Higher-order Dataflow and its Implementation on Stock Hardware

Panos Rondogiannis        William W. Wadge

Abstract

An important shortcoming of current dataflow implementations of functional languages is the treatment of higher-order functions. The solution usually adopted is through the use of closures. However, such an approach, besides being expensive, does not seem to harmonize with the basic principles of tagged dataflow. This paper presents a technique for implementing higher-order lazy functional languages using only simple dataflow concepts. The proposed approach, when implemented on stock hardware, competes with the modern reduction-based implementations of functional languages.

Keywords: Higher-order dataflow, Functional programming, Intensional programming.

1 Introduction

There has traditionally been a 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]. As a result, it would 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 non-dataflow concepts such as closures [3]. This results in performance degradation for higher-order programs.

This paper considers the implementation of lazy higher-order functional languages, using only simple dataflow concepts (such as tags). Given a program of order \( N \), the technique gradually transforms it into a zero-order program extended with appropriate context (tag) manipulation operators. We describe the semantics of these operators and present a machine architecture that ensures efficient execution of the resulting code.

2 The First-order Case

Before considering higher-order programs, we outline the approach we adopt for the first-order case; this was initially developed in [18] and also described in [4]. The algorithm given in [18] transforms a first-order program into a set of zero-order definitions that contain context-manipulation operations. As the semantics of the resulting code is based on Montague’s Intensional Logic [14], the resulting definitions are also referred in [18] as intensional definitions.

The functional language adopted in [18] is ISWIM [11]. Programs are initially flattened using a technique similar to \( \lambda \)-lifting [8]. The algorithm (Figure 1) is then applied to this flattened code. As an example, consider the following program:

\[
\begin{align*}
\text{where} & \\
& f(x) = g(x + 1); \\
& g(y) = y \ast y;
\end{align*}
\]

\*

1. Let $f$ be a function appearing in the program. Number the textual occurrences of alls to $f$ starting at 0 (including calls in the body of the definition of $f$).

2. Replace the $i$-th call of $f$ by $\text{call}_i(f)$.

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 $\text{actuals}$ 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.

---

Figure 1: Algorithm for First-order Programs

The following zero-order intensional program is obtained when the algorithm is applied.

```plaintext
call_0(f)
where
\begin{align*}
  f &= \text{call}_0(g); \\
  g &= y \ast y; \\
  x &= \text{actuals}(5); \\
  y &= \text{actuals}(x + 1);
\end{align*}
end
```

An execution model is established by considering the $\text{call}_i$ and $\text{actuals}$ as operations on finite lists of natural numbers (referred to from now on as tags or contexts). Execution of the program starts by demanding the value of the subject of the $\text{where}$ clause of the intensional program, under the empty tag $[]$. The operator $\text{call}_i$ augments a tag $j$ by prefixing it with $i$. On the other hand, $\text{actuals}$ takes the head $i$ of a tag and uses it to select its $i$-th argument. Formally, the semantic equations, as introduced in [18], are:

\[
\begin{align*}
  (\text{call}_i(A)) &= A_{\text{cons}(i,t)} \\
  (\text{actuals}(A_0, \ldots, A_{m-1}))_{\text{cons}(i,t)} &= (A_i)_t \\
  \theta(A_1, \ldots, A_m)_t &= \theta((A_1)_t, \ldots, (A_m)_t)
\end{align*}
\]

where $A, A_0, \ldots, A_m$ are expressions in the target (intensional) language and $\theta$ is an $m$-ary operation symbol. Constants in the target language have the same value under any tag, e.g., $(2)_t = 2, \forall t$.

In Figure 2, the execution of the program is given using the above semantic rules. The technique just described has been used in the implementations of the Lucid functional dataflow language [17], as well as in Lucid-based systems [7].

3 The Higher-order Case

In this section, we extend the technique described above to apply to higher-order programs. The ideas presented below extend and refine previous results obtained by one of the authors [16]. The language we adopt must satisfy the following two requirements.

1. Functions have finite types.

2. Functions have first-order result types.

The first of the above requirements excludes functions such as $\text{selfapply}(f) = f(f)$ that do not have finite order. The second requirement disallows functions of the form $f(x) = \text{sq}$, where $\text{sq}$ is a
The evaluation of the left-hand side of the product begins:

\[
\text{EVAL}(y, [0, 0])
\]
\[
\text{EVAL(\text{actuals}(x + 1), [0, 0])}
\]
\[
\text{EVAL}(x + 1, [0])
\]
\[
\text{EVAL}(x, [0]) + 1
\]
\[
\text{EVAL(\text{actuals}(5), [0])} + 1
\]
\[
\text{EVAL}(5, [1]) + 1
\]
\[
6
\]

The right-hand side of the product can be evaluated in a similar way. However, the second calculation need not be performed if the value of \(y\) under the tag \([0, 0]\) is appropriately saved the first time.

