A Dataflow Implementation Technique for Lazy Typed Functional Languages*

P. Rondogiannis and W. W. Wadge

Abstract

This paper presents an efficient alternative to the reduction-based implementations of functional languages. The proposed technique systematically eliminates user-defined functions by appropriately introducing context-manipulation operators in the program. An abstract architecture for executing the resulting code is presented. A performance evaluation indicates that the proposed approach performs considerably well compared to graph-reduction implementations.

A novel feature of the technique is that it is a *fixed-program* machine. As a result, passing unevaluated arguments to functions has little or no overhead. In addition, the technique is suitable for implementation on dataflow architectures. In fact, our approach gives a solution to the dataflow implementation of higher-order functions, which until now were inefficiently implemented using an expensive scheme based on closures.

1 Introduction

This paper presents a dataflow approach to the implementation of lazy typed functional languages. Given a functional program of order N, the technique gradually transforms it into a zero-order program extended with appropriate context-manipulation operators. We describe the semantics of these operators and present a machine architecture that ensures efficient execution of the transformed program.

The rest of the paper is organized as follows: §2 gives a description of the proposed technique. The first-order case is initially presented and subsequently extended to apply to higher-order programs. §3 discusses the implementation structure of our approach. §4 gives a comparison with graph reduction and also outlines the novel features of the proposed technique. The paper concludes by presenting performance results and giving pointers for future work.

2 The Technique

In order to illustrate the proposed approach, we proceed in two steps. We first consider the case of first-order functional programs. Subsequently, we show how the technique for the first-order case can be generalized to apply to higher-order programs. The programs given are meant to be simple enough so as to be understood without any further syntactic or semantic explanations.

2.1 The First-Order Case

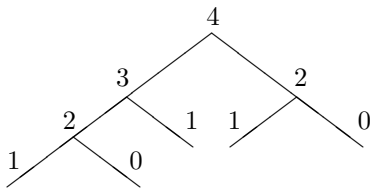
The technique for the first-order case was developed in [15] and described in [3]. However, its extension to the higher-order case remained problematic for many years. To illustrate the method

*Presented to the *Sixth International Symposium on Lucid and Intensional Programming*, Département d'Informatique, Université Laval, Québec, Canada, 26–27 April 1993.

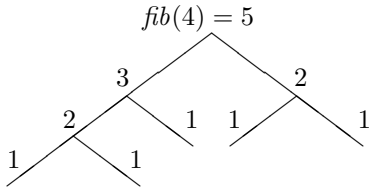
for first-order programs, consider

$$
\begin{array}{l}
\mathit{fib}(4) \\
\texttt{where} \\
\quad \mathit{fib}(n) = \texttt{if } n < 2 \texttt{ then } 1 \texttt{ else } \mathit{fib}(n-1) + \mathit{fib}(n-2); \\
\texttt{end}
\end{array}
$$

which computes the fourth Fibonacci number. In order to compute $\mathit{fib}(4)$, we need to know $\mathit{fib}(3)$ and $\mathit{fib}(2)$. Similarly, $\mathit{fib}(3)$ requires $\mathit{fib}(2)$ and $\mathit{fib}(1)$, and so on. Therefore, one can actually think of the formal parameter n as being a labeled tree of the form shown in Figure 1(a).



(a)



(b)

Figure 1: (a) Tree for the parameter n; (b) Tree for the function *fib*.

Similarly, the function *fib* can be thought of as a labeled tree that has been created by "consulting" the tree for n. Figure 1(b) illustrates the corresponding tree. The bottom labels of the tree for *fib* are all equal to 1, because this is the value that *fib* takes when the corresponding value of n is less than 2. As we move up the tree for *fib*, the label on each node is formed by adding the values of the right and left children of the node. The initial program can be transformed into a new one that reflects the above ideas:

$$
\begin{array}{l}
\texttt{call}_0\ \mathit{fib} \\
\texttt{where} \\
\quad \mathit{fib} = \texttt{if } n < 2 \texttt{ then } 1 \texttt{ else } (\texttt{call}_1\ \mathit{fib}) + (\texttt{call}_2\ \mathit{fib}); \\
\quad n = \texttt{actuals}(4, n-1, n-2); \\
\texttt{end}
\end{array}
$$

The new program does not contain any user-defined functions. Instead, it uses the operators `actuals` and $\texttt{call}_i$ to create the same effect as the trees given above. The definition of n in terms of `actuals` expresses the fact that n is a tree with root labeled 4; the root of the left subtree is equal to the current root minus one, and the root of the right subtree is the current root minus two. Clearly, one can proceed in this way and create the whole tree for n, as given in Figure 1(a). The

operators $\texttt{call}_i$ are complementary to `actuals`: they traverse the tree that `actuals` creates. More specifically, $\texttt{call}_0$ selects the root of the tree, $\texttt{call}_1$ the left subtree and $\texttt{call}_2$ the right subtree.

The algorithm for performing the above transformation in a systematic way is given in [15]. We give a brief description for the purposes of this paper. The source language adopted in [15] is ISWIM. A set of semantic-preserving transformations is used to transform the source programs into ones that have a simpler structure. More specifically, the resulting programs do not have nested `where` clauses, all variables (including formal parameters) are distinct and function definitions do not have global occurrences of nullary variable symbols. The final form of the programs is obtained by the following algorithm [13]:

1. Let f be a function appearing in the program. Number the textual occurrences of calls to f starting at 0.
2. Replace the i-th call of f by $(\texttt{call}_i\ f)$.
3. Remove the formal parameters from the definition of f, so that f is defined as an ordinary individual variable.
4. Introduce a definition for each formal parameter of f. The right-hand side of the definition is the operator `actuals` applied to a list of the actual parameters corresponding to the formal parameter in question, listed in the order in which the calls are numbered.

