FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Compiling TAIL to Futhark

An adventure in compiling functional data-parallel constructs

Bachelor's thesis

Anna Sofie Kiehn (mvq686) & Henriks Urms (mgs837)

Supervisor: Martin Elsman

June 8, 2015

Abstract

We present an implementation independent scheme for compiling a subset of the intermediate array language TAIL to the functional programming language Futhark, preserving the data parallelism of the host language by using built-in data parallel functions in the target language to express the TAIL operations. We also present an implementation of the compilation scheme using this implementation to demonstrate the usefulness of compiling TAIL to Futhark by comparing the execution time of selected benchmarks on sequential back-ends to both languages.

Resumé

Vi præsenterer et implementations uafhængigt oversættelses skema for en delmængde af det intermediære array sprog TAIL til det funktionelle sprog Futhark der bibeholder den data parallelisme der er i TAIL ved at bruge indbyggede data parallele funktioner i Futhark til at udtrykke TAIL operationerne i. Vi præsenterer også en implementation og bruger implementationen til at demonstrere anvendeligheden af at oversætte TAIL til Futhark ved at sammenligne udførselstiden af udvalgte benchmarks på sekventielle backends til begge sprog.

Contents

1 Introduction

In this report we examine if it is possible, effectively to compile TAIL programs, produced by the APLTAIL compiler, into Futhark programs and thereby make use of the Futhark infrastructure for optimization and the possibility for targeting parallel hardware.

In recent years, there has been much focus on leveraging the power of parallel hardware. One approach has been to design programming languages with explicit data-parallel constructs that can be compiled into highly parallel code. One such language is Futhark [7]. The aim of Futhark is to target parallel hardware such as GPUs while still being the target of more programmer-productivity oriented languages. The Futhark compiler performs several optimizations, such as fusion, which enhance the degree of parallelism [10] [9] [8].

APL was created in the 1960's by Kenneth E. Iverson, and is an array programming language. Its main type is the multi-dimensional array and most of the built-in functions in the language are array operators that work on this type. Most of its built-in functions or operators are represented by unicode symbols allowing for very concise code. The APL language is dynamically typed. It supports first and second order functions and these functions work on arrays of any rank and base type. APL features a large set of built-in operations, which, through 50 years of history, have shown to be suitable for a large range of applications for example in the financial world where large code bases are still operational and actively developed [6] [2].

Efforts in compiling APL to parallel backends already exist in for example the form of the language TAIL (Typed array intermediate language) and its compiler [6] that compiles a subset of APL. The APLTAIL compiler captures the parallelism inherent in APL source code and brings it to a much more manageable form.

In our work we provide a compiler from TAIL to Futhark thus bridging the gap between APL and Futhark.

The compilation between TAIL and Futhark is described in terms of a compilation scheme, which is the main contribution of this work. Figure 1 gives an overview of the main compilers involved in this project and the code they produce. The figure gives an overview of how our compiler (the TAIL2Futhark compiler) fits between the already existing APLTAIL compiler, which compiles APL to TAIL code, and the Futhark compiler that compiles Futhark to either sequential or parallel code C-code [7].

A major motivation for this work is that compiling APL to Futhark through TAIL the Futhark compiler can be used to generate parallel code from APL once a parallel back-end for Futhark is completed.

One of the main point of interest in the compilation between TAIL and Futhark is compiling the four array operators of TAIL: `each`, `eachV`, `reduce` and `zipWith` to Futhark source code, which involves the four second-order array combinators in Futhark: `map`, `filter`, `reduce` and `scan` [6] [7]. However as the functionality of these functions is entirely different the work lies in creating a mapping map the parallelism in the original code to parallel constructs in the target language. This can be seen in the example below which illustrate the difference between the functions. The APL code is given first. We do not describe APL in detail but the comments on each line explain what happens.

```
a ← 2 2 ⍴ 2 3 4 5     ⍝  make a 2x2 matrix
b ← ×/ a              ⍝  multiply the elements in each row
+/ b                  ⍝  add the products together
```

The APL code becomes the following TAIL code when using the APLTAIL compiler and now contains type information. The reason for the `i2d` (int to double) operator is that the APLTAIL compiler only accept programs that returns doubles at the moment.

```
let v1:[int]2 = reshape{[int],[1,2]}([2,2],[2,3,4,5]) in
let v4:[int]1 = reduce{[int],[1]}(muli,1,v1) in
i2d(reduce{[int],[0]}(addi,0,v4))
```

This TAIL code is then compiled to Futhark code where the reduce function is mapped to a nested reduce function in the Futhark language.

```
fun real main() =
  let t_v1 = reshape((2,2),reshape1_int((2 * (2 * 1)),reshape(((
      size(0,[2,3,4,5]) * 1)),[2,3,4,5]))) in
  let t_v4 = map(fn int ([int] x) => reduce(*,1,x),t_v1) in
  toFloat(reduce(+,0,t_v4))
```

The nesting of the operator happens because the reduce function in APL and therefore TAIL works on the innermost dimension of the array but the reduce function in Futhark works on the outermost dimention of the array. In order to get the same functionality, namely reducing the content of the inner arrays, the Futhark function have to be mapped onto them. This can be seen in the definition of the `t_v4` variable. The function `reshape1_int` is a library function that will be explained later.

This report contributes with a compilation scheme that is implementation independent, showing a replicable way of how to translate TAIL, to the functional language Futhark. Also, this report presents an implementation of the previous mentioned scheme in Haskell. The effectiveness of this implementation has been tested by comparing benchmark results on code generated by the C-backend to TAIL and the generated Futhark source code by using Futhark's back-end. The project is open source and the source code can be found at:
https://github.com/henrikurms/tail2futhark. Both Futhark and TAIL are ongoing research projects and are therefore subject to change. Thus the references cited may not be up to date (the versions of the languages used in this project was the versions available on github from Februar 2015 until early May 2015). For a up to date version of the languages and their compilers we refer to their resepctive github repositories (links for these repositories can be seen below):

TAIL: https://github.com/melsman/apltail
Futhark: https://github.com/HIPERFIT/futhark

The reader of this report is assumed to have understanding of computer science concepts of the bachelor level and therefore general computer science concepts (e.g. parser and compiler) will not be explained.

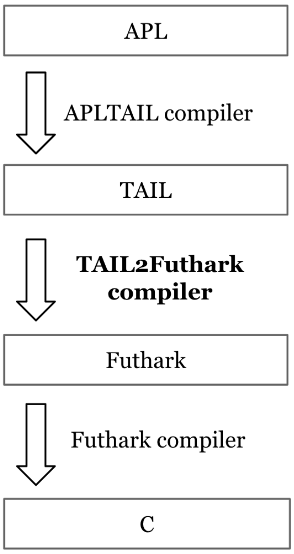

Figure 1: The three compilers involved in this project and the code they produce

1.1 Scope

In this project we create an implementation independent compilation scheme showing a compilation between TAIL and Futhark as well as a Haskell implementation of the compilation scheme, creating the TAIL2Futhark compiler. We also test the implementation of the compiler.

We have used selected benchmarks that we will adapt to work with our project and present their results in section 8. We will use benchmarks to see if the compiled Futhark code is more efficient than the original code.

We implement a subset of the TAIL language so not all TAIL operators are supported by us. Also we have worked with the version of TAIL that was published before February 2015 and up until early May 2015.

We will not do a detailed analysis of the results of the benchmarks or discuss the optimization that influence their running time.

We will not present an overview of APL but only refer to [11] and [13].

1.2 Report outline

The following sections of this report is structured as follows. Section 1 includes the introduction containing the scope and methods and tools used in this project. Section 2 and Section 3 gives an introduction to the source and the target language respectively. Section 4 presents the overall strategy for compiling TAIL to Futhark is given. Section 6 describes the compilation scheme in detail. Section 7 is an overview of the Haskell implementation and tests. Section 8 describes the benchmarks used to measure the efficiency of the generated Futhark code by comparing it to the TAIL back-end. Finally, Section 9 and Section 10 provide a discussion and a conclusion of the results and contain ideas to possible future work.

1.3 Methods and tools

In this section we will describe and explain the reasoning behind the methods and tools we have used in the project.

1.3.1 Compilation scheme and the notation

In this report, a compilation scheme done in a form of mathematical notation is presented. The reason for using a mathematical notation is to be able to express the compilation of the different components of the compilation separately and in a detailed precise manner. We call the compilation of a specific component a conversion rule. The notation should also help the reader getting an overview of the entire main part of the compiler as well as create a way of talking about specific conversions. The notation is inspired by similar notation used in other projects [14] [5] to describe compilation schemes but is not built on a specific standard as no such standard is known to us. Instead, we have invented our own notation.

The scheme gives a conceptual understanding of the compilation that are not cluttered by implementation details. The scheme simply illustrates the concepts of the compilation and is implementation independent. It should therefore be possible to use the scheme to create another implementation of the compiler.

Having the compilation scheme also make the implementation easier because it helps to structure the implementation.

1.3.2 Library functions

To keep the implementation scheme simple, we have made a small library of functions, which we present in Section 5. We have coded the library functions in the compiler itself for several reasons. One reason is that we would like the compiler to always output a valid (runnable) Futhark program given a valid TAIL input program, so we would like to be able to include the library in the output when we run the compiler. Furthermore, since Futhark is a statically typed language with no polymorphism, we would like to be able to generate functions with the same implementations but different types from a template. That way we can be sure the different versions have the same implementation. Finally, because we expect future versions of Futhark to feature polymorphism and a module system, we would like the solution to be easy to remove once it is no longer needed [7].

1.3.3 Choice of language for implementation

The implementation described in this project is written in the functional programming language Haskell. The language constructs in Haskell are similar to our mathematical notation and functional languages are good for developing compilers in general [14].

1.3.4 Other tools

For building our project and managing external libraries, we have used the cabal packaging system [12]. The cabal packaging system is the standard build architecture for Haskell and should make it easy to build our code.

We have created a Makefile for building our benchmarks. This made it much easier for us to rebuild the benchmarks and can also be used as a reference of how to build them manually.

We have used the Linux command-line tool `time` for measuring the runtime of our benchmarks. It is not necessarily the best way but because of time constraints we have not looked for another solution. One reason it is not ideal is because it also includes the time spend on reading data from files. We have however tried to create benchmarks where the execution of the computations overshadow any overhead introduced by input and output. In particular only, one of our benchmarks read input from files and the measured difference between `time` and a built-in timing function of the program only differed by 1 ms.

