On Compiling Functional Programming Languages for Maxeler's Dataflow Engines

Sven Cerk

University of Ljubljana, Faculty of Computer and Information Science
Večna pot 113, SI-1000 Ljubljana, Slovenija
E-mail: sven.cerk@gmail.com

Abstract

It has long been conjectured that functional programming languages might be particularly suited for programming dataflow computers. A method for compiling a Haskell-like functional programming language for Maxeler's dataflow engines which represent a viable alternative to parallel coprocessors, e.g., Xeon Phi, or GPUs, is outlined.

1 Dataflow engines

A Maxeler's dataflow processing system consists of a CPU application running on a computer and a dataflow engine (DFE) performing pipelined computations. In a DFE the data are streamed from memory through a processing chip where they flow from one arithmetic unit to another and finally back into the memory [1].

The arrangement of arithmetic units is described as a graph consisting of various types of nodes called a kernel graph. The available types of nodes are:

- **computation nodes** describe arithmetic or logical operations, e.g., addition, subtraction, multiplication, ..., or type casting operations;
- **value nodes** contain a single value which is either constant or set by the CPU at runtime;
- **offset nodes** describe access to past or future elements of the data stream;
- **multiplexer nodes** describe a choice between multiple input streams;
- **counter nodes** describe a stream of data starting with an initial value and incrementing at every clock tick;
- **input/output nodes** describe the stream of data flowing into or out of the DFE.

At the time being, Maxeler supports a programming model where the program running on the CPU is written in C (or Python or Matlab) while kernels (running on DFEs) are written in a-kind-of Java [1], i.e., in an imperative way. But decades ago a stream based functional programming language Sisal was compiled for various parallel architectures including Manchester Prototype Dataflow Computer and CSIRO/RMIT Dataflow Computer [2]. It is worth investigating how well kernels written in a functional programming language can be compiled for DFEs.

2 Kernel Graphs as Functions

We hypothesise that any configuration in a kernel graph can be successfully modelled as a composition of pure functions operating on either atomic values or on streams of atomic values.

For this purpose we define a programming language named MaxHs that borrows its syntax and semantics from Haskell [3]. This language is actually a very small subset of Haskell with some predefined data types, operators and functions not present in the actual Haskell programming language.

2.1 Streams

A stream of data is a list of atomic values of the same type flowing through the DFE. There can be multiple streams flowing through a complex computational kernel graph on the DFE simultaneously.

In MaxHs we represent streams as values of the data type `Stream a` where `a` is an atomic type (a type on which a Maxeler DFE can operate). `Stream a` can be thought of as a list of values of type `a`. Informally we could interpret `Stream a` as

```
data Stream a = EmptyStream
              | ConsStream a (Stream a)
```

in Haskell. However, when compiled `Stream a` will correspond to a stream of some kernel graph node.

2.2 Composition

In a functional programming language one of the main constructs is function composition which, in terms of a kernel graph, translates into a connection between two parts of a graph: an input to the first function is exactly the output of the second.

2.3 Computation Nodes

Computation nodes perform all computations on streams of data. They have one or two input streams and one output stream. At every clock tick they perform a basic operation on the elements waiting on their inputs and place the result on their output stream.

The inputs to a computation node can either come from two streams, a stream and a value node or two value nodes. Each of these situations can be described as a pure function operating on streams and values.

2.3.1 Processing Streams

The basic stream processing functions are

```
map :: (a -> b) -> Stream a -> Stream b
zipWith :: (a -> b -> c)
  -> Stream a -> Stream b -> Stream c
```

In both cases, the semantics is the same as in Haskell. Types a, b and c must be atomic DFE types while the function supplied in the first argument can be a single DFE operator or a complex arrangement of them.

2.3.2 Two Values

A computation on two values corresponds to a function of two atomic arguments producing an atomic result. The type of such a function could be a -> b -> c, where a, b and c must be atomic types.

2.4 Value Nodes

Value nodes correspond to any atomic values used in our program.

2.5 Offset Nodes

Offset operations are represented as a function on streams offsetting the stream by the supplied number of places:

```
offset :: Int -> Stream a -> Stream a
offset n s = if n > 0
    then    (drop n s)
         ++ (take n $ repeat 0)
    else    (take (-n) $ repeat 0)
         ++ (take (-n) s)
```

With these constructs we can already implement some basic computations such as the moving average of a stream of values. The following function in MaxHs computes a stream of consecutive three-point averages of an input stream:

```
movingAverage :: Stream a -> Stream b
movingAverage xs = let
    ys = zipWith (+) xs (offset (-1) xs)
    zs = zipWith (+) (offset 1 xs) ys
  in
    map (/ 3) zs
```

The code above is translated into a dataflow graph shown in Figure 1 (left).

2.6 Multiplexer Nodes

Multiplexer nodes choose between their input streams according to a control stream:

```
mux :: Stream Int -> [Stream a]
  -> Stream a
mux EmptyStream _ = EmptyStream
mux (ConsStream c cs) streams =
  ConsStream (head $ streams !! c)
             (mux cs $ map tail streams)
```

2.7 Counter Nodes

Counter nodes produce streams of incrementing integers and thus correspond to infinite Streams in MaxHs:

```
counter start increment = ConsStream start
  (counter (start + increment) increment)
```

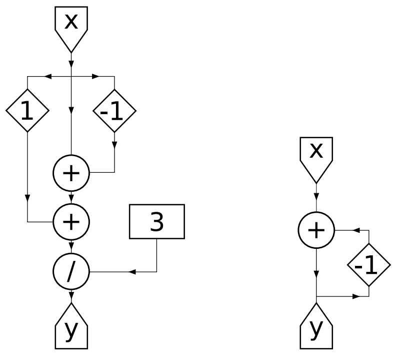

Figure 1: The translation of the “moving-average” (left) and the “rolling sum” (right) examples.

2.8 Loops in Kernel Graphs

Some applications require loops in a kernel graph, i.e., a stream of data flowing to a previous stage of its own computation. The simplest example of such a computation is the calculation of a rolling sum of a stream (the computation of consecutive partial sums of a stream).

In a non-strict programming language, e.g., Haskell, one can describe such a dependency as a recursive value definition – a value that contains a reference to itself in its definition. For example, a function computing a rolling sum of a stream could be implemented in MaxHs as

```
rollingSum :: Stream a -> Stream a
rollingSum xs = ys where
    ys = zipWith (+) xs (offset (-1) ys)
```

The code above is translated into a dataflow graph shown in Figure 1 (right).

3 Conclusion

For every type of nodes found in a kernel graph we have described a way to model it as a function in a purely-functional programming language similar to Haskell. We believe that with these basic functions on streams and their compositions one can describe any kernel graph configuration desired.

At the time being, the compiler based on the described ideas is being implemented as a part of the work on the authors B.Sc. thesis. The beta version is to be available in mid-September 2015.

Acknowledgement This work was done under the supervision of Boštjan Slivnik, who is with the Univeristy of Ljubljana, Faculty of Computer and Information Science. The author is grateful to Miha Eleršič for many valuable discussions.

References

[1] *Multiscale Dataflow Programming*, London, UK, 2014.

[2] S. K. Skedzielewski, *Sisal*, in Parallel Functional Languages and Compilers, B. K. Szymanski (ed.), ACM Press, New York (NY), USA, 1991, pp. 105–158.

[3] S. Marlow et. al., *Haskell 2010 Language Report*, S. Marlow (ed.), Cambridge, UK, 2010. Retrieved July 13th 2015.