Figure 2: Execution of First-order Programs

function identifier. However, with appropriate preprocessing of the program, this restriction can in general be lifted.

The main idea of the generalized transformation is that an order-\(N\) functional program can first be transformed into an order-(\(N - 1\)) intensional 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 intensional program.

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. In the following, we present the transformation algorithm, describe the semantics of the generalized operators and illustrate the execution of the resulting code. Consider the following simple second-order program.

\[
\begin{align*}
\text{apply}(sq, 2) \\
\text{where} \\
\text{apply}(f, x) &= f(x); \\
\text{sq}(y) &= y \ast y; \\
\end{align*}
\]

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

\[
\begin{align*}
call_{00} \text{ apply}(2) \\
\text{where} \\
\text{apply}(x) &= f(x); \\
\text{sq}(y) &= y \ast y; \\
f &= \text{actuals}_{0}(sq); \\
\end{align*}
\]
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. For the first step, we use the operators \texttt{actuals}_0 and \texttt{call}_i, where \( i \) ranges as in the first-order case. For the second step, we use \texttt{actuals}_1 and \texttt{call}_i, 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 \( z \):

\[
\begin{align*}
\text{call}_{00} \quad & \text{apply}(2) \\
\text{where} \\
& \text{apply}(x) = f(x); \\
& sq(y) = y \ast y; \\
& f(z) = (\text{actuals}_0(sq))(z); \\
\end{align*}
\]

It is necessary to pass \( z \) inside the \texttt{actuals} before performing the next stage of the transformation. However, the \texttt{actuals} operator alters the tags. In order for \( z \) to be evaluated in the outer tag, it has to be advanced before entering the scope of the \texttt{actuals}. This is done as follows:

\[
\begin{align*}
\text{call}_{00} \quad & \text{apply}(2) \\
\text{where} \\
& \text{apply}(x) = f(x); \\
& sq(y) = y \ast y; \\
& f(z) = \text{actuals}_0(sq(\text{call}_{00}(z))); \\
\end{align*}
\]

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

\[
\begin{align*}
\text{call}_{10} \quad & (\text{call}_{00} (\text{apply}i)) \\
\text{where} \\
& \text{apply} = \text{call}_{10} f; \\
& sq = y \ast y; \\
& f = \text{actuals}_0(\text{call}_{10} (sq)); \\
& z = \text{actuals}_1(x); \\
& y = \text{actuals}_1(\text{call}_{00}(z)); \\
& x = \text{actuals}_1(2); \\
\end{align*}
\]

The algorithm for the higher-order case is shown in Figure 3. 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 \( \langle t_0, \ldots, t_{N-1} \rangle \) to denote a tag. The operators \texttt{call} and \texttt{actuals} can now be thought of as operations on these more complicated tags. More specifically, \texttt{call} now has two indices (we represent it by \texttt{call}^s_i). Given a tag, \( s \) is used in order to select the corresponding list from the tag. The list is then prefixed with \( i \) and returned to the tag.

On the other hand, \texttt{actuals}^s takes from the tag the list corresponding to \( s \), uses its head \( i \) to select the \( i \)-th argument of \texttt{actuals}, and returns the tail of the list to the tag. The new semantic equations are:

\[
\begin{align*}
(\texttt{call}^s_i(A))_{\langle t_0, \ldots, t_i, \ldots, t_{N-1} \rangle} &= A_{\langle t_0, \ldots, \text{cons}(i, t_i), \ldots, t_{N-1} \rangle} \\
(\texttt{actuals}^s(A_0, \ldots, A_m))_{\langle t_0, \ldots, t_i, \ldots, t_{N-1} \rangle} &= (A_{\text{head}(t_i)})_{\langle t_0, \ldots, \text{tail}(t_i), \ldots, t_{N-1} \rangle}
\end{align*}
\]
Let $s = 0$. Increment $s$ by one every time the order of the program is reduced by one. Repeat the following steps until the program becomes zero-order:

1. Let $f$ be a function of the current highest order. Number the textual occurrences of calls to $f$ starting at 0 (including calls in the body of the definition of $f$).

2. Replace the $i$-th call of $f$ by $\text{call}^i_{\text{act}}(f(\text{Act}))$, where $\text{Act}$ is the list of actual parameters of $f$ with the parameters corresponding to the current highest order removed.