1.3.5 Modifying an existing parser

The parser we used for this project is not done by us but was created in another project that also worked with compiling TAIL to a parallel back-end [1]. The latest version of the parser can be found in the github repository: https://github.com/mbudde/aplacc. We did therefore not create the parser ourselves, instead we modified the existing parser where needed which enabled us to focus our work on the core of our project.

2 TAIL

In this section we present an overview on the language TAIL [6].

The syntax of types in TAIL can be seen below. Types are divided into base types (κ), shape types (ρ), types (τ), and type schemes (σ).The letter i denotes an integer scalar value and the letter α, and the letter γ denotes type variables and shape variables, respectively.

$$\begin{aligned}
\kappa &::= \texttt{int} \mid \texttt{double} \mid \texttt{bool} \mid \alpha \\
\rho &::= i \mid \gamma \mid \rho + \rho' \\
\tau &::= [\kappa]^{\rho} \mid \langle\kappa\rangle^{\rho} \mid \texttt{S}_{\kappa}(\rho) \mid \texttt{SV}_{\kappa}(\rho) \mid \tau \to \tau' \\
\sigma &::= \forall\vec{\alpha}\vec{\gamma}.\tau
\end{aligned}$$

The type system of TAIL supports array types ($[\kappa]^{\rho}$) that keeps track of the rank of the array in its type. The integer scalar in the array's shape type denotes the rank of the array and must be a non-negative integer. The type system also supports vector types ($\langle\kappa\rangle^{\rho}$), which are used specifically to denote vectors of a specific length. For example, `<int>8` denotes a vector of ints of known length 8. If a vector's length is not statically known, it can instead be expressed as an array of rank 1. Scalar values that are statically known can be given the type ($\mathrm{S}_{\kappa}(\rho)$), which represents integers, and booleans, for which the value is contained in the type. In addition, there also exists single-element integer, double, and boolean vector types ($\mathrm{SV}_{\kappa}(\rho)$) for singleton vectors where the element is statically known. Finally there exists function types ($\tau \to \tau'$).

The type system makes use of substitution in order to express instances of type schemes (σ). A type substitution (S_t) maps type variables to base types and shape substitution (S_s) maps shape variables to shape types. A general substitution (S) is a pair (S_t,S_s) of a type substitution and a shape substitution. Using the substitution S on an object B means applying both S_t and S_s on objects in B. A type τ' is an instance of a type scheme $\sigma = \forall\vec{\alpha}\vec{\gamma}.\tau$ (written $\sigma \geq \tau'$) if a substitution S exists such that $S(\tau) = \tau'$. All type schemes are assumed closed.

The syntax of operators and expressions is given below. The letter x is used to denote program variables.

```
// operators
op ::= addi | subi | multi | mini | maxi | addd | subd |
       muld | mind | maxd | andb | orb | xorb |  nanb |
       norb | notb | lti | ltei | gti | gtei | eqi | neqi |
       ltd | lted | gtd | gted | eqd | neqd | iota | each |
       reduce | i2d | b2i | reshape0 | reshape | rotate |
       transp | transp2 | zipWith | shape | take | drop |
       first | cat | cons | snoc | shapeV | catV | consV |
       snocV | iotaV | rotateV | takeV | dropV | firstV

//expressions
e ::= v
    | x
    | [e⃗]
    | e e′
    | let x = e1 in e2
    | op(e⃗)

// values
v ::= [a⃗]^δ
    | λx.e
```

A TAIL program always consists of a single expression. An expression e can then be a value, a variable, a list of expressions, a let expression or an operator. Each TAIL operator has a unique type scheme.

One of the operators with a simple type scheme is the binary operator maxi that takes two arguments a and b and evaluates to the argument with the highest value. Its type scheme is as follows:

$$\texttt{maxi : int} \to \texttt{int} \to \texttt{int}$$

Other operators have more complex type schemes. Examples of those are the parallel operators. There are four parallel operators in the subset of TAIL that we consider, namely `each`, `eachV`, `reduce` and `zipWith`. The functions `each` and `eachV` are known in many languages as map. The type scheme for the function `each` is:

$$\texttt{each :} \ \forall\alpha\beta\gamma.(\alpha \to \beta) \to [\alpha]^{\gamma} \to [\beta]^{\gamma}$$

Given a function f and an array a, the application `each(f,a)` evaluates to an array where f is applied to each element of a giving the value $[f(a_1), .., f(a_n)]$. If the rank of the array is greater than 1 the `each` function works as a map on the fattened representation of the array, that is, the function is applied on the inner most dimension of the array, or seen in another way, on each basic value.

The `eachV` function is a special case of `each` and is used on vector types.

The function `reduce` works similarly to fold known from functional languages. The type scheme for `reduce` is:

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

The function takes as arguments an associative binary operator op (for instance `addi`), a neutral element n, (for instance 0) and an array a. The function application evaluates to the reduction of the elements using the operator. An array of rank $\gamma + 1$ is reduced to an array of rank γ along the inner-most dimension. Unlike fold, reduce makes no guarantees as to the order of application of the operator. Therefore, the operator has to be associative and the provided element has to be neutral, which is of course necessary for parallel execution.

The `zipWith` function's type scheme is given as follows:

```
zipWith   : ∀α₁α₂βγ.(α₁ → α₂ → β) → [α₁]^γ → [α₂]^γ → [β]^γ
```

Given a function f that works on a pair (x, y) and two arrays a and b, `zipWith(f,a,b)` evaluates to an array where the i'th element is f applied to the pair (a_i, b_i) Like the other three operators, it works on the inner-most dimension of the array [6].

There are other important operators besides the parallel ones. One of them is the operator `reshape(`a_1`,`a_2`)`. Given two arrays, it reshapes the flattened representation of the second array a_2 to the shape given by the first array, thus $reshape([2, 3], [1, 2, 3, 4, 5, 6])$ evaluates to $[[1, 2, 3], [4, 5, 6]]$. $reshape([2, 3], [1, 2, 3, 4, 5, 6])$ evaluates to $[[1, 2, 3], [4, 5, 6]]$. If a_2 is too long the elements not needed are dropped. That is, $reshape([2, 3], [1, 2, 3, 4, 5, 6, 7, 8])$ would evaluate to the same as the first example. If a_2 is shorter than needed the elements of a_2 are repeated. That is $reshape([2, 3], [1, 2, 3])$ evaluates to $[[1, 2, 3], [1, 2, 3]]$. Notice that this is not how arrays are represented in TAIL. Instead of using nested brackets to represent the dimensions, arrays in TAIL are represented with a shape (i.e. $[1, 2, 3, 4, 5, 6]^{[2,3]}$). However, using this representation can make what happens less obvious so we use the nested brackets representation instead.

Other important operator expressions are `take(`i`,`a`)` and `drop(`i`,`a`)`. They return an array containing the 1st to ith element of a, and the array containing the i'th to nth element of a, respectively. If the array is multi-dimensional, the operators work on the outermost dimension of the array. That is, $take(2, [[1, 2], [3, 4], [5, 6]])$ evaluates to $[[1, 2], [3, 4]]$. If the array contains too few elements, the array is padded with zeros, whereas the `drop` operator returns the empty array in the case that more elements are dropped that than a contains.

The operator `snoc(a,e)` takes two arrays a and e and returns an array where the i'th element of e is appended onto the end of the i'th row of a. If there are too few elements in e an error occurs, except if there is only one element in e in which case the operator evaluates to an array where the one element from e is appended onto each row of a.

The operator `cons(e,a)` has very similar semantics as the `snoc` operator. The only difference is that it appends the contents of e not on the end but at the beginning of each row.

The operator `cat(`a_1`,`a_2`)` takes two arrays that have to have the same outer dimension and returns an array where the i'th element (i.e., a row if the array is two-dimensional) of a_2 is appended onto the end of the i'th element of a_1.

The `transp` operator takes an array and returns the transposed array. For instance, $transp([[1, 2, 3], [4, 5, 6]])$ evaluates to $[[1, 4], [2, 5], [3, 6]]$. If the array is multi-dimensional (i.e., a three-dimensional array with the shape $2 \times 3 \times 4$), the function returns an array with the shape $4 \times 3 \times 2$.

TAIL was designed with the purpose of targeting parallel architectures such as GPUs and allows parallel programs to be expressed in a highly abstract manner. The TAIL compiler can also efficiently compile TAIL code into sequential code in a C-like language. The subset of APL operators that TAIL support are shown earlier in this section.

The language TAIL is statically typed and supports polymorphism. Most of the operators in TAIL are very general. That is, they are polymorphic with respect to array ranks and base types. Although for some operations a specific type is needed. An example is the `take` function. It takes as argument a number (of type int) and an array of type $[\alpha]^\gamma$. The TAIL compiler infers types for the values in the APL program and can annotate polymorphic bindings with instance declarations. Instance lists provide the base types and ranks of arrays involved in operations.

TAIL's type system takes the dynamic types of APL and transforms it to a more manageable form adding explicit type information to the constructs. Another benefit of the expressiveness of TAILs type system is that it allows the (TAIL) compiler to express some operators that are primitive in APL using simpler operators. One such operator is that of the inner product [6].

The aplacc parser for TAIL represents the TAIL expressions in the abstract syntax tree as variables, constants, infinity, the negative representation of the expression, let expressions, operators and lambda expressions.

Generally it is not possible to define higher-order lambda expressions in TAIL, however higher-order operators may use currying of lambda expressions to express multi-argument functional arguments. This means that lambda expressions that return lambda expressions can occur as arguments in higher-order operator applications and nowhere else.

For details about the TAIL types system, see [6].

3 Futhark

In this section we give a short introduction to the Futhark language. We will only cover the parts necessary to understand the reasoning behind our compilation approach. For the full language reference please consult [7].