The above procedure is performed for every function in the program. One can easily verify that the algorithm, when applied to the Fibonacci function, gives the code we intuitively introduced earlier.

In order to understand how the resulting code can be executed, one can again think in terms of trees. Nodes in a tree can be identified by finite lists of natural numbers (*tags*): the root corresponds to the tag [0], the leftmost child of the root to $[1,0]$, the second child of the root to $[2,0]$, the leftmost child of the leftmost child of the root to $[1,1,0]$, etc. An execution model is established by considering the `actuals` and $\texttt{call}_i$ as tag-manipulation operators. Intuitively, $\texttt{call}_i$ augments a tag j by prefixing it with i. On the other hand, `actuals` takes the head i of a tag and uses it to select its i-th argument. Formally, the semantics equations, as introduced in [15], are:

$$\begin{aligned} \big(call_i(A)\big)_t &= A_{cons(i,t)} \\ \big(actuals(A_0,\ldots,A_{m-1})\big)_{cons(i,t)} &= (A_i)_t \\ \theta(A_0,\ldots,A_{n-1})_t &= \theta\big((A_0)_t,\ldots,(A_{n-1})_t\big) \end{aligned}$$

where $A, A_0, \ldots, A_{m-1}$ are expressions in the target language and θ is an n-ary constant symbol. Constants in the target language have the same value under any tag, e.g., $(2)_t = 2$, $\forall t$.

The evaluation of the program starts with an empty tag. One can actually verify that by using the above semantic equations, the program, when executed, gives the desired result (see Appendix A for an example). In fact, this technique has been used in the implenentations of the Lucid functional-dataflow language [4, 14], as well as in other Lucid-based systems [9, 8]. The difficulty of extending this technique to higher-order functions has been the main reason that Lucid was introduced as a first-order dataflow language [14].

2.2 The Higher-Order Case

The idea presented above has recently been extended by author Wadge [13] to apply to typed higher-order programs. In this and the subsequent sections, we refine the technique in [13] and examine its potential for the efficient implementation of functional languages.

The main idea of the generalized transformation is that an order-N functional program can first be transformed into an order-$(N-1)$ program, using a similar technique as the one for the first-order case. The same procedure can then be repeated for the new program, until we finally get a zero-order program (extended with appropriate operators).

The idea of tags is now more general: for a program of order N, a tag is an N-tuple of lists, where each list corresponds to a different order of the program. The code that results from the transformation can be executed following the same basic principles as in the first-order case. The above ideas are illustrated with the following simple second-order program:

$$
\begin{array}{l}
apply(sq, 2) \\
\texttt{where} \\
\quad apply(f, x) = f(x); \\
\quad sq(y) = y * y; \\
\texttt{end}
\end{array}
$$

The function *apply* is second-order because of its first argument. The generalized transformation, in its first stage, eliminates this argument:

$$
\begin{array}{l}
\texttt{call}_{00}\ apply(2) \\
\texttt{where} \\
\quad apply(x) = f(x); \\
\quad sq(y) = y * y; \\
\quad f = \texttt{actuals}_0(sq); \\
\texttt{end}
\end{array}
$$

The transformation for the higher-order case consists of a number of stages. Each stage corresponds to a different order that is eliminated from the program. Therefore, we use a different set of operators at each step ($\texttt{call}_0$, $\texttt{actuals}_0$ for the first step, $\texttt{call}_1$, $\texttt{actuals}_1$ for the second, and so on). We see that the program that resulted above is first-order; all the functions have zero-order arguments. The only exception is the definition of f, which is an equation between function expressions. We can easily change this by introducing an auxiliary variable z:

$$
\begin{array}{l}
\texttt{call}_{00}\ apply(2) \\
\texttt{where} \\
\quad apply(x) = f(x); \\
\quad sq(y) = y * y; \\
\quad f(z) = \big(\texttt{actuals}_0(sq)\big)(z); \\
\texttt{end}
\end{array}
$$

It is necessary to pass z inside the `actuals` before performing the next stage of the transformation. The semantics of the operators require that z should be "advanced" before entering the scope of `actuals`. This is done as follows:

$$
\begin{array}{l}
\texttt{call}_{00}\ apply(2) \\
\texttt{where} \\
\quad apply(x) = f(x); \\
\quad sq(y) = y * y; \\
\quad f(z) = \texttt{actuals}_0\big(sq(\texttt{call}_{00}(z))\big); \\
\texttt{end}
\end{array}
$$

This completes the first stage of the transformation. Now, we have a first-order program (extended with operators), and we can apply the technique for the first-order case, which gives the final program:

$$
\begin{array}{l}
\texttt{call}_{10}\big(\texttt{call}_{00}(\mathit{apply})\big) \\
\texttt{where} \\
\quad \mathit{apply} = \texttt{call}_{10}(f); \\
\quad sq = y * y; \\
\quad f = \texttt{actuals}_0\big(\texttt{call}_{10}(sq)\big); \\
\quad z = \texttt{actuals}_1(x); \\
\quad y = \texttt{actuals}_1\big(\texttt{call}_{00}(z)\big); \\
\quad x = \texttt{actuals}_1(2); \\
\texttt{end}
\end{array}
$$

In the execution model for a program of order N, tags are N-tuples of lists of natural numbers, and each list corresponds to a different order of the initial program (or equivalently, a different stage in the transformation). We will use the notation $\langle L_0, \ldots, L_{N-1} \rangle$ to denote a tag. The operators `call` and `actuals` can now be thought of as operations on these more complicated tags. More specifically, `call` now has two indices (we represent it by $\mathit{call}_i^{\mathit{Stage}}$). Given a tag, *Stage* is used in order to select the corresponding lists from the tag. The list is then prefixed with i and returned to the tag.

