Compiling APL to Accelerate Through a Typed IL

Michael Budde

University of Copenhagen
skx295@alumni.ku.dk

6. November 2014

Abstract

APL is a functional array programming language from the 1960's. While APL no longer has widespread adoption as a general programming language, there still exist large codebases in APL in the industry. Accelerate is an array language embedded in Haskell for doing high-performance computation on GPU's.

In this report I present a compiler written in Haskell for compiling a typed intermediate array language (TAIL), as produced by the APL compiler by Elsman [2], to Haskell programs using the Accelerate library that can be run on GPU's by using the backends available for Accelerate.

1 Introduction

APL was created by Kenneth E. Iverson in the 60's and is a functional array language that supports multi-dimensional arrays and first and second order functions. While APL is an old language, some industries such as the financial industry still has large code bases in APL that are being actively developed [3]. APL features a very succinct syntax where symbols are used for most of the built-in operations such as 'ρ' for reshape and ',' (comma) for concatenation of arrays. The same symbol is also used for different operations and the choice of operation depends on the number of arguments to the operator. The syntax of APL means that programs can be written concisely but it also means that programs can be difficult to understand and to maintain. The syntax also creates some difficulties when it comes to compiling the language since it can be difficult to parse. This project builds on the work on the APLTAIL compiler by Elsman [2], which can compile a subset of APL to a typed intermediate array language. This intermediate language is easy to parse, reason about and perform transformations on. For a formal description of TAIL please refer to the paper by Elsman and Dybdal [3].

Accelerate is an array language embedded in Haskell that supports multi-dimensional arrays. Accelerate can generate code for different architectures by using different backends. One of the available backends is a CUDA backend that allows the code to be offloaded to a GPU and run in parallel.

The goal of this project was to do the ground work of making a compiler from a subset of APL to Accelerate and evaluate the performance of the compiled programs running on a GPU. This report will present a compiler from the intermediate language TAIL to Accelerate that enables compilation of a subset of APL to programs that can use CUDA supporting GPUs to perform calculations in parallel. In the rest of the report I will refer to this compiler as the APLACC compiler. The code is available online [1].

Section 2 will give a description of the TAIL syntax. In section 4, we will see how the transformation from TAIL to Accelerate is done. Finally in section 8 we come to a conclusion and discus possible future work.

2 TAIL

The APLACC compiler takes TAIL programs as input. What follows is a description of the TAIL syntax. We use i to range over integers and d to range over doubles. We use x to range over identifiers. Identifiers start with a letter and is followed by zero or more alphanumeric characters or underscores. For some z,

$$
\begin{array}{rcll}
v & ::= & i \mid d \mid \texttt{inf} \mid x \mid \texttt{\sim}v & \text{(values)} \\
e & ::= & v \mid \texttt{[}\vec{v}\texttt{]} \mid x\,\iota\texttt{(}\vec{e}\texttt{)} & \text{(expressions)} \\
 & \mid & \texttt{fn}\ x\texttt{:}\tau\ \texttt{=>}\ e & \\
 & \mid & \texttt{let}\ x\texttt{:}\tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 & \\
 & & & \\
\kappa & ::= & \texttt{int} \mid \texttt{double} & \text{(base types)} \\
\tau & ::= & \texttt{[}\kappa\texttt{]}i \mid \texttt{Sh(}i\texttt{)} \mid \texttt{Si(}i\texttt{)} \mid \texttt{Vi(}i\texttt{)} & \text{(types)} \\
\iota & ::= & \texttt{\{[}\vec{\kappa}\texttt{], [}\vec{i}\texttt{]\}} \mid \epsilon & \text{(instance lists)}
\end{array}
$$

Figure 1: Grammar describing the TAIL syntax supported by the compiler. The symbols i, d and x denote an integer, a double and an identifier, respectively.

we write $\vec{z}^{(n)}$ to denote the sequence $z_0, z_1, \ldots, z_{n-1}$. If the exact length of the sequence is unknown or irrelevant, we leave it out and just write $\vec{z}$.

Figure 1 shows the TAIL syntax supported by the compiler. This syntax deviates a bit from the grammar described in the TAIL paper [3] but is based on the actual output of the APLTAIL compiler.

A TAIL program consists of a single toplevel expression (e). An expression can either be a value (v), a vector of values, a function call, a lambda expression or a let expression. Values can be in integers (i), doubles (i), infinity, an identifier (x) or the negation of a value.