The syntax of Futhark types can be seen below.

```
t  ::= int            (Integers)
     | real           (Float)
     | bool           (Booleans)
     | char           (Characters)
     | {t1 ,... ,tn}  (Tuples)
     | [t]            (Arrays)
     | *[t]           (Unique arrays)
```

The types in Futhark consist of: integers, floating points, booleans, chars, tuples (`{t1,...,tn}`), arrays (`[t]`), and unique arrays (`*[t]`). Tuple types are written as a comma separated list of types surrounded by braces. For example `{int,bool}` represents pairs of integers and booleans. Unlike TAIL, Futhark allows nesting of arrays. Indeed, nested array types are how multi-dimensional arrays are expressed in Futhark. Array types are denoted by the elements (base) type enclosed by brackets. The layer of brackets indicates the dimensionality of the array type. For instance `[int]` is a one-dimensional array of integers, and `[[[bool]]]` is a tree-dimensional array of booleans. Arrays must be regular. That is, all sub arrays in an array must have the same number of elements.

The Futhark language is statically typed but does not use type inference. Also, the type system of Futhark is not able to express polymorphism. This means that it is not possible to make polymorphic functions in Futhark. The exception to this rule is that a lot of the built-in functions can be used on multiple types.

The syntax of Futhark expressions is show below as follows:

```
k ::= n               (Integer)
    | d               (Decimal number)
    | b               (Boolean)
    | c               (Character)
    | {v1 ,... ,vn}   (Tuple)
    | [v1 ,... ,vn]   (Array)
```

```
e ::= k                                  (Constant)
    | v                                  (Variable)
    | {e1,...,en}                        (Tuble expression)
    | [e1,...,en]                        (Array expression)
    | e1 ⊙ e2                            (Binary operator)
    | -e                                 (Prefic minus)
    | !e                                 (Logical negation)
    | if e1 then e2 else e3              (Branching)
    | v[e1,...,en]                       (Indexing)
    | v(e1,...,en)                       (Function call)
    | let p = e1 in e2                   (Pattern binding)
    | zip(e1,...,en)                     (Zipping)
    | unzip(e)                           (Unzipping)
    | iota(e)                            (Range)
    | replicate(en,ev)                   (Replication)
    | size(i,e)                          (Array length)
    | reshape((e1,...,en),e)             (Array reshape)
    | transpose(e)                       (Transposition)
    | split(e1,e2)                       (Split e2 at index e1)
    | concat(e1,e2)                      (Concatination)
    | let v1 = v2 with                   (In-place update)
```

```
               [$e_1, ..., e_n$] <- $e_v$ in $e_b$
     | loop ($p$ = $e_1$) = for $v$ < $e_2$ do   (Loop)
               $e_3$ in $e_4$
```

```
$p$ ::= $id$                          (Patterns)
    | {$p_1$, . . . ,$p_n$}
```

```
$fun$ ::= fun t v(t$_1$ v$_1$,...t$_n$ v$_n$) = $e$
```

```
$prog$ ::= $\epsilon$ | $fun$ $prog$
```

```
$l$ ::=  fn $t$ ($t_1$ $v_1$,...,$t_n$ $v_n$) => $e$   (Anonymous function)
    | $id$ ($e_1$,...,$e_n$)              (Curried function)
    | op $\odot$ ($e_1$,...,$e_n$)            (Curried operator)
```

```
$e$ ::= map($l$, $e$)
    | filter($l$, $e$)
    | reduce($l$, $x$, $e$)
    | scan($l$, $x$, $e$)
```

Notice that the syntactical construct denoted by l can only occur in `map`, `filter`, `reduce` and `scan`. The functions `map`, `filter`, `reduce` and `scan` are second-order array combinators, or SOACs for short.

The SOACs operate on arrays with first-order functions given as arguments. Functional arguments used can be function names of first-order functions (either user-defined or built-in), binary operators, or lambda expressions. Furthermore, in a SOAC expression, operators and functions can be curried. Lambda expressions require explicit type annotations for the return type and argument types, and argument bindings follow the normal shadowing rules.

We do not target the SOACs `filter` and `scan` in our compilation, and we will therefore not discuss them in detail here. The SOACs can be used on arrays of any type even though it cannot be expressed by Futhark types. For clarity we give the type for each SOAC that it would have had in a polymorphic language. Below we shortly discuss `map` and `reduce`.

The function `map` has the following type:

```
map : $\forall\alpha\beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$
```

The function `map(`l`,`a`)` takes a function l and an array a and evaluates to the array consisting of l applied to each element of a. In contrast to TAIL, if the array is multi-dimensional the function is applied to the outer-most dimension. This means that if the function l is mapped onto a 2-dimensional array, the function would be applied to an array not the elements of the array.

The type of the function `reduce` is:

```
reduce : $\forall\alpha.(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to \alpha$
```

Given a binary operator/function f, the neutral element e of f and an array a, `reduce` evaluates to the result of applying f to combine all the elements of a, that is,

```
$e \odot a[0] \odot \ldots \odot a[n]$ where $x \odot y = f(x,y)$
```

Like `map`, `reduce` applies the function on the outer-most dimension of the array [7].

The first-order segment of Futhark has many of the typical language features like constants, variables, many of the usual binary operators, branching, array indexing and some additional features like in-place updates and looping, which we do not use.

Futhark features array zipping with the built-in `zip`, which produces an array of pairs from a pair of arrays. The resulting arrays can then be mapped over with binary operators such as $+$.

The `iota` function, given an integer n, produces an array with integer values ranging from 0 to $n-1$. The `replicate` function, given an integer n and an array a, returns an array consisting of n copies of a. The `size` primitive will, given a positive integer i and an array a, return the i'th dimension, or put in another way the length of the arrays nested with depth i in a. Recall that these arrays will all have the same length. The `reshape` function takes a number of dimensions $(dim_1, ..., dim_n)$ and an array a and returns an array where the elements of a is reshaped into the shape specified by the list of dimensions. The number of elements in a must be equal to the product of the dimensions (i.e. $elements\ of\ a = dim_1 * ... * dim_n$).

The function `transpose` takes an array a and returns the transposed a. Transposing a three-dimensional array with dimensions $2 \times 3 \times 4$ is not like in TAIL an array with dimensions $4 \times 3 \times 2$ but instead an array with dimensions $3 \times 4 \times 2$.

The function `split`, given an integer n and an array a, partitions a into two arrays $a[0, .., n]$ and $a[n+1, ...]$ and returns them as a tuple. The function `concat` takes two arrays and concatenates them by concatenating the row/elements of one array with another. The shape of the two arrays have to be the same except in the first dimension.

The undocumented `rearrange` function takes as arguments a comma separated list of dimensions (surrounded in parentheses) and an array. It then rearranges the shape of the array to the by the list specified.

The aim of Futhark is to be an attractive choice for expressing complex parallel programs. This goal is pursued by featuring high expressive power without losing the ability to do aggressive optimization and managing parallelism. This is a challenge because higher expressive power means optimizations become more difficult. However, Futhark does support nested parallelism as this is a feature many programs depend upon even though it does make optimization more difficult [7].

4 The compilation strategy

In this section we will present our general strategy for compiling TAIL to Futhark.

Where possible, TAIL primitives have been mapped directly to their corresponding versions in the Futhark language. Where direct translation is not possible, the approach has been to use existing operations as much as possible and generate code to bridge the gap.

The general strategy for compiling TAIL expressions was to aim for the simplest conversion and use as much as possible the built-in functions of Futhark to make it easy for the Futhark compiler to optimize away the overhead that the compilation from TAIL to Futhark creates. This means that we have not directly focused on optimization in the compilation. Also, as it was not in the scope of this project. Still, we have tried as much as possible not to introduce any unecessary inefficiencies.

In the cases where it was not possible to use built-in Futhark functions, library functions was created instead.

Many of the monomorphic first-order functions of TAIL are mapped directly to a library function of the same name. This also allows us to use the same mapping when the functions occur as arguments in SOAC applications.

5 Library functions

In this section explain some of the nontrivial library functions we have defined and discuss their usefulness. The rest of the library functions can be found in the end of this section.

5.1 The take1, drop1 and reshape1 functions

The `take1`, `drop1` and `reshape1` functions implement the TAIL operators `take`, `drop` and `reshape` in the one-dimensional case. In Section 6, we see how they can be used to implement the multi-dimensional cases. It is advantageous to use a library function for only the one-dimensional case as we would otherwise need a separate library function for each rank and basic type combination which we then needed to call since Futhark only allows declaration of monomorphic functions [7]. We have implemented the functions (take1, drop1, and reduce1) as templates written in Haskell. A template is a function that given a type returns Futhark code for that function with the given type. We have done this so we can use the same template for making all four functions (one for each base type) and can thereby be sure to have the same function code for each type and make maintaining the functions easier.

5.1.1 The take1 functions

The `take1` functions is defined as follow:

```
fun [int] take1_int(int l,[int] x) =
  if (0 <= l)
  then if (l <= size(0,x))
       then let {v1,_} = split((l),x) in v1
       else concat(x,replicate((l - size(0,x)),0))
  else if (0 <= (l + size(0,x)))
       then let {_,v2} = split(((l + size(0,x))),x) in v2
       else concat(replicate((l - size(0,x)),0),x)
```

Notice that this is the `int` version. The template, is as mentioned used to make a boolean, char, and double version as well. See Appendix B for the template function.

The function first checks if it should perform a positive of negative take and then checks whether it should split so it can return part of the argument or pad the argument with zeros based on whether the take size was smaller or bigger than the array.

5.1.2 The drop1 functions

The `drop1` functions is defined as follows:

```
fun [int] drop1_int(int l,[int] x) =
  if (size(0,x) <= if (l <= 0) then -l else l)
  then empty(int)
  else if (l <= 0)
       then let {v1,_} = split(((l + size(0,x))),x) in v1
       else let {_,v2} = split((l),x) in v2
```

Again we show only the int version.

5.1.3 The reshape1 functions

The `reshape1` function's int version can be seen below.

To adjust the array, we first make sure it is long enough by extending it using the function replicate and then truncate it to the correct length with split.

```
fun [int] reshape1_int(int l,[int] x) =
  let roundUp = ((l + (size(0,x) - 1)) / size(0,x)) in
  let extend = reshape(((size(0,x) * roundUp)),replicate(roundUp,x)) in
  let {v1,_} = split((l),extend) in v1
```

When we replicate an array in Futhark, the rank of the array increases by one, thus, we have to reshape the array back to rank 1 before we split it. The number of times we should replicate the array is the target size divided by the array size rounded up. This is computed in the variable roundUp. We add denominator plus one to the enumerator to round up as normal integer division rounds down.