On the other hand, $\mathit{actuals}^{\mathit{Stage}}$ takes from the tag the list corresponding to *Stage*, uses its head i to select the i-th argument of `actuals`, and returns the tail of the list to the tag. The semantic equations are:

$$
\begin{aligned}
\big(\mathit{call}_i^{\mathit{Stage}}(A)\big)_{\langle L_0,\ldots,L_{\mathit{Stage}},\ldots,L_{N-1}\rangle} &= A_{\langle L_0,\ldots,\mathit{cons}(i,L_{\mathit{Stage}}),\ldots,L_{N-1}\rangle} \\
\big(\mathit{actuals}^{\mathit{Stage}}(A_0,\ldots,A_{m-1})\big)_{\langle L_0,\ldots,L_{\mathit{Stage}},\ldots,L_{N-1}\rangle} &= (A_{\mathit{head}(L_{\mathit{Stage}})})_{\langle L_0,\ldots,\mathit{tail}(L_{\mathit{Stage}}),\ldots,L_{N-1}\rangle}
\end{aligned}
$$

For n-ary constant symbols θ, the semantic equation is the same as in the first-order case (the only difference being that the tag is now more complicated). The new operators can therefore be viewed as a generalization of the operators for the first-order case. The evaluation of a program starts with an N-tuple that contains N empty lists, one for each order. Execution proceeds as in the first-order case, the only difference being that the appropriate list within the tuple is accessed every time. The exception of the *apply* program is given in Appendix B.

It should be noted that the above technique does not apply when the program contains functions with higher-order result types. In this case, a preprocessing phase should be used that eliminates higher-order result types by introducing extra arguments in the corresponding functions.

3 Implementation

An implementation of the proposed technique should focus on two important issues:

Efficient tag operations. A list-based implementation of tags would be prohibitively expensive.

Avoidance of recomputations. The value of an identifier under a specific context (tag) may be demanded many times during program execution. A scheme that avoids duplication of computational effort is necessary.

In the following, we describe the implementation architecture of our technique (Figure 2), giving particular emphasis on how it resolves the above two issues.

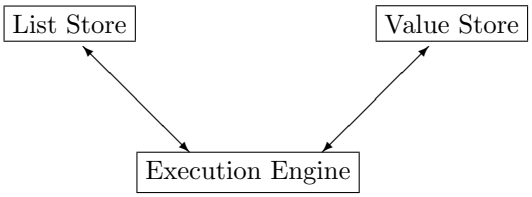


Figure 2: Structure of the implementation.

3.1 The List Store

The purpose of this component is to ensure a compact representation of tags as well as efficient operations on them. Recall that for an N-order functional program, tags are N-tuples $\langle L_0, \ldots, L_{N-1} \rangle$ of lists of natural numbers. The solution adopted is to convert each list into a natural number, using a technique known as "hash-consing" (described below). In this way, tags appear as N-tuples $\langle K_0, \ldots, K_{N-1} \rangle$ of natural numbers and their manipulation is much more convenient The *List-Store* component in Figure 2 takes care of the details of hash-consing and also supports functions that can be used to implement the `call` and `actuals` operators.

	head	**tailCode**	
1	0	0	[ ]
2	1	1	[1]
3	5	1	[5]
4	2	3	[2, 5]
5	1	4	[1, 2, 5]
⋮			

Figure 3: The hash-consing technique.

The idea of hash-consing is to store a list in a hash table as a pair (*head*, *position of tail*). The list is then represented by the position of the tuple in the table (Figure 3). The following primitive functions are supported by the List Store:

- *hashCons*(*head*, *tailCode*): Uses a hash function to check if the pair (*head*, *tailCode*) already exists in the hash table. If it does not, then it inserts it. Finally, it returns the position of the pair in the table.
- *hashHead*(*listCode*): It returns the first element of the pair found in the *listCode* position of the hash table.
- *hashTail*(*listCode*): It returns the second element of the pair found in the *listCode* position of the hash table.

The above operations can be performed very efficiently. They are used by the Execution Engine to implement the semantic equations of the generalized `call` and `actuals` operators described above. Moreover, the space occupied by the hash table is reasonable. Encoding lists as natural numbers using the above primitives allows a greater flexibility on tag manipulation.

3.2 The Value Store

During the execution of a program, many identical computations take place. The technique described in this paper has an inherent potential for avoiding unnecessary recalculations. More specifically, consider an identifier i whose value v has already been computed under a specific context (tag) t. The *Value Store* is a hash table whose purpose is to keep this kind of information.

An alternative way is to look at the Value Store as a generalized form of symbol table, suitable for storing the tree-like identifiers in §2.1. A schematic description is given in Figure 4. If the value of the identifier i with the same tag t is demanded again during program execution, then a lookup of the Value Store can potentially save significant time.

Ident	**Tag**	**Value**
x	$\langle 6, 39 \rangle$	20
y	$\langle 10, 8 \rangle$	100

Figure 4: The Value Store.

It is important to note that the Value Store is not a *required* component of the architecture. In fact, the only space that is actually essential for the technique is the table used for implementing hash-consing. However, our experience shows that the use of the Value Store gives an important benefit to the technique by drastically reducing the number of steps that have to be performed in order to evaluate a program. The size of the Value Store can be controlled in two ways:

Heuristics. A particularly successful one is the *retirement age scheme*, which has been used in Lucid implementations. It follows the same ideas as the "Least Recently Used" technique for performing page replacement in operating systems.

