Compiling Functional Programs to Dataflow Code*

Hans Wolfgang Loidl
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
Internet: hwloidl@risc.uni-linz.ac.at

December, 1991

Presented at: Workshop on Parallel Computers and Programming Languages, Dagstuhl, Germany, February 26–28, 1992

Abstract

This paper describes the compilation process from a program in a non-strict functional programming language to a program in the assembler language of an abstract hybrid dataflow/controlflow machine. In this translation, dataflow graphs are used as an intermediate language. The outcome of such a compilation is a control-driven program that simulates the dataflow behaviour which is described by the corresponding dataflow graph.

1 Introduction

Among all programming languages, functional programming languages have the advantage of containing most inherent parallelism. The main problem with these languages is to exploit this parallelism by creating efficient code for an imperative (control-driven) machine. Our approach for solving this problem is to use dataflow graphs as an intermediate language and then to create code that simulates the dataflow behaviour on an abstract hybrid dataflow/controlflow machine.

The work presented here is part of the ADAM & EVE project. The main goal of this project is to study the behaviour of programs in the functional programming language EVE that have been compiled to the assembler code of the hybrid dataflow/controlflow machine ADAM. For executing these programs a simulator on a network of 16 transputers has been implemented. For a detailed description of the ADAM machine, see [Schreiner, 1990].

This paper describes the results of the second part of the project, which consisted of the implementation of a compiler generating ADAM assembler code from the functional programming language EVE. The basic ideas lying behind this translation process will be discussed here.

* *Sponsored by: Austrian Forschungsförderungsfonds, project no. P-06931-PH*

This project has been influenced by the work at MIT on dataflow architectures and compilers for these architectures. For example, the handling of data structures is done with I-structures that are described in [Arvind *et al.*, 1987]. Similar ideas for the compilation of a functional language to the assembler code of a hybrid dataflow/controlflow machine can be found in [Iannucci, 1988] and also in [Traub, 1986].

2 Source, Intermediate and Target Language

2.1 Source Language

As the source language in our compilation process we have chosen a non-strict functional programming language. The non-strict semantics minimizes the number of synchronization points in function calls and in the usage of compound data-structures. However, to simplify the compilation process a very small functional language, called EVE, has been defined that incorporates only the most important constructs of a functional language and lacks most of the constructs that do not add anything to the expressive power of the language. So, all the features of other functional programming languages that are missing in EVE can be simulated by the existing EVE constructs[1].

```
IntList      == LIST OF INT;
PredOfInt    == BOOL[INT];

Filter[p,list] : IntList[PredOfInt,IntList] ==
  IF NULL[list]
    THEN IntList[]
    ELSE { hd : INT     == list.HEAD;
           tl : IntList == list.TAIL }
         IF p[hd]
           THEN hd :: Filter[p,tl]
           ELSE       Filter[p,tl] ;
```

Figure 1: An EVE Program

In this paper no formal definition of the source language will be given. However, the meaning of the various constructs should be clear from the context. To illustrate the structure of an EVE program, Figure 1 shows an EVE program for implementing the well known filter function. The semantics of this function can be described as follows:

- **Given:** A predicate p, a list l of the structure $[l_1, \ldots, l_n]$.
- **Find:** A list l' of the structure $[l'_1, \ldots, l'_m]$ for which the following holds:

[1] Mainly those constructs that are regarded as "syntactic sugar", such as list comprehensions or pattern matching, have not been included into the language.

$\exists f : \{1, \ldots, m\} \to \{1, \ldots, n\}$: index mapping
  $\forall 1 \leq i \leq m$:
    $l_{f_i} = l'_i$, same elements
    $p(l'_i)$, p holds
    $f_i < f_{i+1}$, f monotonic
    $\forall f_i < j < f_{i+1}$: all elements between
      $\neg p(l_j)$. do not fulfill p

Looking at the syntactic structure of this program the following has to be mentioned:

- In EVE data types can be defined by using the atomic data types `INT`, `BOOL` and `CHAR` and the data type constructors `LIST OF`, `TUPLE OF`, `UNION OF` and `...[...]`[2].
- Every function in EVE is split into a head and a body where
  - The head is again split into two parts:
    - A construct stating the names of the function and of the parameters and
    - a construct stating the corresponding types.
  - The body contains an expression defining the function. In this example it can be seen how functions operating on compound data structures become larger than in other functional languages as there is no pattern matching available in EVE.