5.2 Bool equality

Futhark has no bool equality so we implemented our own:

```
fun bool eqb(bool x,bool y) =
  (!((x || y)) || (x && y))
```

Two booleans are equal if they both are true or none of them are true.

5.3 Xor

Likewise there is no logical xor operation so we included it in the library:

```
fun bool xorb(bool x,bool y) =
  (!((x && y)) && (x || y))
```

The Xor of two booleans is true if one but not both of them are true.

5.4 All other library functions

The rest of the library functions are implemented very straightforward and are therefore only mentioned as a list here:

```
boolToInt,negi,negd,absi,absd,mini,mind,signd,signi,maxi,maxd,
nandb,norb,neqi,neqd,resi
```

The implementation of these functions can be found the compiler source code in Appendix B.

6 The compilation scheme

The main contribution of this work as mentioned earlier is the compilation scheme presented in this section. It shows a set of conversion rules of a subset of TAIL's syntax to Futhark source code. Also in this section the notation and the compilation of some of the nontrivial operators or expressions of TAIL is described in detail.

The main part of the compilation scheme that contains the expressions can be seen in Figure 2. In Figure 3 is the conversion rules for lambda expressions. In Figure 4 are the functions that are compiled directly to a corresponding function in Futhark and in Figure 5 are the compilation of the binary operators. Notice that the schemes in the above mentioned figures are all mutually recursive.

When e is some TAIL expression, and e' is some Futhark expression we specify the translation as conversion rules of the form $[\![e]\!] = e'$. The rules are syntax-directed in the sense that they follow the structure of e, recursively.

6.1 The notation

Each line in the scheme consists of a TAIL expression in double brackets $[\![\cdot]\!]$ on the left, followed by an equals sign in the middle and a Futhark expression on the right side. This means that the TAIL expression on the left side should be compiled to the Futhark expression on the right side. We call such a line a conversion rule. Some rules have side conditions after a comma which means some conditions must be met before that rule is legal; otherwise another rule must be chosen. This can be thought of as similar to pattern matching in functional languages where side conditions are guards. The rules are exhaustive and non-overlapping. In practice, the compilation can be implemented using pattern matching by choosing the right ordering of patterns and it is indeed how our compiler is implemented. We have tried to use such an ordering of rules in our presentation. An expression wrapped in double brackets can also occur on the right side of the equal sign, means that this expression should be compiled recursively as part of the compilation of the parent expression.

Some TAIL expressions have type information as part of their declaration in their instance lists. This type information is expressed in the compilation scheme as subscript to the expression. The type information can be either just a type t or a combination of both type and rank r. The type consist of a type that are one of the TAIL types described in Section 2. The rank is the number of dimensions.

Some of the rules are subscripted with either op, fun of fn. These are separate sets of rules that are invoked on the right hand side of regular rules. We also call such a set a rule. From context it will be clear what we mean when we say rule, for example when talking about the set of rules subscripted by fn we will just say: the $[\![\cdot]\!]_{fn}$ rule. The $[\![\cdot]\!]$ rule is also called the default rule.

$$
\begin{array}{lcl}
\llbracket x \rrbracket & = & x \\
\llbracket i \rrbracket & = & i \\
\llbracket d \rrbracket & = & d \\
\llbracket c \rrbracket & = & c \\
\llbracket -e \rrbracket & = & \text{-}\llbracket e \rrbracket \\
\llbracket \text{let } x : t = e_1 \text{ in } e_2 \rrbracket & = & \text{let } \llbracket x \rrbracket = \llbracket e_1 \rrbracket \text{ in } \llbracket e_2 \rrbracket \\
\llbracket [e_1, ..., e_n] \rrbracket & = & [\llbracket e_1 \rrbracket, ..., \llbracket e_n \rrbracket] \\
\llbracket \text{op}[e_1, e_2] \rrbracket & = & \llbracket e_1 \rrbracket \ \llbracket \text{op} \rrbracket_{op} \ \llbracket e_2 \rrbracket, \text{ op} \in \mathit{binops} \\
\llbracket \text{op}[e_1, ..., e_n] \rrbracket & = & \llbracket \text{op} \rrbracket_{fun} \ (\llbracket e_1 \rrbracket, ..., \llbracket e_n \rrbracket), \text{ op} \in funs \\
\llbracket \text{each}_{[t_1,t_2,r]}(f, a) \rrbracket & = & \begin{cases} \text{map}(\llbracket f \rrbracket_{fn}^{\llbracket t_2 \rrbracket}, \llbracket a \rrbracket) & r = 1 \\ \text{map (fn } t_2^r \ (t_1^r \ x) \texttt{ => } \llbracket \text{each}_{[t,r-1]}(f, x) \rrbracket, \llbracket a \rrbracket) & r > 1 \end{cases} \\
\llbracket \text{eachV}_{[t_1,t_2,r]}(f, a) \rrbracket & = & \text{map}(\llbracket f \rrbracket_{fn}^{\llbracket t_2 \rrbracket}, \llbracket a \rrbracket) \\
\llbracket \text{reduce}_{[t,r]}(f, n, a) \rrbracket & = & \begin{cases} \text{reduce}(\llbracket f \rrbracket_{fn}^{\llbracket t \rrbracket}, \llbracket n \rrbracket, \llbracket a \rrbracket) & r = 1 \\ \text{map (fn } t^{r-1} \ (t^r \ x) \texttt{ => } \llbracket \text{reduce}_{[t,r-1]}(f, n, x) \rrbracket, \llbracket a \rrbracket) & r > 1 \end{cases} \\
\llbracket \text{zipWith}_{[t_1,t_2,t_3,r]}(f, a_1, a_2) \rrbracket & = & \\
\end{array}
$$

$$
\begin{cases} \text{map}(\llbracket f \rrbracket_{fn}^{\llbracket t_3 \rrbracket}\text{,zip}(\llbracket a_1 \rrbracket, \llbracket a_2 \rrbracket)) & r = 1 \\ \text{map(fn } t_3^{r-1} \ (t_1^{r-1} \ x, t_2^{r-1} \ y) \texttt{ => } \llbracket \text{zipWith}_{[t_1,t_2,t_3 r-1]}(f, x, y) \rrbracket\text{,zip}(\llbracket a_1 \rrbracket, \llbracket a_2 \rrbracket)) & r > 1 \end{cases}
$$

$$
\begin{array}{lcl}
\llbracket \text{vrotate}_{[t,r]}(i, a) \rrbracket & = & \text{map(fn x} \texttt{ => } \text{a[x + i \% size(0,a)], iota(size(0,a))} \quad \text{, x is fresh} \\
\llbracket \text{vreverse}_{[t,r]}(a) \rrbracket & = & \text{map(fn x} \texttt{ => } \text{a[size(0,a)}-\text{x}-\text{1], iota(size}(0, a)) \quad \text{, x is fresh} \\
\llbracket \text{reverse}_{[t,r]}(a) \rrbracket & = & \text{rearrange}((r-1, \ldots, 0), \llbracket \text{vreverse}_{[t,r]}(\text{transp}_{[t,r]}(a)) \rrbracket) \\
\llbracket \text{rotate}_{[t,r]}(i, a) \rrbracket & = & \text{rearrange}((r-1, \ldots, 0), \llbracket \text{vrotate}_{[t,r]}(i, \text{transp}_{[t,r]}(a)) \rrbracket) \\
\llbracket \text{reshape}_{[t,r_1,r_2]}(a_1, a_2) \rrbracket & = & \text{reshape}(\llbracket a_1 \rrbracket, (\text{reshape1}_{\llbracket t \rrbracket}(\text{osize}, \text{reshape}(\text{isize}, \llbracket a_2 \rrbracket)))) \\
 & & \qquad \text{where osize} = \text{size}(0, a_1) * \ldots * \text{size}(r_1, a_1) \\
 & & \qquad\qquad\quad \text{isize} = \text{size}(0, a_2) * \ldots * \text{size}(r_2, a_2) \\
\llbracket \text{cat}_{[t,r]}(a_1, a_2) \rrbracket & = & \begin{cases} \text{concat}(\llbracket a_1 \rrbracket, \llbracket a_2 \rrbracket) & r = 1 \\ \text{map (fn } \llbracket t \rrbracket^{r-1} \ (\llbracket t \rrbracket \ x, \llbracket t \rrbracket \ y) \texttt{ => } \llbracket \text{cat}_{[t,r-1]}(x, y) \rrbracket, \text{zip}(\llbracket a_1 \rrbracket, \llbracket a_2 \rrbracket) & r > 1 \end{cases} \\
\llbracket \text{first}_{[t,r]}(a) \rrbracket & = & \text{let x} = \llbracket a \rrbracket \text{ in } x[\underbrace{0, ..., 0}_{\text{r times}}] \\
\llbracket \text{firstV}_{[t,r]}(a) \rrbracket & = & \llbracket \text{first}_{[t,1]}(a) \rrbracket \\
\llbracket \text{take}_{[t,r]}(i, a) \rrbracket & = & \text{reshape(oshape,take1}_{\llbracket t \rrbracket}\text{(osize,reshape(isize,}\llbracket a \rrbracket))) \\
 & & \qquad \text{where oshape} = (|i|, \text{size}(1, \llbracket a \rrbracket), \cdots \text{,size(r,}\llbracket a \rrbracket)) \\
 & & \qquad\qquad\quad \text{osize} = (i * \text{size}(1, \llbracket a \rrbracket) * \ldots * \text{size(r,}\llbracket a \rrbracket)) \\
 & & \qquad\qquad\quad \text{isize} = \text{size(0,}\llbracket a \rrbracket) * \ldots * \text{size(r,}\llbracket a \rrbracket) \\
\llbracket \text{takeV}_{[t]}(d, a) \rrbracket & = & \text{take1}_{\llbracket t \rrbracket}(\llbracket d \rrbracket, \llbracket a \rrbracket) \\
\llbracket \text{drop}_{[t,r]}(i, a) \rrbracket & = & \text{reshape(oshape, drop1}_{\llbracket t \rrbracket}(\text{osize}, \text{reshape}(\text{isize}, \llbracket a \rrbracket)) \\
 & & \qquad \text{where oshape} = (\text{max(0,size(0,}\llbracket a \rrbracket\text{)-}|i|\text{),size(1,}\llbracket a \rrbracket), \ldots \text{,size(r,}\llbracket a \rrbracket)) \\
 & & \qquad\qquad\quad \text{osize} = (i * \text{size(1,}\llbracket a \rrbracket) * \ldots * \text{size(r, }\llbracket a \rrbracket)) \\
 & & \qquad\qquad\quad \text{isize} = \text{size(0,}\llbracket a \rrbracket) * \ldots * \text{size(r,}\llbracket a \rrbracket) \\
\llbracket \text{dropV}_{[t]}(d, a) \rrbracket & = & \text{drop1}_{\llbracket t \rrbracket}(\llbracket d \rrbracket, \llbracket a \rrbracket) \\
\llbracket \text{transp}_{[t,r]}(a) \rrbracket & = & \text{rearrange}((r-1, \cdots, 0), \llbracket a \rrbracket) \\
\llbracket \text{transp2}_{[t,r]}([a_1, \cdots, a_n], b) \rrbracket & = & \text{rearrange}((a_1 - 1, \cdots, a_n - 1), \llbracket a_2 \rrbracket), \ a_1, \ldots, a_n \text{ literals} \\
\llbracket \text{cons}_{[t,r]}(e, a) \rrbracket & = & \text{rearrange}((r, \ldots, 0), \text{concat}(\llbracket \text{transp}_{[t,r+1]}(e) \rrbracket, \llbracket \text{transp}_{[t,r+1]}(a) \rrbracket)) \\
\llbracket \text{snoc}_{[t,r]}(a, e) \rrbracket & = & \text{rearrange}((r, \ldots, 0), \text{concat}(\llbracket \text{transp}_{[t,r+1]}(a) \rrbracket, \llbracket \text{transp}_{[t,r+1]}(e) \rrbracket)) \\
\llbracket \text{iota}(a) \rrbracket & = & \text{map}(+\ (1), \text{iota}(\llbracket a \rrbracket)) \\
\llbracket \text{iotaV}(a) \rrbracket & = & \llbracket \text{iota}(a) \rrbracket \\
\llbracket \text{shape}_{[t,r]}(a) \rrbracket & = & [\text{size}(0, \llbracket a \rrbracket), ..., \text{size}(r-1, \llbracket a \rrbracket)] \\
\llbracket \text{shapeV}_{[t,r]}(a) \rrbracket & = & [r]
\end{array}
$$