3. Remove from the definition of $f$ the formal parameters that correspond to the current highest order.

4. For every formal parameter $F_p$ of $f$ that was eliminated, introduce a definition as in the first-order case.

5. Introduce new variables $z_1, \ldots, z_n$ according to the type of $F_p$. Add the variables to both sides of the definition of $F_p$, advancing them before they enter the scope of the $\text{actuals}$.

Figure 3: Algorithm for Higher-order Programs

4 Implementation on Stock Hardware

Modern dataflow architectures [3, 6] are ideal candidates for executing the code that results from the transformation of the previous section; this is because tag manipulation is an essential component of their design. In this paper, we do not pursue this issue. Instead, we are considering how the intensional code can be compiled for conventional von Neumann architectures.

It turns out that tags can be embedded in traditional activation records (ARs) in such a way that lazy evaluation is ensured. When an expression of the form $\text{call}_i(f)$ is encountered during execution, an activation record is allocated. For every formal parameter of $f$, there exists a corresponding field in the AR. However, these fields are initially left empty: they will later receive a value only if at some point during execution, the corresponding formal parameter is demanded and calculated. In this sense, ARs are filled in a lazy way. They are disposed as usual, as soon as a function returns control.

Tags play the most crucial rôle in the proposed technique: they act as pointers to activation records. When the value of an identifier is demanded under a specific tag, we check whether the activation record pointed to by the tag already contains a computed result. If yes, the result is reused and no recalculations are necessary. Otherwise, the definition for the identifier is used to calculate the required value. The value is subsequently placed in its position in the activation record. In the following, the above ideas are described in detail.

5 Implementation for First-order Programs

To illustrate the proposed technique for the first-order case, consider the execution of the program given in Figure 2. Notice that the formal parameter $x$ of $f$ is demanded at some point under the tag $[0]$. This demand for $x$ is directly related to the demand for $f$ under the same tag $[0]$. In general, whenever a formal parameter (like $x$) is demanded under a tag $t$, there has existed a previous demand, under the same tag, for the function variable (like $f$) in which the formal
The evaluation of the left-hand side of the program begins:

\[
\begin{align*}
&\text{EVAL}(y, \langle [], [0, 0, 0] \rangle) \\
&= \text{EVAL}(\text{actuals}(\text{call}_{00}(z)), \langle [], [0, 0, 0] \rangle) \\
&= \text{EVAL}(\text{call}_{00}(z), \langle [], [0, 0] \rangle) \\
&= \text{EVAL}(z, \langle [0], [0, 0] \rangle) \\
&= \text{EVAL}(\text{actuals}_{1}(x), \langle [0], [0, 0] \rangle) \\
&= \text{EVAL}(x, \langle [0], [0] \rangle) \\
&= \text{EVAL}(\text{actuals}_{1}(2), \langle [0], [0] \rangle) \\
&= \text{EVAL}(2, \langle [0], [] \rangle) \\
&= 2
\end{align*}
\]

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

Figure 4: Execution for Higher-order Programs

belong. Intuitively, for every invocation of a specific function at runtime, there exists a unique tag that characterizes it. The above remarks lead to the following implementation strategy:

1. Initially, the number of formal parameters for each function in the source functional program is recorded. This information will be used to determine the size of the activation record that has to be allocated when a call to this function occurs. Moreover, for every formal parameter, an offset from the beginning of the AR is assigned.

2. Tags are represented as tuples (head, pointer to tail) and they are stored in the beginning of ARs (this is further discussed below). Therefore, a tag can be uniquely characterized by the address in the stack of this tuple. The current value of the tag is stored in a global variable \( t \). Initially, \( t \) is set equal to zero. This value corresponds to the empty tag \( [] \). As stack addresses start at 1, we can equivalently say that initially \( t \) is not pointing to any stack address.

3. Let \( f \) be a function of the source functional program and let \( x_1, \ldots, x_n \) be its formal parameters. Whenever during execution an expression of the form \text{call}_i(f) is encountered and the current tag is \( t \), an activation of the form shown in Figure 5 is allocated. Associated with every entry in the AR, there exists a presence bit (not shown in Figure 5) which indicates whether the entry holds a value or not. The new value of \( t \) is set to point to the beginning of the newly created activation record. The old value of \( t \) is saved as the second component of
the tuple, in the beginning of the new activation record. The first component is the head of the tag and is equal to $i$. Activation records usually contain additional fields, such as saved machine status, temporaries, and so on. As these fields are not important in the following discussion, they are omitted from the figures.

4. When an expression of the form $\text{actuals}(A_0, \ldots, A_m)$ is considered, the position that $t$ is pointing to is located. This position contains the head and a pointer to the tail of $t$. The head is used to select the appropriate expression among $A_0, \ldots, A_m$ and the tail is used as the new value of $t$.