The possible types in TAIL are multi-dimensional arrays (`[`κ`]`i, written as $[\kappa]^i$ when not describing the syntax), shape vectors (`Sh(`i`)`), singleton integers (`Si(`i`)`) and single-element integer vectors (`Vi(`i`)`). The type of elements in arrays can be of one of the base types, which are integers or doubles.

Function calls in TAIL can be annotated with instance lists that specify which particular instance of a polymorphic function is being used. The first list contains the type instantiations and the second list contains the rank instantiations. The number of elements in the two lists depend on the function. For example, the type of the `reduce` function is given as:

$$\forall\alpha\gamma.\ (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha]^{\gamma+1} \to [\alpha]^{\gamma}$$

The type is parameterized over two type parameters α and γ, where α denotes a base type and γ a rank type. In addition, α is used to denote the scalar array type $[\alpha]^0$. An instantitation list for a call to `reduce` would then be the particular values of α and γ for that call. For example, for $\alpha =$ `int` and $\gamma = 0$ a call to `reduce` could look like:

```
reduce{[int], [0]}(addi, 0, [1,2,3])
```

Instantiation lists were added (by Martin Elsman) to the output of the APLTAIL compiler as we found that this information was needed in the transformation APL programs to Accelerate. Since APLTAIL already performs type inference and has this information available, adding the information to the TAIL programs was a better solution than having to do type inference in APLACC.

It is currently assumed that the final result of any TAIL program has the type `[double]0`. To allow other result types the compiler will need a way to get this information. Since APLTAIL already has information about the result type, it would be easy to add this information to the TAIL syntax. Alternativly, the information can be deduced while compiling the TAIL program, although it would require some modifications to the compilation algorithm described in this report.

3 Accelerate

Accelerate is an array language embedded in Haskell. Array computations are instances of the data type `Acc`, which represents an abstract syntax tree of the computation. When running the array computation, the tree is optimized in various ways and then passed on to a backend which takes care of executing the computation and returning the result. There exist a number of different backends to Accelerate but the primary one is the CUDA backend. This backend will compile the Accelerate AST to CUDA code and run it in parallel on the GPU. There is also a backend for running the computations sequentially on the CPU, mainly for testing purposes.

In addition to the `Acc` data type Accelerate has another data type `Exp` that represents a scalar expression. Arrays are multi-dimensional and are represented by the type

$$\texttt{(Shape}\ sh\texttt{, Elt}\ e\texttt{)} \Rightarrow \texttt{Array}\ sh\ e$$

Shapes have the number of dimensions encoded in the type and are constructed using the *snoc* operator (which is both a type constructor and a type level operator):

```
Z :. 5 :. 8  :  Z :. Int :. Int
```

The following is an example of an Accelerate program. The qualifier `P` is the Haskell prelude module and `Prim` is the module containing implementations of primitive functions. Unqualified functions and types come from the Accelerate module.

```
program :: Acc (Scalar P.Double)
program =
  let a = 3.14 :: P.Double in
    Prim.reduce (+) (constant (0.0 :: P.Double))
      (Prim.each (\x -> x + constant (a :: P.Double))
         (use (fromList (Z :. 3) [1.0, 2.0, 3.0]) ::
            Acc (Array DIM1 P.Double)))
main = P.print (Backend.run program)
```

4 From TAIL to Accelerate

The translation of a TAIL program to an Accelerate program in the APLACC compiler is split into three stages: parsing the TAIL source code, converting the TAIL AST to an AST representing Accelerate code and finally converting the Accelerate AST into Haskell code. The last two steps will be presented as a single step in the following sections.

The TAIL program is translated to Haskell source code instead of compiling directly to an Accelerate computation. The latter would require a function definition of the form

```
compileTail : String → Acc (...)
```

where "..." could be a number of different types depending on the resulting type of the TAIL program. A benefit of choosing the source code approach is that the output is human readable and can be useful as a starting point for manually translating the APL code to Haskell.