Figure 2: Conversion rules for expressions.

$$
\begin{array}{lcl}
[\![\text{fn } x : t \texttt{ => } e]\!]^{\tau}_{fn} & = & \text{fn } \tau([\![t]\!]\ x) \texttt{ => } [\![e]\!] \\
[\![\text{fn } x : t_1 \texttt{ => } \text{fn } y : t_2 \texttt{ => } e]\!]^{\tau}_{fn} & = & \text{fn } \tau([\![t_1]\!]\ x, [\![t_2]\!]\ y) \texttt{ => } [\![e]\!] \\
[\![\text{op}]\!]^{\tau}_{fn} & = & \begin{cases} [\![\text{op}]\!]_{fun} & op \in funs \\ [\![\text{op}]\!]_{op} & op \in binops \end{cases}
\end{array}
$$

Figure 3: Conversion rules for lambda expressions.

$$
\begin{array}{lcl}
[\![i2d]\!]_{fun} & = & \text{toReal} \\
[\![catV]\!]_{fun} & = & \text{concat} \\
[\![b2i]\!]_{fun} & = & \text{boolToInt} \\
[\![b2iV]\!]_{fun} & = & \text{boolToInt} \\
[\![ln]\!]_{fun} & = & \text{log} \\
[\![expd]\!]_{fun} & = & \text{exp} \\
[\![notb]\!]_{fun} & = & !
\end{array}
$$

idFuns = negi, negd, absi, absd, mini, mind, signd, signi, maxi, maxd, eqb, xorb, nandb, norb, neqi, neqd, resi.

Figure 4: Conversion rules for functions names and functions with a 1:1 correspondence.

$$
\begin{array}{lcl}
[\![\text{addi}]\!]_{op} & = & + \\
[\![\text{addd}]\!]_{op} & = & + \\
[\![\text{subi}]\!]_{op} & = & - \\
[\![\text{subd}]\!]_{op} & = & - \\
[\![\text{multi}]\!]_{op} & = & * \\
[\![\text{multd}]\!]_{op} & = & * \\
[\![\text{ltei}]\!]_{op} & = & \leq \\
[\![\text{lted}]\!]_{op} & = & \leq \\
[\![\text{eqi}]\!]_{op} & = & == \\
[\![\text{eqd}]\!]_{op} & = & == \\
[\![\text{gti}]\!]_{op} & = & > \\
[\![\text{gtd}]\!]_{op} & = & > \\
[\![\text{gtei}]\!]_{op} & = & \geq \\
[\![\text{gted}]\!]_{op} & = & \geq \\
[\![\text{andb}]\!]_{op} & = & \&\& \\
[\![\text{orb}]\!]_{op} & = & || \\
[\![\text{divi}]\!]_{op} & = & / \\
[\![\text{divd}]\!]_{op} & = & / \\
[\![\text{powi}]\!]_{op} & = & pow \\
[\![\text{powd}]\!]_{op} & = & pow \\
[\![\text{lti}]\!]_{op} & = & < \\
[\![\text{ltd}]\!]_{op} & = & < \\
[\![\text{andi}]\!]_{op} & = & \& \\
[\![\text{andd}]\!]_{op} & = & \& \\
[\![\text{ori}]\!]_{op} & = & | \\
[\![\text{shli}]\!]_{op} & = & << \\
[\![\text{shri}]\!]_{op} & = & >>
\end{array}
$$

Figure 5: Conversion rules for binary operators.

Apart from the $[\![\cdot]\!]$ rule, there are also the $[\![\cdot]\!]_{op}$, $[\![\cdot]\!]_{fun}$, and $[\![\cdot]\!]^{\tau}_{fn}$ rules. The first two are simple lookup rules that map to Futhark operators and functions respectively. The third is used to compile lambda expressions. Unlike the other rules the fn rule is parametrized by a Futhark type variable τ which we denote with a superscript so the rule will usually be written $[\![\cdot]\!]^{\tau}_{fn}$. The parameter τ represents the return type of the lambda expression the rule compiles, and must be passed by the caller when the rule is used.

The set *binops* is the defined as the set of operators that have an *op* rule, similarly the set $funs$ is the set of operators that have a fun rule.

6.2 Explanation of the compilation of selected parts of TAIL

Below is the motivation and explanation for the nontrivial conversion rules from the compilation scheme.

6.2.1 Basic structural constructs

Basic structural constructs are translated to their Futhark counterparts directly. In let-expressions the type annotations that exist in TAIL variable bindings are ignored in Futhark [6] [7].

The letters x, i, d, and c denote variables, integers, doubles, booleans, and chars respectively. They are all translated to their Futhark equivalents.

This part of the language was easy to compile.

6.2.2 The each operator

In TAIL, applying the `each` operator produces an array where the argument function is applied to each basic element in the argument array, regardless of the rank of the array [6]. Since Futhark views a multidimensional array as nested simple arrays, it applies the function to every array in the array. That is, it maps the function into the outer-most dimension of the array [7].

To solve this problem we introduce nested `maps` to the depth of the array with the required function. For example, an `each` operation over an array of rank 2 would have two `maps` nested in each other so that the function is mapped on each element of the basic type.

For example an each operation on an array of rank 2 will look like:

```
each(f,a)        =>       map(fn x => map (f,x), a)
```

This rule targets the Futhark `map` SOAC as directly as possible.

6.2.3 The reduce operator

The `reduce` operator in TAIL uses an associative binary operator to reduce an array of rank $\gamma + 1$ to an array of rank γ by reducing along the inner-most dimension [6]. The Futhark `reduce`, on the other hand, reduces each array in the outer array, (i.e. it reduces along the outer-most dimension [7]).

We have adopted the same approach as with each by using nested maps to map the reduce on the innermost dimension.

For example reducing an array of rank 2 emits the following code:

```
reduce(+,a)      =>       map(fn x => reduce(+,x), a)
```

Lifting the reduce operation with maps into the inner-most array was the simplest solution. It utilizes only parallel operations.

6.2.4 The zipWith operator

The `zipWith` operator applies a scalar binary operator on pairs of elements from two arrays of the same shape to produce a third array of the same shape as the input arrays [6].

To do this in Futhark, we use the zip function to convert two arrays to an array of tuples and map the binary operator on that array of tuples [7].

The rationale behind this rule was the same as in `each`.

The compilation of the three parallel higher-order operators (`each`, `reduce`, `zipWith`) has the same recursive structure. Because of the recursive structure some information can be said to have been lost in the compilation, for example a single `each` operation might have been compiled to a set of nested map operations, which seems harder to compile to lower-level parallel code, since the compiler must inspect the nested maps to discover that the expression is completely parallel. We rely here on the flattening analysis of the Futhark compiler to rediscover this information and we believe it will be able to do so.

6.2.5 The reshape operator

Futhark has a reshape function that only works for arrays of the correct dimensions [7].

To actually change the rank of the array we first ensure that the array is the correct size and then use the Futhark reshape function to do the final step.

To adjust the size we operate on the flat representation of the array, which is easy to produce using Futhark reshape.

To adjust the size of the array we use the previously defined library function `reshape1`. Actually we use the variant with the correct type, this type is conveniently available in the instance list.

We make use of the existing reshape operation in Futhark because we assume this approach has the best chance of optimization by the Futhark compiler [7] [9], [8].

6.2.6 The transp operator

There exists a `transpose(a)` function in Futhark which does not have the same semantics as the `transp` operator from TAIL [7] [6]. The Futhark transpose on a three dimensional array, for example, produces a (2,0,1) permutation of the dimensions whereas we are looking for a (2,1,0) or more generally the reverse permutation of the dimensions. By inspecting the Futhark IL (internal language) generated from a call to transpose, we discovered that, internally, a function called `rearrange` is being called with an explicit permutation parameter. This function is also available in the external language and simply needed to be called with the correct parameters to match the behavior of the `transp` operator.

This conversion is as direct as we could hope.