2.2 Intermediate Language

The intermediate language that represents the abstract computation model is formed by dataflow graphs. The main advantage of dataflow graphs is the fact that they show the maximal parallelism that lies in a program which is only restricted by data-dependencies between the objects of the program. However, usually such dataflow graphs contain too much parallelism for an efficient implementation. Therefore, some compile time analysis is necessary to reduce this parallelism to a degree where creating and synchronizing parallel processes do not need the main part of the computation time.

The interface of a dataflow graph as it is used here is shown in Figure 2. It is important to notice that there are two special arcs needed in the graph:

- The *trigger* arc is an additional input arc that is needed for starting a non-strict function even before the first of its argument has arrived.
- The *signal* arc is an additional output arc that is needed to guarantee that no token is left in the dataflow graph when tokens have appeared on every output arc (*self-cleaning property*). Without this arc, the self-cleaning property would be lost for dataflow graphs that contain side-effect nodes.

[2] This is also called the function type constructor.

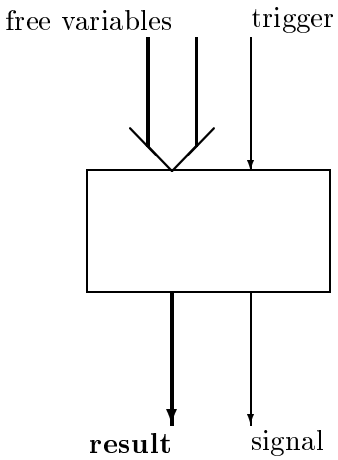

Figure 2: Interface of a Dataflow Graph

2.3 Target Language

The outcome of the whole compilation process should be a program in the assembler code of the ADAM machine. This assembler language is similar to the language of any control-driven model but it is augmented with constructs for the creation and synchronization of parallel tasks. With these constructs it is possible to simulate the dataflow model on a control-driven architecture.

The most important of these constructs are:

- SPLIT, that splits the controlflow into two parallel threads. From the dataflow point of view, the instruction represents a node that takes one input token and splits it into two independent output tokens. This construct is used to create parallelism.
- WAIT, that joins two threads in the controlflow into one thread. From the dataflow point of view, the instruction represents a node that takes two input tokens and merges them into one output token. This construct is used to synchronize parallel parts of the program.

3 Compilation

In this section, the compilation process from a functional program to the assembler code of the ADAM machine will be discussed with respect to the parallelism that is extracted out of the program. This process is split into two parts with dataflow graphs as the intermediate language. In this paper we will concentrate on only one construct namely the function call. However, the most important aspects of the translation of a

functional language construct to a dataflow graph and then to ADAM assembler code can already be shown with this construct.

3.1 From a Functional Program to a Dataflow Graph

One of the most interesting constructs of a non-strict functional language with respect to its implicit parallelism is the function call. The dataflow graph for this construct is shown in Figure 3. Obviously, all the arguments of the function call can be evaluated in parallel. However, due to the non-strict semantics of the language, the function that is called can be evaluated in parallel with its arguments. When in the evaluation of the function the value of an argument is needed that has not been computed yet, the evaluation of the function will be suspended.

