructions, which indicates that our technique does not affect the efficiency of deterministic computations. This is exactly the result we were looking for, and expected, since our system is designed

	Current VM	New VM
fibo	19902232	19902232
last	27764	8058
permute	825415	199076

Table 1: Number of Instructions Needed For a Given Computation

to worry about bottom-up computations and additional state only when absolutely necessary.

The second and third rows are even more encouraging because they show a considerable improvement in the efficiency of the VM when our new technique is applied to computations involving primarily non-determinism and narrowing. In particular, the number of instructions executed decreased by roughly 71% for *last*, and it decreased by roughly 76% for *permute*. These are pretty dramatic improvements. More importantly, the benchmark provides reproducable, quantitative, proof that our research has been a success.

5.3 Time Benchmark

The results above may provide quantitative data that validates the effectiveness of our technique, but in the real world what matters to users is time. Therefore, we present and analyze the amount of time taken by the VM to execute the same programs as above. These results provide a more tangible benchmark of the performance of our system. However, they are not as constant as the numbers above because elapsed time can be influenced by external factors. Table 2 presents the results of our benchmarking. As stated previously, the times shown are the average of ten runs of each test so that the effects of external factors can be minimized.

	Current VM (time in ms)	New VM (time in ms)
fibo	35910	36615
last	2595	682
permute	40967	11373

Table 2: Elapsed Time Needed

Once again, these results seem to confirm that we have achieved the goals we set out to attain in our research. The *fibo* test indicates that the two implementations perform equally well on deterministic computations, and the difference in execution time can probably be chalked up to the additional function call overhead our new approach introduces when a term is popped off of the pre-narrex stack; the percent increase is only 2%, which is hardly statistically significant. *last*and *permute* show a 74% and 72% decrease in elapsed time respectfully.

Both benchmarks also correlate with each other nicely. To us, this correlation lends extra credibility to our results. When two seperate benchmarks produce the same results, it is much more likely that the data is correct rather than that the benchmarks are skewed.

5.4 Permutation In-Depth

The permutation benchmark is worth studying in more detail because it exhibits some behavior that the other two tests do not. Namely, it is the only test that can produce multiple answers for the same input because of non-determinism. This search for subsequent answers really stress-tests the VM and emphasizes the performance gap between the current implementation and ours.

Figure 7 shows both VMs trying to calculate several different permutations of an eight-element list. The most obvious fact that jumps out is that our implementation can calculate four permutations of a list using a little over half the time and half the instructions it takes for the current VM to calculate even one. More interestingly, the current VM is not able to calculate the fourth permutation at all while our implementation does so without issue.

Both of these phenomena are due to the greater length of time the average computation lives in the current VM. As we noted in Section 3.3, the pushing and popping of already evaluated terms takes a considerable amount of time. Also consider that each one of these pushing or popping operations constitutes one "step" in the VM. Many computations therefore waste many "steps" trying to die. As these computations languish, other computations are being introduced by further narrowing and non-deterministic steps, which leads to computations having to wait longer between productive "steps." This continues until the machine cannot continue to perform any useful work, and actually runs out of memory. By making sure that computations die as young as possible our approach avoids this problem, or at least prolongs the time it takes for it to manifest.

Figure 7: Calculating Several Permutations in Both VMs

```
[Current VM]
permute [1,2,3,4,5,6,7,8] -> :(1,:(2,:(3,:(4,:(5,:(6,:(7,:(8,[])))))))))
   Elapsed time:40223ms.   steps:37030  instructions:825415  terms:154798
permute [1,2,3,4,5,6,7,8] -> :(2,:(1,:(3,:(4,:(5,:(6,:(7,:(8,[])))))))))
   Elapsed time:67983ms.   steps:46850  instructions:1041629  terms:203159
permute [1,2,3,4,5,6,7,8] -> :(1,:(3,:(2,:(4,:(5,:(6,:(7,:(8,[])))))))))
   Elapsed time:84078ms.   steps:49128  instructions:1089571  terms:213795
java.lang.OutOfMemoryError

[Our Implementation]
permute [1,2,3,4,5,6,7,8] -> :(1,:(2,:(3,:(4,:(5,:(6,:(7,:(8,[])))))))))
   Elapsed time:11724ms.   steps:13895  instructions:199076  terms:66724
permute [1,2,3,4,5,6,7,8] -> :(_4,:(1,:(3,:(4,:(5,:(6,:(7,:(8,[])))))))))
   Elapsed time:22294ms.   steps:24233  instructions:358700  terms:123914
permute [1,2,3,4,5,6,7,8] -> :(1,:(_8215,_8216))
   Elapsed time:24283ms.   steps:27104  instructions:402516  terms:139472
permute [1,2,3,4,5,6,7,8] -> :(1,:(2,:(_3927,_3928)))
   Elapsed time:26206ms.   steps:28134  instructions:418763  terms:145543
permute [1,2,3,4,5,6,7,8] -> ...
```