6.2.7 The transp2 operator

Like we did for the `transp` operator, we have also converted the operator `transp2` to a `rearrange` application. The only thing we needed to change was to substract one from each number in the first argument since Futhark indexes dimensions from zero [7]. Notice that `rearrange` only supports a list of integer literals in its first argument while TAIL has no such restriction on `transp2`. In practice the TAIL compiler will often have inlined the arguments to `transp2` [6].

The operator `transp2` has thus a very direct conversion.

6.2.8 The cat operator

Futhark has a concatenate function `concat` that we wanted to use but it concatenate the outermost arrays while in TAIL the `cat` operator concatenates the innermost arrays [7] [6]. To solve this we lifted the concatenate operation to the innermost dimension with map. This is the same idea used to compile the `reduce`, `each`, and `zipWith` operators.

Alternatively we could have compiled the `cat` operator using transpose instead like we have done in `snoc` and `cons`. We did not have any particular reason to choose one over the other. Both solutions accomplish our goal of being simple and using Futhark built-in functions.

6.2.9 The take/drop operators

In a similar fashion to the TAIL `reshape` function, we have used library functions to do most of the work. We flatten the array, let the library function work on the flat representation and finally reshape it to the desired shape. This approach has all the benefits mentioned in the `cat` operator section.

We use the same approach to implement the `drop` operator as the take operator.

6.2.10 The cons/snoc operator

The idea behind the compilation of the `cons` operator was to transpose the two arrays, then concatenate the arrays and then transpose the resulting array back again. That way we would get the desired result of the nth elements from the first array added to the nth element of the second array.

The `snoc` operator is compiled similarly the same except the elements are added behind instead in front.

6.2.11 The iota operator

Due to the fact that the 1-indexing is used in TAIL and 0-indexing is used in Futhark, the curried +1 is mapped onto the elements of the array created by using the `iota` function of Futhark.

6.2.12 Lambda expressions

The higher order operators `each, eachV, reduce, and zipWith` all take functions as arguments and these functions are handled by the conversion rules marked with the fn subscript. In Futhark, lambda expressions need type annotations both for the argument and the return type [7]. This return type is provided by the context in which the lambda is used. Namely the type informations is present in the instance lists of the enclosing operator call, be it `each`, `reduce` or `zipWith`. Arguments are already annotated with types in TAIL so those are simply compiled to Futhark types and passed to the resulting lambda. Althougn the syntax of TAIL permits lambda expressions anywhere a expression could be used (as long as the type is correct), when compiled from APL, lambda expressions will only be present in higher-order operator calls after the compilation, due to inlining.

This means that lambda expressions can only occur directly inside of the aforementioned operator call or another lambda.

In TAIL ,currying is used if lambda expressions are to take more than one argument while Futhark does not support currying but supports multi-argument lambdas instead [6]. Since the highest number of arguments that can be used is two (in `zipWith`) we have restricted the compiler to this special case which simplifies the compilation. The actual body of the function is compiled using the expression rule. If the function argument is an identifier, we use the *op* and fun rules to compile them. Lambda expressions with one argument are mapped to lambda expressions with one argument in Futhark.

6.2.13 Binary scalar first-order operators

The binary scalar first-order operators are mapped to their natural Futhark counterparts.

7 Implementation

In this section a Haskell implementation of the compilation scheme is presented. This compiler is divided into three parts: a parser, a compiler that transforms the TAIL abstract syntax three (AST) returned by the parser to a Futhark AST and pretty printer that given the Futhark AST prints the Futhark source code.

Because we have only implemented a subset of TAIL not all TAIL programs can be compiled.

7.1 The parser

As mentioned earlier the parser used in this project was made by someone else in another project [1]. We did not create the parser ourselves. Because of the ongoing development of the TAIL language, however, we modified the parser to work on the latest version of the language. We have forked the original repository to work on the modifications of the parser. The original parser has since been adapted to work with the new version of TAIL by its author. For the latest version of the parser see the github repository: https://github.com/mbudde/aplacc.

Below we discuss some of the changes we had to make to the parser and the abstract syntax tree representation of TAIL.

We had to extend the abstract syntax three to include booleans and chars. The original parser had type constructors ShT (shape type), SiT (singleton integer type), and ViT (single element vector type) that all just took a rank. We have changed these type constructors to VecT (vector type) that takes basic type and rank, S (singleton integers and booleans), and SV (single element vectors) that take rank. This meant updating the parser to read angles (<>) since this is the new syntax of vectors.

The parser with our modifications is located in the `aplacc/` directory of our project. The updated code for the parser can be found in Appendix A.

7.2 The compiler

The main part of the compiler is (placed in `src/Tail2Futhark/Compile.hs`) transforms the TAIL AST (abstract syntax three) that is returned by the `aplacc` parser to a Futhark AST. The definition of the Futhark AST is placed in `src/Tail2Futhark/Futhark/AST.hs`. The implementation of the compiler is very close to the conversion rules presented in the compilation scheme in Section 6. The source code of the compiler can be found in its entirety in Appendix B.

The module defined in the `Compile.hs` file exports one function:

```haskell
compile :: Options -> T.Program -> F.Program
```

The `compile` function produces a Futhark program given options and a TAIL program. Right now the only options provided is `--no-include-lib-funs` that when used makes the compiler not include library functions in the output file.

Since a TAIL program is an expression, the `compile` function calls another function called `compileExpression` on the TAIL program. The `compileExpression` function pattern matches on the TAIL AST to compile the expression. The most difficult case being an operator application. In the case of an operator application, the `compileOperator` function is called, which matches on the operator names. Notice that the structure of the compiler is very similar to the compilation scheme with rules being cases in pattern matching. The resulting Futhark expression is then made the body of the main function in the Futhark program. In our representation of the Futhark AST, a Futhark program consists of a list of function declarations that are represented by the type FunDecl.

```haskell
type Program = [FunDecl]
```

Our library functions are then represented as instances of type FunDecl in the compiler (in the `Compile.hs` file). Library functions are added to the beginning of every file (except if the `--no-include-lib-funs` option is used). Some of the functions are created from our templates. First, the function body is created as a Futhark expression. Then we make a Futhark function that is parameterized over a Futhark type. We represent such a function with the Haskell type `F.Type -> FunDecl`. We can even parametrize the bodies of the functions by giving them the Haskell type `F.Type -> F.Exp`. This type is simply passed from the parametrized function when it is instantiated. We use this for example in the body of `drop` where the empty list of the argument Futhark type has to be returned. Also the body of `take` is paramterized over the expression with which the input should be padded, so it has the Haskell type `F.Exp -> F.Exp`. To produce all the functions we simply map the parametrized versions over the basic Futhark types.

We add `t_` in front of existing variable names when we compile the expression. That way when we need to introduce fresh variable names we can use any variable name not starting with `t_` and not worry about clashing with names in the source code. This was much easier than including a monad to produce fresh names.

In TAIL there is a function called `readIntVecFile` for reading input from a file. This functionality does not exist in Futhark making it difficult to implement [6] [7]. However, we needed the functionality in order to implement some of our benchmarks so to get around this problem we used the fact that Futhark can take input in a program as arguments to the main function by reading from StdIn [7]. We therefore added a check to see if the first expressions in a TAIL program is `readIntVecFile`. If it is we compile the expressions to arguments in the Futhark main function. That means that if the TAIL program reads an input so does the Futhark program. We do not take into account that the syntax of the data the programs read are different from TAIL and Futhark. Furthermore, it is the responsibility of the programmer to ensure that if a TAIL program reads a file, that the programmer pipes the content of the file (maybe in a different format) to StdIn when Futhark programs are read.

7.3 Pretty print

The final part of the compiler is the pretty printer located at `src/ Tail2Futhark/Futhark/Pretty.hs`. The pretty printer takes a Futhark AST and transforms the abstract representation of the components of the AST to correct Futhark source code.

7.4 Test of implementation

In this sections we discuss the test suite of our implementation.

Both the APLTAIL compiler and the Futhark compiler has an interpreter option [6] [7]. To test the correctness of our implementation we compare the output of the APLTAIL interpreter with the output of the Futhark interpreter. The tests can be found in the `tests/our_tests/` directory in our project. The test framework compiles each file with a `tail` extension in the directory to a `fut` file of the same name. Then the Futhark program is executed with the Futhark interpreter and the output is written to a file with the `out` extension. Finally this file is compared to a file with the same name and the `ok` extension, if the files match the test passed. The `ok` files were produced by running the test suite programs with the APLTAIL interpreter.

All tests are originally written in APL code to ensure that the programs we tested up against was indeed correct TAIL and that no error was introduced by writing TAIL code ourselves. Also, TAIL was designed as an intermediate language and this approach comes closer to real world use.

In order to run the tests use '`cabal test`' instruction in the root directory of the repository after cloning it. You need the `tail2futhark` and `futhark` executables in your $PATH to do this or the tests will fail. You can also run '`ghc tests/Test && tests/Test`' to see the test results in colored output instead for easier readability.

The above mentioned framework enabled us to run the tests whenever new functionality was added and thereby check to see that the new functionality had not introduced bugs that impacted the existing functionality. Whenever a function was implemented during the development process, test cases would be added to the framework along with the already existing ones. These tests are all run using our implementation and compared to the correct output.

We use the tasty package [4] which provides a test framework that we use to implement our tests. Furthermore we have used the package tasty-golden [3] which is a plug-in for the tasty framework that allows to test against "golden" files. A lot of projects implement their own test frameworks and we could have done the same. We chose not to do so as this freed us to use our resources on developing the compiler instead.

The test results can be found in Appendix E. Based on the results of the test, we assume our implementation works as expected on the subset of TAIL we have tested. However, because the tests are not exhaustive we cannot be sure everything works. In the next, section we discuss the limitations of the tests.

7.4.1 Limitations of the tests

In order to thoroughly test the compiler, all functions should be have specific tests that test the below features that meaningfully apply to the function. The function should:

- work on different data types (both base types and rank)
- work on edge-cases
- work on positive and negative input
- work correct in all branches in if-then-else expressions

However, due to time restraints we have chosen to focus on a smaller subset of the functionality.

We have not tested for or taken into account the possibility of TAIL code that is incorrect as it is generated by a compiler and only used as an intermediate language [6].