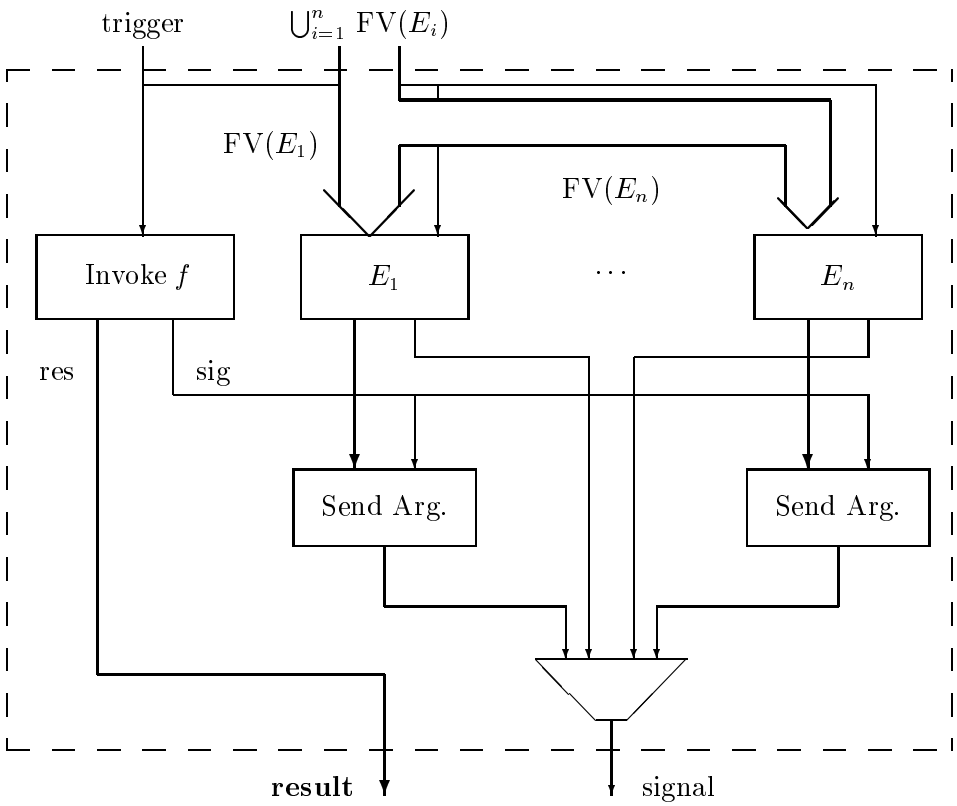

Figure 3: Dataflow Graph for the Function Call (f E_1 ... E_n)

Another consequence of the non-strict semantics is the necessity of splitting the invocation of a function and the sending of the arguments to this function into two parts.

As an argument can only be sent to a function after having invoked this function, a synchronization between function invocation and the sending of arguments has to take place.

The nodes for the invocation of a function and for the sending of an argument are special in the sense that they perform some side-effects. For a further description of these side-effects see Section 3.2.

3.2 From a Dataflow Graph to an Assembler Program

The ADAM assembler code is essentially a conventional two address code in which almost all instructions contain a reference to their successors at which the computation will be continued. Usually the successor is addressed via a relative distance (written as a signed number). However, one can also use symbolic labels instead of relative distances. As described in Section 2.3, the instructions SPLIT and WAIT are used for the creation of parallelism and for synchronization, respectively.

trigger :	SPLIT	*invoke*, +1	; Feed trigger to
	SPLIT	$trigger_{E_1}$, +1	; the function and
	⋮		; to each argument
	SPLIT	$trigger_{E_{n-1}}$, $trigger_{E_n}$	
invoke :	FASTCALL	f, +1	; Invoke Function
	COPYI	$r_{act}, r_{act}, send$	; Fct activation arrives
	COPYI	$r, r, result$	; Result of fct arrives
send :	SPLIT	$send_{E_1}$, +1	; One send for
	⋮		; each argument
	SPLIT	$send_{E_{n-1}}$, $send_{E_n}$	
	$\mathbf{C}_{E_i}$		; $i = 1, \ldots, n$
$result_{E_i}$:	WAIT	e_i, +2	; Synchronize result of
$send_{E_i}$:	WAIT	e_i, +1	; arg. with fct. invocation
	SEND	r_{E_i}, r_{act}, i	; send ith arg. to fct.
	JUMP	arg_{E_i}	
	$\mathbf{C}_{SignalTree}$		

Figure 4: Assembler Code for a Function Call

Figure 4 shows the assembler code that is produced from a function call whose dataflow graph has been shown in Figure 3. For clarification of the main aspects of the code, the

distribution of the free variables among the expressions E_i $(i = 1, \ldots, n)$ is not shown here. The input and the output of the whole graph are the following labels:

- **Input:** *trigger*.
- **Output:** *result*, *signal*.