Analysis-Based Techniques. Compile-time analysis of the source program can be used to predict how long a specific entry needs to stay in the Value Store [7].

The efficiency of the above scheme can be drastically improved by performing *dimensionality analysis* at compile time. Space and time savings can result from determining the dependencies between identifiers and tag fields. Then one need only save, for each identifier, those components of the tag that are absolutely necessary.

3.3 The Execution Engine

The Execution Engine coordinates the actions of the List and Value components of the architecture. Its main purpose is to evaluate the code that results after applying to the source program the translation described in §2. This code has the form:

$$\begin{aligned} output &= def_0; \\ ident_1 &= def_1; \\ &\vdots \\ ident_r &= def_r; \end{aligned}$$

In other words, the code to be executed is a set of definitions of identifiers, with def_0 being the expression in the subject of the `where` clause of the translated program. In the following, we present the rules for a simple interpretative Execution Engine. We consider every case separately:

Identifiers. When an identifier $ident_k$ is encountered during the execution of the program, the following actions are performed. The identifier is first looked up in the Value Store under the current context t. If it is found, then the corresponding value is returned. Otherwise,

the corresponding definition def_k is used to compute its value in the current context. The result of the evaluation is inserted in the Value Store. Using pseudocode:

$\mathbf{EVAL}(ident_k, t) =$
  `if` the entry $(ident_k, t, value)$ is in the Value Store, `then`
    Return($value$);
  `else`
    $value = \mathbf{EVAL}(def_k, t)$;
    Insert $(ident_k, t, value)$ in the Value Store;
    Return($value$);
  `end`

It should be noted that compile-time analysis of the program may indicate that some indentifiers should never be stored in the Value Store because they are never going to be demanded. The above code can be modified accordingly to reflect these optimizations.

`call` **operator.** When the expression $call_i^{Stage}(A)$ is encountered during execution, then the evaluator performs the following. The current tag t is transformed according to the semantic definition of the `call` operator. Then the expression A is evaluated under the new tag. More specifically:

$$\mathbf{EVAL}\big(call_i^{Stage}(A), t\big) = \mathbf{EVAL}\big(A, tagCons(t, i, Stage)\big)$$

where the function *tagCons* performs the following operation on tags:

$$tagCons\big(\langle K_0, \ldots, K_{Stage}, \ldots, K_{N-1}\rangle, i, Stage\big) = \big\langle K_0, \ldots, hashCons(i, K_{Stage}), \ldots, K_{N-1}\big\rangle$$

In other words, *tagCons* uses the *hashCons* operation supported by the List Store, in order to prefix the *Stage*-th element of the current tag with i.

`actuals` **operator.** To evaluate $actuals^{Stage}(A_0, \ldots, A_{m-1})$ the following actions are required. The *Stage*-th element of the current tag is selected, its head is extracted and used to select the corresponding argument of `actuals`. This argument is then evaluated under the new tag. Formally:

$$\mathbf{EVAL}\big(actuals^{Stage}(A_0, \ldots, A_{m-1}), t\big) = \mathbf{EVAL}\big(A_{tagHead(t,Stage)}, tagTail(t, Stage)\big)$$

where the functions *tagHead* and *tagTail* perform the following operations on tags:

$$tagHead\big(\langle K_0, \ldots, K_{Stage}, \ldots, K_{N-1}\rangle, i, Stage\big) = hashHead(K_{Stage})$$
$$tagTail\big(\langle K_0, \ldots, K_{Stage}, \ldots, K_{N-1}\rangle, i, Stage\big) = \big\langle K_0, \ldots, hashTail(K_{Stage}), \ldots, K_{N-1}\big\rangle$$

Binary operators. The implementation of binary operators (addition, multiplication, etc.) is straightforward:

$$\mathbf{EVAL}(E_1 + E_2, t) = \mathbf{EVAL}(E_1, t) + \mathbf{EVAL}(E_2, t)$$

Conditional. The implementation of the conditional is also immediate:

$$\mathbf{EVAL}(\texttt{if } B \texttt{ then } E_1 \texttt{ else } E_2, t) = \texttt{if } \mathbf{EVAL}(B, t) \texttt{ then } \mathbf{EVAL}(E_1, t) \texttt{ else } \mathbf{EVAL}(E_2, t)$$

All the other features of the source language can be implemented using the same principles. Execution of a program starts by invoking

$$\mathbf{EVAL}(output, initialTag)$$

The *initialTag* is an N-tuple of the form $\langle \epsilon, \ldots, \epsilon\rangle$, where ϵ is the encoding using hash-consing of the empty list. For example, if the table of Figure 3 is used, then $\epsilon = 1$.

4 Properties of the Technique

In this section, we review the main characteristics and present some of the novel features that are inherent in our approach.

4.1 General Characteristics

The proposed method presents some important differences compared to the reduction-based implementation. We summarize them below:

- In graph reduction, expressions are evaluated by performing transformations on the program graph until a canonical form is obtained. In other words, the program changes during execution. This is not the case with our approach: the program is read-only, and this can be verified by looking at the way the Execution Engine works. Instead, the tags are used in order to reflect the current context of the computation.
- The powerful tagging scheme that forms the backbone of the proposed technique gives rise to an efficient memoization mechanism. In fact, the use of the Value Store proves indispensable: the number of steps that have to be performed in order to execute a program is considerably less than by using graph reduction. Moreover, as it is is shown in the next section, the space consumption is comparable and in many cases better than graph reduction.