The parsing of TAIL is very straightforward and was easy to implement using the parser combinator library Parsec in Haskell. The parsed TAIL program is represented by the abstract syntax tree shown in Figure 2.

```
rank    = R integer
btype   = IntT | DoubleT
type    = ArrT btype rank | ShT rank
        | SiT rank | ViT rank
instdecl = ([btype], [integer])
exp     = Var string | I integer | D double
        | Inf | Vc [exp] | Neg exp
        | Let string type exp exp
        | Op string (Maybe instdecl) [exp]
        | Fn string type exp
```

Figure 2: AST for representing TAIL code.

4.1 Translating Types

Lets first consider the translation of TAIL types to Accelerate types. As we have already seen, in TAIL we have four type classes: arrays, shapes, singleton integers and single-element integer vectors. Scalar values are represented by a zero dimensional array or singleton integer. There are also a number of subtyping relations: shapes and single-element vectors are subtypes of one dimensional arrays and singleton integers are a subtype of zero dimensional arrays.

In Acclerate the situation is a bit more complicated. There are two main data types: `Acc` for array-valued computations and `Exp` for scalar expressions. In addition to that there are also plain values[1], e.g. `Int`s, and shapes. Finally we have shapes. It is possible to convert values between certain types but there are not the same subtyping relations as in TAIL. This means for example that we cannot treat a shape vector like it was an array. Instead we must explicitly convert the shape to an array or pass it as an argument to a function that works on shapes. Accelerate gives us the following three functions to convert between different types:

```
constant : Elt e ⇒ e → Exp e
    unit : Elt e ⇒ Exp e → Acc (Scalar e)
     the : Elt e ⇒ Acc (Scalar e) → Exp e
```

Converting `Int` values to `Double` can be done with the Haskell built-in function `fromIntegral` and con-

[1] Not to be confused with the `Plain` associated-type for the `Lift` type class in Accelerate.

verting from `Exp Int` to `Exp Double` can be done with the `fromIntegral` function from Accelerate.

Consider the following TAIL program:

```
let a:[int]0 = 5 in
reduce{[int], [0]}(addi, a, [a])
```

When translating this code we have a choice in which type to give `a`. We can either give it type `Int`, `Exp Int` or `Acc (Scalar Int)`. The choice we make decides which conversions we will need to do in the call to `reduce` since the second argument to `reduce` should be of type `Exp Int` while the in the vector literal it should be a plain `Int`. One thing to note is that we do not have a way to convert from `Exp a` to `a`. This means that choice that will work in this case is storing `a` as `Int` because otherwise we can't use it in the vector literal.

After some experimentation I have ended up with the following mapping:

```
ArrT e (R 0)  →  Exp e
ArrT e (R r)  →  Acc e r
ShT _         →  Acc 1 IntT
SiT _         →  Exp IntT
ViT _         →  Acc 1 IntT
```

A couple of notes: The mapping says to convert scalar arrays and singleton integers to `Exp` but this is not the whole story. As I will explain later these types will be converted to plain types if possible. Secondly, shapes in TAIL are converted to vectors. This is an area of compiler that still needs more work. In section 6 I will discuss this problem in more detail.

This mapping is used when translating the types in let- and lambda-expressions. Since values can be used in different contexts that require different types in Accelerate we also need rules for converting between types. For this reason the APLACC compiler has a set of *type casting* rules. Type casting of a Haskell expression ϵ from type τ_1 to τ_2 takes the following form:

$$(\epsilon : \tau_1) \rightsquigarrow \tau_2 = \epsilon'$$

where ϵ' is some expression of type τ_2. The explicit type of ϵ is sometimes left out if the type is obvious from the context. Figure 3 shows the type casting rules in APLACC. When casting from `Plain` to `Exp` we also add a type signature since Haskell is not always able to deduce the type of ϵ.

$$
\begin{array}{lll}
(\epsilon : \texttt{Plain}\ \kappa) \rightsquigarrow \texttt{Exp}\ \kappa & = & \textit{constant}\ (\epsilon\ \textit{::}\ \kappa) \\
(\epsilon : \texttt{Plain}\ \kappa) \rightsquigarrow \texttt{Acc}\ r\ \kappa & = & \textit{unit}\ (\textit{constant}\ (\epsilon\ \textit{::}\ \kappa)) \\
(\epsilon : \texttt{Exp}\ \kappa) \rightsquigarrow \texttt{Acc}\ 0\ \kappa & = & \textit{unit}\ \epsilon \\
(\epsilon : \texttt{Acc}\ 0\ \kappa) \rightsquigarrow \texttt{Exp}\ \kappa & = & \textit{the}\ \epsilon \\
(\epsilon : \tau) \rightsquigarrow \tau & = & \epsilon \\
(\epsilon : \tau_1) \rightsquigarrow \tau_2 & = & \textbf{fail}
\end{array}
$$

Figure 3: Type casting rules for converting expressions from one type to another.