All tests returns the expected results. As we have not done exhaustive testing we cannot be sure that all functionality work as expected, only that the one we have tested for does.

8 Benchmarks

We used a number of benchmarks to measure the performance of the code generated by our compiler. All benchmarks can be found in the `tests/benchmarks` directory in our project. The benchmarks are written in APL and then compiled to C-code using the APLTAIL compiler to create the TAIL version and the APLTAIL compiler, our compiler, and the Futhark compiler to make the Futhark version. The path from APL code to executable file can be seen in Figure 6.

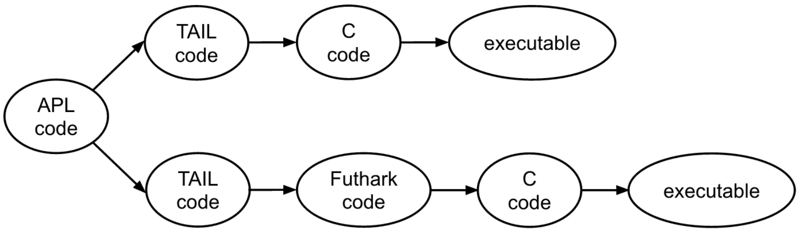

Figure 6: The path for a benchmark from APL to an executable file.

Because no parallel back-end was finished for either TAIL or Futhark, for running the benchmarks, we use a sequential back-end for both languages [6] [7]. The C-code is then compiled using `gcc` with the flags `-lm -std=c99 -O3` and the command-line tool `time` is used to measure the execution time. Each benchmark is run 10 times each and then the average is reported. The benchmarks and the results of the benchmarks are listed in Table 1.

The benchmarks is run on an Intel(R) Core(TM) i7-4500U CPU @ 1.80GHz.

Benchmark	Problem size	TAIL C	Futhark C
Matrix multiplication	512×512	2663.4	2634.2
Pi	40 000 points	8190	663.4
Black sholes	-	1	1
Easter	400	639.1	665.6
Primes	10 000	652.8	480.1

Table 1: Benchmark timing in milliseconds.

We made a Makefile to manage the building of the benchmarks. The Makefile is also placed in the `tests/benchmarks` directory.

8.1 Matrix multiplication

The matrix multiplication benchmark takes a matrix and multiplies it with itself transposed. It then reduces the resulting matrix twice, once by using times and once by using plus (times is used on the outer dimension). The implementation is in the file `matmul.apl`.

To run the Futhark version of the benchamark run '`fut_matmul < matmul.in`'

8.2 Pi

The pi benchmark approximates pi by computing the ratio of points in the range $[0,1] \times [0,1]$ that have a distance to $(0,0)$ of less than 1. It does so by using $n \times n$ evenly spaced points in the interval $[0,1] \times [0,1]$, it can be helpful to imagine the set of points as a regular grid. The more fine grained the grid the closer the approximation to pi.

The program first generates n evenly spaced points in the interval $[0,1]$. It then squares those points before replicating them n times to a $n \times n$ matrix. Then the matrix is added with its own transpose and from the resulting matrix a boolean matrix is produced where all the entries with points with distance less than one from 0 are set to 1 and the rest to 0. Is is not necessary to take the square root of the sums since that square root will be less that 1 only if the original sum is less that 1. Finally the matrix is reduced with plus two times to get the number of points. The amount is divided by the total number of points and this number is multiplied by 4 to get pi. The implementation is in the file `pi.apl`

8.3 Black-Scholes

The Black-Scholes benchmark is taken directly from the benchmark suite of the APLTAIL compiler repository and computes the price of European style options. We have not modified this benchmark and while it doesn't give any insight into performance it demonstrates that it is possible to compile this benchmark. The implementation is in the file `blackscholes.apl`

8.4 Easter

The easter benchmark computes the date of easter and is found in the apltail project as `tests/easter3000.apl`. The only modification we have done is to make the date the result from the program instead of printing it and changed a parameter to scale the program up.

The implementation is in the file `easter3000.apl`

8.5 Primes

The primes benchmark computes the number of primes below n and is found in the apltail project as `tests/primes0.apl`. Again we have only made the program return the result instead of printing it and scaled up the parameter n. To show a bigger example we have chosen to show the code in the primes benchmark. Below is the original code in APL as well as the generated TAIL and Futhark code. The TAIL code is as follows:

```
A←1↓⍳9                  ⍝   A stores the array: 2 3 4 5 6 7 8 9
residual ← A∘.|A        ⍝   residual stores all remainders af all numbers in
                        ⍝   A by all numbers in A
b ← 0=residual          ⍝   b is a boolean matrix where all entries with
                        ⍝   zero vauled remainders are asserted
c ← +⌿ b                ⍝   c counts the number of 0 valued remainders in
                        ⍝   each column
d ← 1=c                 ⍝   d is the boolean vector of where indicies for
                        ⍝   columns that have one zero valued remainder are asserted
e ← +/ d                ⍝   e counts the number of prime numbers less than 10
```

The APLTAIL compiler compiles the code to the following TAIL program:

```
let v1:<int>8 = dropV{[int],[8]}(1,iotaV(9)) in
let v7:[int]2 = transp{[int],[2]}(reshape{[int],[1,2]}([8,8],v1)) in
let v8:[int]2 = reshape{[int],[1,2]}([8,8],v1) in
let v11:[int]2 = zipWith{[int,int,int],[2]}(resi,v7,v8) in
let v18:[int]1 = transp{[int],[1]}(reduce{[int],[1]}(addi,0,each{[bool,int],[2]}(b2i,transp{[bool],[2]}(v13)))) in
let v13:[bool]2 = each{[int,bool],[2]}(fn v12:[int]0 => eqi(0,v12),v11) in
let v20:[bool]1 = each{[int,bool],[1]}(fn v19:[int]0 => eqi(1,v19),v18) in
let v24:[int]0 = reduce{[int],[0]}(addi,0,each{[bool,int],[1]}(b2i,v20)) in
i2d(v24)
```

The Futhark code is as follows:

```
fun real main() =
  let t_v1 = drop1_int(1,map(fn int (int x) => (x + 1),iota(9))) in
  let t_v7 = rearrange((1,0),reshape((8,8),reshape1_int((8 * (8 * 1)),reshape(((size(0,t_v1) * 1)),t_v1)))) in
  let t_v8 = reshape((8,8),reshape1_int((8 * (8 * 1)),reshape(((size(0,t_v1) * 1)),t_v1))) in
  let t_v11 = map(fn [int] ([int] x,[int] y) => map(resi,zip(x,y)),zip(t_v7,t_v8)) in
  let t_v13 = map(fn [bool] ([int] x) => map(fn bool (int t_v12) => (0 == t_v12),x),t_v11) in
  let t_v18 = rearrange((0),map(fn int ([int] x) => reduce(+,0,x),map(fn [int] ([bool] x) => map(boolToInt,x),rearrange((1,0),t_v13)))) in
  let t_v20 = map(fn bool (int t_v19) => (1 == t_v19),t_v18) in
  let t_v24 = reduce(+,0,map(boolToInt,t_v20)) in
  toFloat(t_v24)
```

Notice that the definition of the library functions are omitted from this example to save space. They are included above the main function in the file containing the code. The entire Futhark file can be found in Appendix D.

This example illustrates that the parallel operators of TAIL (`zipWith`, `each` and `reduce`) are compiled to parallel second order functions in Futhark (`map` and `reduce`). We also see that library functions are used (`drop1_int` and `reshape1_int`) on the one-dimensional case.

The implementation can also be seen in the file `primes0.apl`.

8.6 Results

As mentioned previously the results can be seen Table 1. In two benhamarks TAIL and Futhark perform almost the same, in one Futhark is somewhat faster than TAIL and in one we get a significant speed-up from Futhark. In the pi benchmark Futhark performs much better than TAIL. This is due to the fact that the APLTAIL compiler ends up fusing too much and duplicates work. It calculates $x * x$ n^2 times instead of just n times as expressed in the APL program. This is a limitation the authors are aware of and describe in their publication [2].

9 Discussion

In this section we discuss the viability of the approach we have used, what we have learned in the project and ideas for future work.

Our main goal has been to see if it is possible, effectively to compile TAIL programs, into Futhark programs and thereby make use of the Futhark infrastructure for optimization and the possibility for targeting parallel hardware.

We have been able to compile TAIL to Futhark code using parallel constructs in Futhark whenever we encountered parallel constructs in TAIL. Because of this we have reason to believe that no parallelism has been eliminated during the compilation. Although we have not verified the effect the compilation has on performance on parallel hardware we have seen that compiling to Futhark can in some cases speed up the sequential execution of the code. We feel these sequentially executed benchmarks are relevant because they are parallel in their structure and should therefore execute efficiently on parallel hardware.

During this project we have gotten a greater understanding of data-parallelism and some of the things that can influence the efficiency of the parallel code, such as memory constrains in the case of general computing on the GPUs.

During this project we have learned the value of using a mathematical notation to work from and to present and reason about in the form of our compilation scheme. Without our compilation scheme it is not clear how to present the work, either we would have to argue based on the implementation which muddles the picture with implementation details, or we would have to argue in prose which makes it difficult to precisely explain the concepts without being very verbose. The notation is also a good tool for communicating during the development of the ideas for compilation, because it is programming language independent.

Also we have learned that the type systems means a lot when compiling between languages. In our concrete example we had an issue with polymorphism. This made the compilation a lot less simple since we had to make design decisions on how to handle this in the best way and what the best way was. We decided to create library functions for each basic type and create our compilation so that we only used library functions in the one-dimensional case. Otherwise we could have made a function for each basic type and array shape for a limited set of combinations. But this means we could not have supported a significant portion of TAIL.

If we had to redo this project we would have focused more on the benchmarks from the beginning as soon as we started implementing. This makes for a more goal oriented work-flow instead of the check-list like work-flow we had. Maybe we would have implemented fewer operators and instead focused on analyzing the code generated by the compilers (APLTAIL and Futhark) and from that argue about the soundness of our conversion rules.

As we began our project no parallel back-ends for either languages were available, however towards the end of the project a parallel back-end for TAIL using Accelerate had been published [2] and a parallel back-end for Futhark using OpenCL was in development.