4.2 Self-Optimizing Properties

One important impediment to the efficient implementation of functional languages results from the fact that one cannot know in advance whether an expression passed as an argument to a function will be evaluated. An inherent property of the technique is that it tends to isolate those arguments that will not be needed during execution. In this way, the unnecessary components of the program do not put any burden (neither time nor space) on program efficiency. This point can be illustrated with the following example:

$$
\begin{array}{l}
f(100,1) \\
\texttt{where} \\
\quad f(x,y) = \texttt{if } x < 1 \texttt{ then } 0 \texttt{ else } f(x-1, f(x, y+1)); \\
\texttt{end}
\end{array}
$$

The parameter y plays no rôle during execution and should normally be ignored. This is indeed reflected by the code that results after the translation:

$$
\begin{array}{l}
\texttt{call}_{00}(f) \\
\texttt{where} \\
\quad f = \texttt{if } x < 1 \texttt{ then } 0 \texttt{ else } \texttt{call}_{01}(f); \\
\quad x = \texttt{actuals}_0(100, x-1, x); \\
\quad y = \texttt{actuals}_0\big(1, \texttt{call}_{02}(f), y+1\big); \\
\texttt{end}
\end{array}
$$

The parameter y never references in the rest of the program. Therefore, its definition remains isolated during execution and does not affect in any way the performance of the program. In fact, the above program runs using *exactly* the same resources as the simpler one:

$$
\begin{array}{l}
f(100) \\
\texttt{where} \\
\quad f(x) = \texttt{if } x < 1 \texttt{ then } 0 \texttt{ else } f(x-1); \\
\texttt{end}
\end{array}
$$

The above behavior is a direct consequence of the fact that the proposed implementation technique is a *fixed-program* machine. After the translation, formal parameters of functions become definitions, and these definitions are not accessed unless it is absolutely necessary.

4.3 Relationship with Dataflow

Traditionally, there has been a very close relationship between the dataflow model of computation and functional programming. This is evidenced by the fact that all the well-known dataflow languages are functional in nature [1]. Consequently, it would normally be expected that prominent features such as higher-order functions would have been embedded coherently in the dataflow framework. However, this is not the case: higher-order functions, if available, are implemented using closures and an expensive *apply* schema [2]. In other words, non-dataflow features are adopted, a fact that results in performance degradation when it comes to higher-order functions.

Our approach gives a direct solution to this problem. Tags have traditionally been used in dataflow architectures but have never been generalized for higher-order functions. In this sense, the work presented here can be considered as a purely dataflow solution to the implementation of lazy higher-order functional languages.

4.4 Stream-Based Languages

The technique described in this paper can be easily extended to support stream-based languages such as Lucid. Streams as used by Lucid are different in nature to lazy lists. As an example, one can add two streams s_1 and s_2 together by writing $s_1 + s_2$, and this defines the pointwise addition of the two streams. Using lists, one has to explicitly define a function *plus* that first adds the heads of the lists and then manipulates the tails.

To extend the technique, tags are generalized to contain a time parameter which intuitively marks the position of the stream where we currently are. The advantage of this approach is that one can actually access elements of the stream very efficiently. This is achieved using the hashing scheme supported by the Value Store. In this way we have a form of potentially *infinite* lazy arrays. As a result, one can get significant performance benefits (see the next section) for algorithms that require random access, without sacrificing the notational simplicity of the corresponding programs. This suggests a paradigm of functional programming that uses lists whenever sequential access is important and streams when fast random access is essential.

5 Performance Results

This section presents performance results obtained from two different implementations of the proposed technique. The former is an interpreter, whose Execution Engine is very similar to the **EVAL** function presented in §3.3. The latter is a compiler, which when given a program produces a "customized" Execution Engine that reflects the logic of the program.

The List and the Value Stores are identical in both implementations. The List Store is a *closed* hash table and collisions are resolved using *linear probing* [11, Algorithm L]. In other words, whenever a collision occurs, the next position of the table is examined. The Value-Store implementation uses an open scheme [11, pp.513–4] in which collisions are resolved using chaining.

Both systems have been implemented in `C` and the compiler produces `C` code as its output. In the following, we give comparative results with two existing and widely used systems. The programs used are the following:

Fibonacci. A standard benchmark for functional programming-language implementations.

Integration. A second-order program. Given a real function f and an interval $[a, b]$, it estimates the area under the curve that f defines between the two points. This is done by subdividing $[a, b]$ in $b - a$ subintervals and computing the area in each one of them. The results below are for the square function in the interval $[0, 10\,000]$.

Lazy. The program of §4.2. It tests the overhead involved in programs where not every argument is used in every call of a function. The function was invoked with x equal to $50\,000$.

Convolution. Performs convolution of two sequences of natural numbers. It requires non-sequential access to the elements of the two sequences. It is used to demonstrate the performance of the technique for stream-based functional languages. The first 100 elements of the convoluted sequence are produced.

5.1 The Interpreter

In this section, we give a performance comparison with the Miranda[1] interpreter. Although Miranda is clearly slower than some of the recently developed compilers, it is one of the fastest and most robust interpreters for lazy functional languages. Moreover, the comparison will give an idea of the performance of our technique against combinator reduction [12]. The following table gives the timing results for the two techniques on a Sun4 server:

Time (in sec)		
Program	Interpreter	Miranda
Nfib 24	10.5	25.0
Integration	5.2	8.8
Lazy	3.5	10.6
Convolution	4.4	83.2

Clearly, our technique performs better than the combinator-based approach. Similar figures have been obtained for other programs we have tried. A more realistic measure of performance is the number of steps taken in order to execute a program. For Miranda, a step is equivalent to a reduction. For our implementation we count the number of steps that the interpreter performs (including accesses to the Value Store). The following table indicates that the number of steps is significantly smaller for our approach. Moreover, if one considers the execution times, it appears that steps in our case are more fine-grained.