4.2 Translating Expressions

Rules for converting a TAIL expression e (in the form a TAIL AST, see Figure 2) to a Haskell expression ϵ takes the following form:

$$[\![e]\!]\ \mathrm{E}\ \tau = \epsilon$$

where E is an environment that maps identifiers to their types and τ is the *type context*. The type context specifies what type the resulting expression is expected to have. For example should the expression `I 5` in a `Plain IntT` context translate to just the literal `5`, while in a `Exp IntT` context it should translate to `constant (5 :: Int)`. This is where the type casting rules are used.

Figure 4 shows the rules for translating TAIL expressions to Haskell expressions. In the rules, τ^* denotes a TAIL type while τ represents the corresponding Accelerate type as specified by the type mapping shown in subsection 4.1. For any Accelerate type τ, κ_τ denotes the base type of τ, e.g. $\kappa_{\texttt{Exp IntT}} = \texttt{IntT}$.

When translating negations and let expressions we use the function *cancelLift*. The function has the following definition:

$$
\begin{array}{l}
\textit{cancelLift}\ (\texttt{Exp}\ \kappa)\ (\textit{constant}\ (\epsilon\ \textit{::}\ \kappa)) \\
\qquad\qquad = (\epsilon, \texttt{Plain}\ \kappa) \\
\textit{cancelLift}\ \tau\ \epsilon = (\epsilon, \tau)
\end{array}
$$

To see why the function is needed consider the following example:

```
let a:[double]0 = 5.4 in
reduce{[double],[0]}(addd, 0.0, [a, a])
```

Our type mapping says that `[double]0` should be converted to `Exp Double`. That means we will have

$$
\begin{array}{lcl}
[\![\,\texttt{Var x}\,]\!]\ \mathsf{E}\ \tau_1 & = & (\mathrm{x} : \tau_2) \rightsquigarrow \tau_1 \qquad \text{if } \mathsf{E}[\mathrm{x}] = \tau_2, \text{ otherwise } \textbf{fail} \\
[\![\,\texttt{I i}\,]\!]\ \mathsf{E}\ \tau & = & (\mathrm{i} : \texttt{Plain IntT}) \rightsquigarrow \tau \\
[\![\,\texttt{D d}\,]\!]\ \mathsf{E}\ \tau & = & (\mathrm{d} : \texttt{Plain DoubleT}) \rightsquigarrow \tau \\
[\![\,\texttt{Inf}\,]\!]\ \mathsf{E}\ \tau & = & (\textit{infinity} : \texttt{Plain DoubleT}) \rightsquigarrow \tau \\
[\![\,\texttt{Fn x}\ \tau_1^*\ e\,]\!]\ \mathsf{E}\ \tau_2 & = & \texttt{\textbackslash x -> } [\![\,e\,]\!]\ \mathsf{E}[\mathrm{x} \mapsto \tau_1]\ \tau_2 \\
[\![\,\texttt{Neg } e\,]\!]\ \mathsf{E}\ \tau_1 & = & (\texttt{-(}\epsilon\texttt{)} : \tau_2) \rightsquigarrow \tau_1 \\
 & & \quad \text{where } (\epsilon, \tau_2) = \textit{cancelLift}\ (\texttt{Exp } \kappa_{\tau_1})\ ([\![\,e\,]\!]\ \mathsf{E}\ (\texttt{Exp } \kappa_{\tau_1})) \\
[\![\,\texttt{Op x } \iota\ [\vec{e}\,]\,]\!]\ \mathsf{E}\ \tau_1 & = & (\epsilon : \tau_2) \rightsquigarrow \tau_1 \\
 & & \quad \text{where } (\epsilon, \tau_2) = \textit{convertOp}\ \mathrm{x}\ \iota\ [\vec{e}\,]\ \tau \\
[\![\,\texttt{Let x } \tau_1^*\ e_1\ e_2\,]\!]\ \mathsf{E}\ \tau_2 & = & \texttt{let x = } \epsilon \texttt{ :: } \tau_3 \texttt{ in } [\![\,e_2\,]\!]\ \mathsf{E}[\mathrm{x} \mapsto \tau_3]\ \tau_2 \\
 & & \quad \text{where } (\epsilon, \tau_3) = \textit{cancelLift}\ \tau_1\ ([\![\,e_1\,]\!]\ \mathsf{E}\ \tau_1) \\
[\![\,\texttt{Vc } [\vec{e}^{(n)}\,]\,]\!]\ \mathsf{E}\ (\texttt{Acc 1 } \kappa) & = & \texttt{use (fromList (Z :. } n\texttt{) } [\vec{\epsilon}\,]\texttt{) :: Acc (Vector } \kappa\texttt{)} \\
 & & \quad \text{where } \vec{\epsilon} = \epsilon_0, \epsilon_1, \ldots, \epsilon_{n-1} \\
 & & \qquad\quad \epsilon_i = [\![\,e_i\,]\!]\ \mathsf{E}\ (\texttt{Plain } \kappa)
\end{array}
$$

