HIGHER-ORDER DATAFLOW AND ITS IMPLEMENTATION
ON STOCK HARDWARE

P. Rondogiannis and W. W. Wadge
University of Victoria, Canada

KEYWORDS: Higher-Order Dataflow, Functional Programming, Intensional Programming

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 in practice, 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.

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.

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 [7]. 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:

$$f(x) = g(x+1);$$
$$g(y) = y*y;$$
end

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

```c
call_0(f)
where
  f = call_0(g);
  g = y*y;
  x = actuals(5);
  y = actuals(x+1);
end
```

Figure 1 Algorithm for First-Order Programs

1. Let $f$ be a function appearing in the program. 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 $call_i(f)$.
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.

An execution model is established by considering the $call_i$ and $actuals$ as operations on finite lists of natural numbers (referred from now on as tags or contexts). Execution of the program starts by demanding the value of the subject of the where-clause of the intensional program, under the empty tag $[]$. The operator $call_i$ 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 semantic equations as introduced in [18], are:

$$call_i(A),$$
$$actuals(A_0, \ldots, A_m),\theta,\psi = A_{\psi},\psi_i,$$
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 [6].

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 selfapply(f) = f(f), that do not have a finite order. The second requirement disallows functions of the form f(x) = sq, where sq is a 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 $N-1$ functional 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 functional 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:

apply(sq, 2)
where
apply(f, x) = f(x);
sq(y) = y*y;
end

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

call0_0 apply(2)
where
apply(x) = f(x);
sq(y) = y*y;
f = actuals0(sq);
end

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 actuals0 and call0_1, where index $i$ ranges as in the first-order case. For the second step, we use actuals1 and call1_1, 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 $z$:

call0_0 apply(2)
where
apply(x) = f(x);
sq(y) = y*y;
f(z) = (actuals0(sq))(z);
end

It is necessary to pass $z$ inside the actuals before performing the next stage of the transformation. However, the 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 actuals. This is done as follows:

call0_0 apply(2)
where
apply(x) = f(x);
sq(y) = y*y;
f(z) = actuals0(sq(call0_0(z)));
end

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:

call1_0 (call0_0 (apply))
where
apply = call1_0(f);
sq = y*y;
f = actuals0(call1_0(sq));
z = actuals1(x);
y = actuals1(call0_0(z));
x = actuals1(2);
end

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
Figure 3 Algorithm for Higher-Order Programs

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 $\text{call}'(f(\text{Act}))$, where 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 $Fp$ 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 $Fp$. Add the variables to both sides of the definition of $Fp$, advancing them before they enter the scope of actuals.

\begin{equation}
\text{notation} \ <t_0, \ldots, t_{N-1}> \ \text{to denote a tag}. \ \text{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 call}'(\cdot)) \ \text{. 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.}
\end{equation}

On the other hand, actuals' takes from the tag the list corresponding to $s$, uses its head $i$ to select the $i$th argument of actuals, and returns the tail of the list to the tag. The new semantic equations are:

\begin{equation}
\begin{split}
\text{(call}'(A) <t_0, \ldots, t_{N-1}>) &= A <t_0, \ldots, \text{cons}(t_i, t_{i+1}), \ldots, t_{N-1}>
\end{split}
\end{equation}

\begin{equation}
\begin{split}
\text{(actuals}'(A_0, \ldots, A_m) <t_0, \ldots, t_{N-1}>) &= A_0 <t_0, t_1, \ldots, t_{N-1}>
\end{split}
\end{equation}

For $m$-ary operation symbols $\theta$, the semantic equation is the same as in the first-order case. The evaluation of a program starts with an $N$-tuple of 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 execution of the apply program is given in Figure 4.

Implementation on Stock Hardware

Modern dataflow architectures [3],[5], 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 further 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 $\text{call}(f)$ 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 role 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 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.

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 $i$, there has existed a previous demand, under the same tag, for the function variable (like $f$) in which the formal belongs. 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 the 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
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 $x_1, \ldots, x_n$ be its formal parameters (which are all zero-order). Whenever during execution an expression of the form $\text{call}_t^{x_1}(\ldots (\text{call}_t^{x_n}(f))\ldots)$ is encountered, a new AR is allocated on the stack (Figure 6).

4. When an expression of the form $\text{actuals}^t(A_0, \ldots, A_m)$ is considered, the position that $t$ is pointing to is located. This position contains the head and a pointer to the tail of 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 $t_s$.

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.

**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:

**Figure 5 First-Order Activation Record**

### Figure 5 First-Order Activation Record

- New value of $t$
- Old value of $t$
- Points back into the stack
- $i$
- $t$
- $x_1$
- $\vdots$
- $x_n$

**Figure 6 Higher-Order Activation Record**

### Figure 6 Higher-Order Activation Record

- New $t_{s_0}$
- $i_0$
- $t_{s_0}$
- $\vdots$
- $i_k$
- $t_k$
- $x_1$
- $\vdots$
- $x_n$

**Related Work and Performance Results**

The traditional implementation technique for lazy functional languages 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], [5]. In fact, [5] presents an activation record based technique for implementing functional languages on the 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 [4], 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>Program</th>
<th>Intensional</th>
<th>LML</th>
<th>Gofer</th>
<th>Miranda</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fib</td>
<td>3.0</td>
<td>2.0</td>
<td>14.0</td>
<td>45</td>
</tr>
<tr>
<td>Tak</td>
<td>2.7</td>
<td>2.4</td>
<td>16.4</td>
<td>54</td>
</tr>
<tr>
<td>Ack</td>
<td>2.3</td>
<td>1.0</td>
<td>60.0</td>
<td>16</td>
</tr>
<tr>
<td>Mersenne</td>
<td>4.8</td>
<td>1.4</td>
<td>15.8</td>
<td>43</td>
</tr>
<tr>
<td>Integration</td>
<td>3.6</td>
<td>23</td>
<td>-</td>
<td>19</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 in the stack using our approach is usually half the number of the bytes allocated in 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.

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.

Acknowledgments: 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.

E-mail: {prono,wwadge}@car.uvic.ca

References