Number of Steps		
Program	Interpreter	Miranda
Nfib 24	1 350 440	1 650 541
Integration	440 012	660 023
Lazy	400 008	800 017
Convolution	137 852	3 991 225

Finally, we give a space comparison between the two approaches. Surprisingly, our approach consumes considerably less space than Miranda. The figures reported for Miranda show the number of cells claimed from the store manager during execution. In our case, we add the space consumed by hash-consing and the space occupied by all the entries inserted in the Value Store during execution. More specifically, space is calculated using

$$2 * a + (3 + N) * b$$

where a is the number of the occupied entries of the hash-consing table, N is the order of the program and b is the number of Value-Store entries. Each entry of the hash-consing table consists of two fields (head and tail), and each entry of the Value Store consists of $3 + N$ fields (identifier, value, N fields for the tag and a pointer to the next entry). For the convolution program, the time field was taken into account as well.

Space		
Program	Interpreter	Miranda
Nfib 24	900 294	1 650 553
Integration	460 014	610 047
Lazy	300 006	967 256
Convolution	103 216	6 465 715

[1] Miranda is a trademark of Research Software Ltd.

The above results indicate that the technique performs better than Miranda in execution time, steps taken and space used.

5.2 The Compiler

In this section, we give a performance comparison of our technique with the LML compiler [5]. The LML system is based on the G-machine [6, 10], and it is one of the fastest compilers of lazy functional languages available. The following are the timing results for the given programs on a Sun SPARCserver 690MP:[2]

Time (in sec)			
Program	Compiler	LML	LML (not strict)
Nfib 24	2.5	0.5	0.8
Integration	1.7	0.8	0.8
Lazy	0.9	1.5	1.5
Convolution	0.6	1.2	2.1

For a more fair comparison, we have give the results of LML with and without strictness analysis. Our system does not currently support strictness analysis, but similar speedups are excepted. The above timings indicate that our implementation performs particularly well in cases where laziness or random access of list elements is important. Moreover, the timings for the other two programs are satisfactory, especially for the second-order Integration benchmark.

Steps during execution are not reported by the two systems, as it is difficult to obtain this kind of statistic from compiled code. However, we believe that our technique requires fewer steps in general. This can be evidenced by the fact that the two systems run comparably, although the code that our compiler produces does not have any sophisticated optimizations.

Finally, the space consumed by the two compilers is given below. As the LML system reports the heap space used in MB, this is the representation we adopt.

Space (in MB)			
Program	Compiler	LML	LML (not strict)
Nfib 24	3.3	3.0	3.6
Integration	2.0	1.3	1.4
Lazy	1.1	2.8	3.0
Convolution	0.3	3.1	4.5

Again, the space consumption is quite satisfactory compared with LML. It is interesting to note the difference in the Lazy program. The unnecessary overhead creates a heavy argument. On the other hand, our system performs worse in the higher-order program. This is a result of the fact that our implementation does not currently perform any optimizations in the way it stores the tags for higher-order programs.

5.3 General Comments

The above results indicate that our implementations run comparably (if not favorably) with the two reduction-based approaches. The results become more important if one takes into account the fact that both Miranda and LML have been the result of years of research and development and both use sophisticated optimizations. Mirand uses a number of techniques that produce more compact combinator code [12]. LML uses many "source-to-source" transformations to get a program for which generation of efficient code is less complicated. On the other hand, our implementation uses only some simple optimizations. We expect a considerable improvement in execution speed as a result of further research in this direction.

[2]Miranda and LML were not available on the same machine.

6 Conclusions

A new implementation technique for lazy typed functional languages has been presented. In contrast with graph-reduction approaches, it is a fixed-program machine. This fact minimizes and in some cases completely eliminates the overhead required for passing unevaluated arguments to functions. Moreover, the technique can be conveniently implemented on dataflow architectures.

Future work includes investigating optimizations for the intermediate code that the technique produces. Also, a more efficient implementation of the Value Store is required, in order to ensure very fast lookups and insertions.

Acknowledgements

We would like to thank B. Knight for useful discussions and suggestions. This research was supported by the Natural Sciences and Engineering Research Council of Canada and by a University of Victoria Fellowship.

References

[1] William B. Ackerman. Data flow languages. *IEEE Computer*, 15(2):15–25, 1982.

[2] Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Computers*, 39(3):300–318, 1990.

[3] E. A. Ashcroft, A. A. Faustini, and R. Jagannathan. An intensional language for parallel applications programming. pages 11–49, 1991.

[4] Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20(7):519–526, 1977.

[5] L. Augustsson and T. Johnsson. Lazy ML user's manual. Technical report, Department of Computer Science, Chalmers University of Technology, Sweden, 1992.

[6] Lennart Augustsson. A compiler for Lazy ML. In *LISP and Functional Programming*, pages 218–227, 1984.

[7] R. Bagai. Compilation of the dataflow language Lucid. Master's thesis, Department of Computer Science, University of Victoria, Canada, 1986.

[8] Weichang Du and William W. Wadge. A 3D spreadsheet based on intensional logic. *IEEE Software*, 7(3):78–89, 1990.

[9] Weichang Du and William W. Wadge. The eductive implementation of a three-dimensional spreadsheet. *Softw., Pract. Exper.*, 20(11):1097–1114, 1990.

[10] Thomas Johnsson. Efficient compilation of lazy evaluation. In *SIGPLAN Symposium on Compiler Construction*, pages 58–69. ACM, 1984.