5. Whenever an identifier $x$ is demanded, then the activation record pointed to by the current value of $t$ is first located. The field that corresponds to $x$ can be located by using the offset of $x$ that was recorded during compilation. If the presence bit is set to one, then the corresponding value is retrieved. Otherwise, the definition for $x$ is used in order to calculate the value, which is then placed in its position in the AR.

6. When control returns after a $\text{call}_i(f)$, the activation record is deallocated as usual. We assume that the value that has been computed is returned by being saved in a register as usual [2].

6 Implementation for Higher-order Programs

When a higher-order program is transformed into intensional form, the tags used during execution have to be multidimensional. This imposes the following generalizations to the implementation:

1. During the last step of the transformation algorithm (i.e., before the translation from first-order to zero-order), the parameters of each function and their offsets are determined as in the first-order case.

2. Global variables $t_0, \ldots, t_{N-1}$ are assigned, one for each tag component. Initially, all of them are set to zero.

3. Consider the first-order program that results just before the last step of the transformation is performed. Let $f$ be a function in this program and let $x_0, \ldots, x_n$ be its formal parameters (which are all zero-order). Whenever during execution an expression of the form $\text{call}^{t_n}_i(\cdots(\text{call}^{t_k}_i(f))\cdots)$ is encountered, a new AR is allocated on the stack (Figure 6).

4. When an expression of the form $\text{actuals}^s(A_0, \ldots, A_m)$ is considered, the position that $t_s$ is pointing to is located. This position holds a tuple that contains the head and a pointer to the tail of the $s$-th tag component. The head is used to select the appropriate expression from the list $A_0, \ldots, A_m$. The tail becomes the new $t_s$.  

Figure 5: First-order Activation Record

![Figure 5: First-order Activation Record](image-url)
5. Whenever a formal parameter $x$ of a function is demanded under a tag $t$, then the tag component that corresponds to the last step of the transformation (i.e., $t_{N-1}$) is used to find the corresponding AR. Things are working exactly as in the first-order case.

6. As control is returning, activation records are deallocated.

7 Related Work and Performance Results

The traditional implementation technique for lazy functional language is based on graph reduction [10]. 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, which is a fixed-program machine. A different hashing-based approach for implementing higher-order tagged dataflow has been given by the authors in [12]. Our current work is more directly related to the recent work on dataflow languages and architectures [3, 6]. In fact, [6] presents an activation-record-based technique for implementing functional languages on Explicit Token-Store Architecture. However, higher-order functions are not considered and laziness is not discussed.

In the following, we present performance results obtained from an implementation of the proposed technique. Following, a recent trend in functional language implementation [13], our compiler uses as its target machine code the C language. The other systems used for the comparison are all reduction-based: the LML compiler [5], the Gofer compiler [9] and the Miranda™ interpreter [15]. The programs used for the comparison are mostly standard benchmarks for functional languages. Their main characteristic is the high number of function calls, as this is the main comparison criterion. The first three programs are first-order and the last two are second and third-order respectively. The benchmarks were run on a Sun SPARCserver 690MP, with 64MB RAM, running SunOS 4.1.2. The following table shows the corresponding execution times.

<table>
<thead>
<tr>
<th>Time (in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>Fib</td>
</tr>
<tr>
<td>Tak</td>
</tr>
<tr>
<td>Ack</td>
</tr>
<tr>
<td>Mersenne</td>
</tr>
<tr>
<td>Integration</td>
</tr>
</tbody>
</table>

The results indicate that code generated by our compiler runs at about half the speed of the code produced by the LML compiler. The main reason for this is the fact that LML produces
native code and uses many “source-to-source” transformations in order to produce a program for which generation of efficient code is less complicated. The comparison with Gofer and Miranda shows that the code produced by these systems is generally slower. The Gofer system, which is the best candidate for comparison—as it also compiles to C code—is usually around four times slower.

A comparison in terms of space is not, however, straightforward. This is because our technique is stack-based, while reduction is heap-based. Our experiments show that the number of bytes allocated on the stack using our approach is usually half the number of bytes allocated on the heap during graph reduction. However, further experiments are needed in this direction.

In general, as our approach is stack-based, it does not incur the garbage-collection overhead associated with graph reduction: activation records are allocated and deallocated as in conventional compilers for programming languages [2]. However, garbage collection must be supported when objects of non-flat domains (such as lists) are added to the language.

8 Conclusions

A dataflow implementation technique for lazy finitely typable functional languages has been presented. In contrast to graph reduction approaches, it is stack-based, and is closer to traditional implementations of programming languages.

Future work includes investigating optimizations such as strictness analysis as well as intensional code transformations. Also, we are currently considering the extension of the technique to programs that contain functions which are not finitely typable.

Acknowledgements

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

References