$\mathbf{C}_{E_i}$, $\mathbf{C}_{ST}$ denote the graphs of the expressions E_i and of the final signal tree, respectively. The input and the output of these graphs are the following labels:

$\mathbf{C}_{E_i}$, $i = 1, \ldots, n$:

- **Input:** $trigger_{E_i}$.
- **Output:** $result_{E_i}$, $signal_{E_i}$.

$\mathbf{C}_{SignalTree}$:

- **Input:** $signal_{E_i}$, arg_{E_i}, $i = 1, \ldots, n$.
- **Output:** *signal*.

In this code-sequence, the first block of SPLIT instructions creates as many threads as are necessary to execute the function and the arguments in parallel. The next block invokes the function. The FASTCALL command performs a side-effect as it creates a dynamic link to the function f. The signal of the function invocation (in form of a token carrying the address of the function activation) arrives at the next address after the FASTCALL command and the result of the function call arrives at the next address plus one. After the arrival of a token carrying the address of the function activation, it is possible to send arguments to the function (this is done by the code block after the corresponding argument). Therefore, this send token will be split into as many tokens as there are arguments for the function. For each argument the corresponding send token has to be synchronized with the result of the argument. After this synchronization the value of the argument is sent to the function. The last construct in this code-sequence is a signal tree that takes all of the now superflous tokens and merges them together to just one output token which is then the signal of the whole function call.

Although the translation process has only been discussed for one construct, it should be sufficient to give some feeling about how other constructs could be translated, too. For example, for translating a tuple construction one would get essentially the same structure as for a function call but with a node initiating tuple construction instead of the invoke node and nodes for writing into the fields of the tuple instead of nodes sending arguments.

4 Conclusion

In this paper we wanted to show how dataflow graphs, that are quite natural representations of functional programs, can be translated into the assembler code of a conventional control-driven machine that has a few additional commands for handling parallelism. By this translation process one can get maximal parallelism, as sequentialization has to be done neither in the source nor in the intermediate language. On the other hand, the presented target language can be implemented on conventional hardware by adding those few constructs that are necessary for parallelism.

One point that has not been discussed in this paper is the optimization of the resulting code. In the presented translation process sometimes rather inefficient code is produced. Especially, some tests with an existing prototype compiler have shown, that the synchronizations with the trigger token yields quite much overhead. So, one possibility for improving the efficiency of the resulting code would be to ignore the trigger token in purely arithmetical expressions since they are always strict and to forward the trigger directly to the signal token in such a case. Furthermore, a strictness analysis on user defined functions could be performed to determine the arguments in which a function is strict. When this function is called the strict arguments can then be evaluated before the function itself, thereby avoiding a synchronization with this trigger token. However, these considerations on optimizations of the resulting code are part of future work in this area.

References

[Arvind *et al.*, 1987]
Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali.
I-Structures: Data Structures for Parallel Computing.
Computation Structures Group Memo 269, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 1987.
Also in: Proceedings of the Workshop on Graph Reduction, Los Alamos, New Mexico, September 28 – October 1, 1986.

[Iannucci, 1988]
Robert Alan Iannucci.
A Dataflow / von Neumann Hybrid Architecture.
Technical Report MIT/LCS/TR-418, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1988.

[Schreiner, 1990]
Wolfgang Schreiner.
ADAM & EVE — An Abstract Dataflow Machine and Its Programming Language.
Master's thesis, Johannes Kepler University, Linz, Austria, October 1990.
Also: Technical Report 90-42.0, RISC-Linz, Johannes Kepler University, Linz, Austria, 1990.

[Traub, 1986]
Kenneth R. Traub.
A Compiler for the MIT Tagged-Token Dataflow Architecture.
Technical Report MIT/LCS/TR-370, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1986.