6 Future Work

The performance measurements demonstrate a marked improvement in the VM's efficiency, but there are further opportunities for improvement that are worth exploring. The first of these is to implement a "clone-on-write" strategy for copying objects, which delays the cloning of subterms until they are actually rewritten.. Currently, whenever a term is cloned so are all of its subterms. This may not be a very wise strategy, however, as many of these terms may never be rewritten and therefore will never change; this is especially true for terms that are about to fail. Object creation is one of the most expensive operations in Java, and therefore removing unnecessary creations should yield good results.

Perhaps the most promising technique for further exploration is a throttling mechanism within the VM. This throttling mechanism would notice when too many computations were alive in the VM at the same time, and it would then enable computations to take multiple "steps" at a time until the number of computations decreased below a certain threshold. The hope is that this technique would eliminate problems such as the one involving *permute*. However, implementing this technique will not easy. The first problem is that we can not allow computations to execute too many steps at a time because we still need to ensure fairness. Furthermore, it will take a lot of tuning to ensure that the throttling engages as little as possible, but is still invoked when there is a good liklihood of recovery from the resource starvation.

7 Conclusion

In conclusion, this thesis has presented a technique for improving the efficiency of narrowing and nondeterministic computations within the Curry VM. It began by outlining the niche within the programming languages community that functional logic languages occupy. Next, we described the current virtual machine and its shortcomings before presenting our bottom-up technique for improving the efficiency of non-deterministic computations. Finally, we presented some quantitative numbers that showed a 71% decrease in the amount of time needed to execute narrowing computations when using our new technique.

Our technique worked by delaying the cloning of terms until that became necessary for the computation to proceed. This allowed us to reduce the cost incurred by computations that failed quickly after taking a non-deterministic or narrowing step. In practice, this failure happens frequently because these computations are usually designed to have poor choices filtered out, which is what facilitated our efficiency gains.

References

[1] Antoy, S., Hanus M, & Tolmach, A. (2003). A Virtual Machine for Functional-logic Languages. *Preliminary Draft.*

[2] Antoy, S. (2001). Constructor-based conditional narrowing. In *Proc. of the 3rd International Conference on Principles and Practice of Declarative Programming*(PPDP'01) (pp. 199-206). Florence, Italy.

[3] Antoy, S. (2002). Programming with Narrowing. *Preliminary Draft.*

[4] Colmerauer, A. (1990). An Introduction to Prolog III. *Communications of the ACM*, *33*(7), 69-90.

[5] Dershowitz, N. (1993). A Taste of Rewriting. *Lecture Notes in Computer Science 693* (pp. 199-228).

[6] Dijkstra, E. W. (1976). *A Discipline of Programming.* New York, NY: Prentice Hall.

[7] Hanus, M. (1997). A Unified Computation Model for Declarative Programming. In *Proc. of the 1997 Join Conference on Declarative Programming* (pp. 9-24).

[8] Hanus, M. (Ed.). Curry: An Integrated Functional Logic Language. (2000). [On-line]. Available: http://www.informatik.uni-kiel.de/ pakcs

[9] Hanus, M., Antoy S., Koj J., Niederau P, Sadre R., & Steiner F. (2000). PAKCS: The Portland Aachen Kiel Curry System. [On-line]. Available: http://www.informatik.uni-kiel.de/ pakcs/

[10] Horn, A. (1951). On Sentences Which Are True of Direct Unions of Algebras. *Journal of Symbolic Logic*, *16*, 14-21.

[11] Hughes, J. (1989). Why Functional Programming Matters. *The Computer Journal*, *32*(2), 98-107.

[12] Jones, S.P. (Ed.). (2003). *Haskell 98 Language and Libraries.* New York, NY: Cambridge University Press.

[13] Lopez-Fraguas, F., & Sanchez-Hernandez, J. (1999). TOY: A Multiparadigm Declarative System. In *Proc. of the 10th International Conference on Rewriting Techniques and Applications*(RTA'99) (pp. 244-247). Trento, Italy.