Figure 4: Translation of TAIL expressions to Haskell code.

to lift the double to `Exp` using the `constant` function. But to be able to use `a` in the vector literal its type needs to be plain `Double`. Since we cannot cast from `Exp` to `Plain` we need to save `a` as `Plain`. Unfortunately, this does not work in every case. If we instead of the literal `5.4` had `firstSh(iotaSh(5))` the compilation would fail since `firstSh` returns an `Exp` value we can't convert a `Plain Double`.

Then we have the *convertOp* function that does all the dirty work of translating function calls.

$$(\epsilon, \tau_2) = \textit{convertOp}\ \mathrm{x}\ \iota\ [\vec{e}\,]\ \tau_1$$

The input are the function name as a string, the instantiation lists, the arguments and the expected type of the result. The return value is a tuple with the resulting Haskell expression and the actual type of the expression which might not match the expected type τ. What the function does is that it looks the function name up in table of all available primitive functions. If the function name is found, the table entry is a function that, given ι and τ_1, returns information about how to convert each of the arguments and what the return type will be. The argument expressions are then converted according the specification and the everything is combined into the final expression.

When we have the translated the program, all that is left is to plug the Haskell expression into a Haskell module with the correct imports and `main` function that uses an Accelerate backend to execute the program.

5 Primitives

We should now a valid Haskell module that performs the same computation as the TAIL input. The output we get from the APLACC compiler cannot stand on its own. To run the program we need Accelerate implementations of the primitive functions. The following list shows which primitives have been implemented and should work like their APL counterparts and which primitives are still missing or only works for some cases:

Fully implemented: `add{i,d}`, `sub{i,d}`, `divd`, `mul{i,d}`, `min{i,d}`, `max{i,d}`, `neg{i,d}`, `resi`, `i2d`, `iota`, `each`, `reduce`, `shape`, `reshape0`, `reshape`, `cat`, `cons`, `snoc`, `zipWith`, `rotate`, `transp`, `first`.

Partially implemented: `take` and `drop` (does not support negative arguments and taking more elements than available by padding with default element), `reverse` (does not work for scalar arrays).

Not implemented: `transp2`.

Some of these primitives also have version that operates on shapes. They are: `shapeSh`, `takeSh`, `dropSh`, `consSh`, `snocSh`, `firstSh`, `iotaSh`, `rotateSh` and `catSh`. Since shapes are converted to arrays most of these are just aliases of their non-shape counterparts.

As an example of a primitive function here is the implementation of `rotate`:

```
rotate :: (Shape sh, Slice sh, Elt e)
       => Exp Int -> Acc (Array (sh :. Int) e)
       -> Acc (Array (sh :. Int) e)
rotate n arr =
  let sh = Acc.shape arr
      m  = Acc.indexHead sh
      idx sh = Acc.lift
                 (Acc.indexTail sh :.
                  mod (Acc.indexHead sh + n) m)
  in Acc.backpermute sh idx arr
```

6 Shapes

Until now we have treated shapes in TAIL as integer vectors. But another possibility would be to convert TAIL shapes into Accelerate shapes. Unfortunately, there seem to be some road blocks. Consider how to implement `iotaSh`. If we just blindly translate the TAIL type we quickly run into trouble:

```
iotaSh ::  Exp Int -> Exp (Z :. ...)
```

In Accelerate the dimensionality of the shape is not specified as a single integer but is instead derived from the length of the snoc list.