[11] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison Wesley, 1975.

[12] D. A. Turner. A new implementation technique for applicative languages. *Softw., Pract. Exper.*, 9(1):31–49, 1979.

[13] William W. Wadge. Higher-order Lucid. In *4th International Symposium on Lucid and Intensional Programming*, Menlo Park, CA. SRI International.

[14] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, London, 1985.

[15] A. A. Yaghi. *The Intensional Implementation Technique for Functional Languages.* PhD thesis, Department of Computer Science, University of Warwick, UK, 1984.

A First-Order Programs: Example Execution

The following is the execution sequence for computing $\mathit{fib}(2)$. For clarity, we include the translated program:

$$
\begin{array}{l}
\texttt{call}_0\ \mathit{fib} \\
\texttt{where} \\
\quad \mathit{fib} = \texttt{if}\ n < 2\ \texttt{then}\ 1\ \texttt{else}\ (\texttt{call}_1\ \mathit{fib}) + (\texttt{call}_2\ \mathit{fib}); \\
\quad n = \texttt{actuals}(2, n-1, n-2); \\
\texttt{end}
\end{array}
$$

The execution proceeds as follows:

$$
\begin{array}{rl}
 & \mathbf{EVAL}\big(\texttt{call}_0(\mathit{fib}), [\,]\big) \\
= & \mathbf{EVAL}\big(\mathit{fib}, [0]\big) \\
= & \mathbf{EVAL}\big(\texttt{if}\ n < 2\ \texttt{then}\ 1\ \texttt{else}\ \texttt{call}_1(\mathit{fib}) + \texttt{call}_2(\mathit{fib}), [0]\big) \\
= & \texttt{if}\ \mathbf{EVAL}\big(n < 2, [0]\big)\ \texttt{then}\ 1\ \texttt{else}\ \mathbf{EVAL}\big(\texttt{call}_1(\mathit{fib}) + \texttt{call}_2(\mathit{fib}), [0]\big) \\
= & \texttt{if}\ \mathbf{EVAL}\big(\texttt{actuals}(2, n-1, n-2), [0]\big) < 2\ \texttt{then}\ 1 \\
 & \texttt{else}\ \mathbf{EVAL}\big(\texttt{call}_1(\mathit{fib}) + \texttt{call}_2(\mathit{fib}), [0]\big) \\
= & \texttt{if}\ 2 < 2\ \texttt{then}\ 1\ \texttt{else}\ \mathbf{EVAL}\big(\texttt{call}_1(\mathit{fib}) + \texttt{call}_2(\mathit{fib}), [0]\big) \\
= & \mathbf{EVAL}\big(\texttt{call}_1(\mathit{fib}) + \texttt{call}_2(\mathit{fib}), [0]\big) \\
= & \mathbf{EVAL}\big(\texttt{call}_1(\mathit{fib}), [0]\big) + \mathbf{EVAL}\big(\texttt{call}_2(\mathit{fib}), [0]\big)
\end{array}
$$

Now, the two computations can proceed independently. The first one gives the following execution sequence:

$$
\begin{array}{rl}
 & \mathbf{EVAL}\big(\texttt{call}_1(\mathit{fib}), [0]\big) \\
= & \mathbf{EVAL}\big(\texttt{if}\ n < 2\ \texttt{then}\ 1\ \texttt{else}\ \texttt{call}_1(\mathit{fib}) + \texttt{call}_2(\mathit{fib}), [1, 0]\big) \\
= & \texttt{if}\ \mathbf{EVAL}\big(\texttt{actuals}(2, n-1, n-2), [1, 0]\big) < 2\ \texttt{then}\ 1 \\
 & \texttt{else}\ \mathbf{EVAL}\big(\texttt{call}_1(\mathit{fib}) + \texttt{call}_2(\mathit{fib}), [1, 0]\big) \\
= & \texttt{if}\ \mathbf{EVAL}\big(n-1, [0]\big) < 2\ \texttt{then}\ 1\ \texttt{else}\ \mathbf{EVAL}\big(\texttt{call}_1(\mathit{fib}) + \texttt{call}_2(\mathit{fib}), [1, 0]\big) \\
= & \texttt{if}\ \big(\mathbf{EVAL}\big(n, [0]\big) - 1\big) < 2\ \texttt{then}\ 1\ \texttt{else}\ \mathbf{EVAL}\big(\texttt{call}_1(\mathit{fib}) + \texttt{call}_2(\mathit{fib}), [1, 0]\big) \\
= & \texttt{if}\ \big(\mathbf{EVAL}\big(\texttt{actuals}(2, n-1, n-2), [0]\big) - 1\big) < 2\ \texttt{then}\ 1 \\
 & \texttt{else}\ \mathbf{EVAL}\big(\texttt{call}_1(\mathit{fib}) + \texttt{call}_2(\mathit{fib}), [1, 0]\big) \\
= & \texttt{if}\ (2-1) < 2\ \texttt{then}\ 1\ \texttt{else}\ \mathbf{EVAL}\big(\texttt{call}_1(\mathit{fib}) + \texttt{call}_2(\mathit{fib}), [0]\big) \\
= & 1
\end{array}
$$

The second computation proceeds in a similar way.

$$
\begin{aligned}
& \mathbf{EVAL}\big(\mathtt{call}_2(\mathit{fib}), [0]\big) \\
= \;& \mathbf{EVAL}\big(\mathtt{if}\ n < 2\ \mathtt{then}\ 1\ \mathtt{else}\ \mathtt{call}_1(\mathit{fib}) + \mathtt{call}_2(\mathit{fib}), [2,0]\big) \\
= \;& \mathtt{if}\ \mathbf{EVAL}\big(\mathtt{actuals}(2, n-1, n-2), [2,0]\big) < 2\ \mathtt{then}\ 1 \\
& \mathtt{else}\ \mathbf{EVAL}\big(\mathtt{call}_1(\mathit{fib}) + \mathtt{call}_2(\mathit{fib}), [2,0]\big) \\
= \;& \mathtt{if}\ \mathbf{EVAL}\big(n-2, [0]\big) < 2\ \mathtt{then}\ 1\ \mathtt{else}\ \mathbf{EVAL}\big(\mathtt{call}_1(\mathit{fib}) + \mathtt{call}_2(\mathit{fib}), [2,0]\big) \\
= \;& \mathtt{if}\ \big(\mathbf{EVAL}\big(n, [0]\big) - 2\big) < 2\ \mathtt{then}\ 1\ \mathtt{else}\ \mathbf{EVAL}\big(\mathtt{call}_1(\mathit{fib}) + \mathtt{call}_2(\mathit{fib}), [2,0]\big) \\
= \;& \mathtt{if}\ \big(\mathbf{EVAL}\big(\mathtt{actuals}(2, n-1, n-2), [0]\big) - 2\big) < 2\ \mathtt{then}\ 1 \\
& \mathtt{else}\ \mathbf{EVAL}\big(\mathtt{call}_1(\mathit{fib}) + \mathtt{call}_2(\mathit{fib}), [2,0]\big) \\
= \;& \mathtt{if}\ (2-2) < 2\ \mathtt{then}\ 1\ \mathtt{else}\ \mathbf{EVAL}\big(\mathtt{call}_1(\mathit{fib}) + \mathtt{call}_2(\mathit{fib}), [0]\big) \\
= \;& 1
\end{aligned}
$$

The output of the evaluator is obviously the sum of the results of the two subcomputations. It is interesting to note the fact that the second subcomputation demands at some point the value of $\mathbf{EVAL}(n, [0])$. However, this has already been computed, and if it has been saved, it need not be recalculated.

B Higher-Order Programs: Example Execution

The following is the execution sequence for computing $\mathit{apply}(\mathit{sq}, 2)$. For clarity, we include the translated program:

$$
\begin{aligned}
& \mathtt{call}_{10}\big(\mathtt{call}_{00}(\mathit{apply})\big) \\
& \mathtt{where} \\
& \quad \mathit{apply} = \mathtt{call}_{10}(f); \\
& \quad \mathit{sq} = y * y; \\
& \quad f = \mathtt{actuals}_0\big(\mathtt{call}_{10}(\mathit{sq})\big); \\
& \quad z = \mathtt{actuals}_1(x); \\
& \quad y = \mathtt{actuals}_1\big(\mathtt{call}_{00}(z)\big); \\
& \quad x = \mathtt{actuals}_1(2); \\
& \mathtt{end}
\end{aligned}
$$

The execution proceeds as follows:

$$
\begin{aligned}
& \mathbf{EVAL}\big(\mathtt{call}_{10}(\mathtt{call}_{00}(\mathit{apply})), \langle [\,], [\,] \rangle\big) \\
= \;& \mathbf{EVAL}\big(\mathtt{call}_{00}(\mathit{apply}), \langle [\,], [0] \rangle\big) \\
= \;& \mathbf{EVAL}\big(\mathit{apply}, \langle [0], [0] \rangle\big) \\
= \;& \mathbf{EVAL}\big(\mathtt{call}_{10}(f), \langle [0], [0] \rangle\big) \\
= \;& \mathbf{EVAL}\big(f, \langle [0], [0,0] \rangle\big) \\
= \;& \mathbf{EVAL}\big(\mathtt{actuals}_0(\mathtt{call}_{10}(\mathit{sq})), \langle [0], [0,0] \rangle\big) \\
= \;& \mathbf{EVAL}\big(\mathtt{call}_{10}(\mathit{sq}), \langle [\,], [0,0] \rangle\big) \\
= \;& \mathbf{EVAL}\big(\mathit{sq}, \langle [\,], [0,0,0] \rangle\big) \\
= \;& \mathbf{EVAL}\big(y * y, \langle [\,], [0,0,0] \rangle\big) \\
= \;& \mathbf{EVAL}\big(y, \langle [\,], [0,0,0] \rangle\big) * \mathbf{EVAL}\big(y, \langle [\,], [0,0,0] \rangle\big)
\end{aligned}
$$

Now, the two identical computations can proceed independently and they both have the same execution sequence:

$$
\begin{aligned}
& \mathbf{EVAL}\big(y, \langle [\,], [0,0,0]\rangle\big) \\
= \;& \mathbf{EVAL}\big(\mathtt{actuals}_1(\mathtt{call}_{00}(z)), \langle [\,], [0,0,0]\rangle\big) \\
= \;& \mathbf{EVAL}\big(\mathtt{call}_{00}(z), \langle [\,], [0,0]\rangle\big) \\
= \;& \mathbf{EVAL}\big(z, \langle [0], [0,0]\rangle\big) \\
= \;& \mathbf{EVAL}\big(\mathtt{actuals}_1(x), \langle [0], [0,0]\rangle\big) \\
= \;& \mathbf{EVAL}\big(x, \langle [0], [0]\rangle\big) \\
= \;& \mathbf{EVAL}\big(\mathtt{actuals}_1(2), \langle [0], [0]\rangle\big) \\
= \;& \mathbf{EVAL}\big(2, \langle [0], [\,]\rangle\big) \\
= \;& 2
\end{aligned}
$$

The final result is the product of the results of the two subcomputations and equals 4.