While it may be difficut or even impossible to construct arbitrary shape lists we can still optimize some uses of shapes. Take for instance the following program snippet:

```
let s:Sh(3) = shape{[int],[3]}(a) in
reshape{[int],[2,3]}(s, b)
```

Here we should convert `a` to be an Accelerate shape. This is similar to the problem with `Plain/Exp` for which we had the *cancelLift* function. Perhaps a similar solution could work in this case where instead of to compiling to this:

```
let x = shToArray (ϵ) :: Acc (Vector Int)
```

we would cancel out the `shToArray` and instead compile to something like the following:

```
let x = ϵ :: Exp DIMn
```

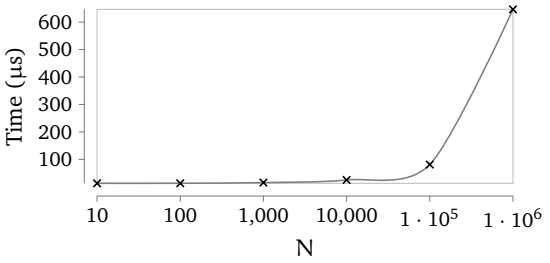

Figure 5: Benchmark of the `signal` program for various input sizes. The time measured is the time spent executing on the GPU without counting the time spent transferring data.

7 Performance

We would like to evaluate Accelerate as target for compiling APL. Creating a meaningful benchmark for GPU programs can be hard though. First one has to decide what should actually be measured. If we just measure the running time of the whole program we will include time spent setting up the GPU and transferring data which may dominate the time spent actually computing the result. Even if one is only measuring computation time on the GPU one has to be careful that data isn't getting generated on the GPU.

With these things in mind I had a go at making a tiny benchmark. I chose the `signal` program from the APLTAIL test suite. The TAIL program can be seen in Figure 6. The benchmark was run on a server with a nVidia GeForce GTX 690. Running time was measured using the `nvprof` profiler. For each input size the program was run three times and the average of three running times was recorded. Only the time spent running computation on the GPU was counted in the total running time and not any data transfer. The profiler shows amount of data transferred to the GPU which confirmed that the data was not generated on the GPU.

Figure 5 shows the result of the benchmark. We see that the program is quite fast—even with 1 million data points the running time is less than 1 milisecond. For comparison the running time of the whole program including setup was around 5 seconds, so setting up the GPU takes a considerable amount of time. The running time of the program seem to scale well.

8 Conclusion and Future Work

I have presented a compiler that through a typed intermediate language can compile a subset of APL to Accelerate that can be run using a GPU.

Future work could be to increase the subset of APL that can be compiled. Work is already being done on APLTAIL compiler to make it accept more APL programs. This report also highlights some areas where the APLACC compiler is still lacking. Implementation of the missing primitives and better handling of shapes are things that could be worked on.

Finally it would be interesting to see the results of a more comprehensive benchmark.

9 References

[1] Michael Budde. *APL to Accelerate compiler through a typed IL*. URL: https://github.com/mbudde/aplacc.

[2] Martin Elsman. *APL Compiler targeting a typed array intermediate language*. URL: https://github.com/melsman/apltail.

[3] Martin Elsman and Martin Dybdal. "Compiling a Subset of APL Into a Typed Intermediate Language". In: *Proceedings of the 1st ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. ARRAY'14. ACM, 2014.


```
let v0:Sh(100) = iotaSh(100) in
let v3:Sh(101) = consSh(0,v0) in
reduce{[double],[0]}(addd, 0.00,
 each{[double,double],[1]}(
  fn v11:[double]0 => maxd(i2d(~50), v11),
  each{[double,double],[1]}(
   fn v10:[double]0 => mind(i2d(50), v10),
   each{[double,double],[1]}(
    fn v9:[double]0 => muld(i2d(50), v9),
    zipWith{[double,double,double],[1]}(divd,
     each{[int,double],[1]}(i2d,
      drop{[int],[1]}(1,
       zipWith{[int,int,int],[1]}(
        subi, v3, rotateSh(~1,v3)))),
     each{[double,double],[1]}(
      fn v2:[double]0 => addd(0.01, v2),
      each{[int,double],[1]}(i2d,v0)))))))
```

Figure 6: The `signal` test program.