It would be interesting to test the benchmarks with the parallel back-ends to compare TAIL and Futhark on parallel hardware. Depending on the results it could be interesting to look at the generated C code to see the cause of the differences in the runtime of the benchmarks.

Finally, it could also be interesting to try out bigger and more comprehensive benchmarks and compare the running times.

10 Conclusion

In this report we have described relevant part of the two languages TAIL and Futhark. We have also presented a compilation between the two languages shown in an implementation independent mathematical notation as well as an implementation of this scheme in Haskell and test of this implementation. Finally, we have compared the execution time of selected benchmarks.

In this project we wanted to examine if it was possible, effectively to compile TAIL programs, into Futhark programs and thereby make use of the Futhark infrastructure for optimization and the possibility for targeting parallel hardware.

We have shown that it is possible to effectively compile TAIL to Futhark by expressing this compilation in a compilation scheme done in a mathematical notation that is language independent and also implement this compilation scheme in Haskell. We have used the Haskell implementation to test the correctness of the compilation scheme.

We have argued that the parallelism in the code is preserved by ensuring that all parallel operators in TAIL are mapped to parallel functions in Futhark. This parallelism in the resulting code means that the Futhark infrastructure for optimization can be used to optimize the code as well as the create a possibility for targeting parallel hardware.

We have shown that compiling the code with the Futhark compiler has the benefit of optimizing the code as we have measured speed-ups from utilizing the Futhark compiler in our benchmarks.

References

[1] Michael Budde. Compiling APL to accelerate through a typed IL. Msc project, Department of Computer Science, University of Copenhagen, November 2014.

[2] Michael Budde, Martin Dybdal, and Martin Elsman. Compiling APL to accelerate through a typed array intermediate language. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY'15. ACM, June 2015.

[3] Roman Cheplyaka. The tasty-golden package. https://hackage.haskell.org/package/tasty-golden. [Online; accessed 19-April-2015].

[4] Roman Cheplyaka. The tasty package. https://hackage.haskell.org/package/tasty. [Online; accessed 19-April-2015].

[5] Martin Elsman. Optimising Typed Programs. http://www.elsman.com/pdf/optimiser.pdf, 1998. [Online; accessed 19-April-2015].

[6] 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, June 2014.

[7] Troels Henriksen. Exploiting functional invariants to optimise parallelism: a dataflow approach. Msc thesis, Department of Computer Science, University of Copenhagen, February 2014.

[8] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. Size slicing - a hybrid approach to size inference in futhark. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional High-Performance Computing*. ACM, September 2014.

[9] Troels Henriksen and Cosmin E. Oancea. A t2 graph-reduction approach to fusion. In *2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*. ACM, September 2013.

[10] Troels Henriksen and Cosmin E. Oancea. Bounds checking: An instance of hybrid analysis. In *ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY'14. ACM, September 2014.

[11] Kenneth E. Iverson. *A programming Language*. John Wiley and Sons, 1962.

[12] Isaac Jones and Duncan Coutts. The Cabal package. https://hackage.haskell.org/package/Cabal. [Online; accessed 19-April-2015].

[13] Bernard Legrand. *Mastering Dyalog APL: A Complete Introduction to Dyalog APL*. Dyalog Limited, 2009.

[14] Torben Mogensen. *Introduction to Compiler Design*. Springer, 2011.

A Parser source code

In this appendix is the source code for the APLACC parser [1] with the modifications we added. We did not develop the code presented in this appendix but only made small alterations to the existing code. A description of the alterations we did add can be found in Section 7.1 and can also be seen in the commits in the github repository for our project.

```
module APLAcc.TAIL.Parser (parseFile) where

import System.IO (Handle, hGetContents)
import Control.Monad (liftM, liftM2)
import Data.Char (isSpace)
import Data.Either (partitionEithers)
import Text.Parsec hiding (Empty)
import Text.Parsec.String
import Text.Parsec.Expr
import Text.Parsec.Pos
import qualified Text.Parsec.Token as Token

import APLAcc.TAIL.AST


parseFile :: Handle -> String -> IO Program
parseFile handle filename =
  do str <- hGetContents handle
      case parse program filename str of
        Left e  -> error $ show e
        Right r -> return r


tailDef = Token.LanguageDef {
                Token.commentStart     = "(*"
              , Token.commentEnd       = "*)"
              , Token.commentLine      = ""
              , Token.nestedComments   = False
              , Token.identStart       = letter
              , Token.identLetter      = alphaNum <|> char '_'
              , Token.opStart          = oneOf ""
              , Token.opLetter         = oneOf ""
              , Token.reservedOpNames  = []
              , Token.reservedNames    = [ "let", "in", "int", "
                 double", "fn", "inf" , "tt", "ff"]
              , Token.caseSensitive    = True
  }

lexer = Token.makeTokenParser tailDef

identifier = Token.identifier lexer
reserved   = Token.reserved   lexer
reservedOp = Token.reservedOp lexer
stringlit  = Token.stringLiteral lexer
charlit    = Token.charLiteral lexer
parens     = Token.parens     lexer
brackets   = Token.brackets   lexer
angles     = Token.angles     lexer
braces     = Token.braces     lexer
integer    = Token.integer    lexer
semi       = Token.semi       lexer
comma      = Token.comma      lexer
colon      = Token.colon      lexer
symbol     = Token.symbol     lexer
whitespace = Token.whiteSpace lexer
decimal    = Token.decimal    lexer
```

```haskell
float      = Token.float      lexer
lexeme     = Token.lexeme     lexer

withPrefix :: Parser a -> Parser b -> (a -> b -> b) -> Parser b
withPrefix pre p f =
  do x <- optionMaybe pre
     y <- p
     return $ case x of
       Just x' -> f x' y
       Nothing -> y

program :: Parser Program
program =
  do whitespace
     prog <- expr
     eof
     return prog

-----------------
-- Expression

expr :: Parser Exp
expr = opExpr
   <|> arrayExpr
   <|> letExpr
   <|> fnExpr
   <|> valueExpr
   <?> "expression"

valueExpr :: Parser Exp
valueExpr = try (liftM D $ lexeme float)
         <|> liftM I (lexeme decimal)
         <|> try (reserved "inf" >> return Inf)
         <|> (char '-' >> liftM Neg valueExpr)
         <|> liftM C charlit
         <|> liftM B ((reserved "tt" >> return True) <|> (
             reserved "ff" >> return False))
         <|> liftM Var identifier
         <?> "number or identifier"

arrayExpr :: Parser Exp
arrayExpr = liftM Vc $ brackets (sepBy (opExpr <|> valueExpr)
    comma)

letExpr :: Parser Exp
letExpr =
  do reserved "let"
     (ident, typ) <- typedIdent
     symbol "="
     e1 <- expr
     reserved "in"
     e2 <- expr
     return $ Let ident typ e1 e2

instanceDecl :: Parser InstDecl
instanceDecl = braces $
  do btyps <- brackets $ sepBy basicType comma
     comma
     ranks <- brackets $ sepBy (lexeme decimal) comma
     return (btyps, ranks)

opExpr :: Parser Exp
opExpr =
```

```
  do ident <- try $ do { i <- identifier; lookAhead $ oneOf "({"; return i }
     instDecl <- optionMaybe instanceDecl
     args <- parens $ sepBy expr comma
     return $ Op ident instDecl args

fnExpr :: Parser Exp
fnExpr =
  do reserved "fn"
     (ident, typ) <- typedIdent
     symbol "=>"
     e <- expr
     return $ Fn ident typ e

typedIdent :: Parser (Ident, Type)
typedIdent =
  do ident <- identifier
     colon
     typ <- typeExpr
     return (ident, typ)

------------------
-- Types

typeExpr :: Parser Type
typeExpr = arrayType <|> vectorType <?> "type"
--typeExpr = liftM (foldr1 FunT) $
--  sepBy1 (arrayType <|> vectorType <?> "type") (symbol "->")

arrayType :: Parser Type
arrayType = liftM2 ArrT (brackets basicType) rank

-- vectortype as replacement for shapeType
vectorType :: Parser Type
vectorType = liftM2 VecT (angles basicType) rank
         <|> (try (symbol "SV") >> parens (do {t <- basicType ; comma ; r <- rank ; return $ SV t r}))
         <|> (try (symbol "S") >> parens (do {t <- basicType ; comma ; r <- rank ; return $ S t r }))
         <?> "vector type"

--shapeType :: Parser Type
--shapeType = shape "Sh" ShT
--        <|> shape "Si" SiT
--        <|> shape "Vi" ViT
--        <?> "shape type"
--  where shape name con = try (symbol name) >> liftM con (parens rank)

rank :: Parser Rank
rank = liftM R (lexeme decimal)
   -- <|> (liftM Rv identifier)  Unsupported
   <?> "rank"

basicType :: Parser BType
basicType = (reserved "int" >> return IntT)
        <|> (reserved "double" >> return DoubleT)
        <|> (reserved "bool" >> return BoolT)
        <|> (reserved "char" >> return CharT)
        <|> (char ’\’’ >> many1 alphaNum  >>= return . Btyv)
        <?> "basic type"

------------------
```

```
-- Debug functions

parseString :: Parser a -> String -> a
parseString parser str =
  case parse parser "" str of
    Left e  -> error $ show e
    Right r -> r
```

B Compiler source code

This appendix contains the source code of the TAIL2Futhark compiler found in file `src/Tail2Futhark/Compile.hs`.

```haskell
module Tail2Futhark.Compile (compile) where

import APLAcc.TAIL.AST as T -- the TAIL AST
import Tail2Futhark.Futhark.AST as F -- the futhark AST
import Data.List
import Data.Maybe
import Data.Char
import Options (Options(..))

--------------------------
-- THE MAIN FUNCTION --
--------------------------

compile :: Options -> T.Program -> F.Program
compile opts e = includes ++ [(RealT, "main", signature, compileExp rootExp)]
  where includes = (if includeLibs opts then builtins else [])
        (signature, rootExp) = compileReads e

-------------------------
-- HELPER FUNCTIONS --
-------------------------

compileReads (T.Let id  _ (T.Op "readIntVecFile" _ _) e2) = ((F.ArrayT F.IntT , "t_" ++ id):sig,e')
  where (sig,e') = compileReads e2
compileReads e = ([],e)

----------------------------------------
-- AUX FUNCTIONS OF LIBRARY FUNCTIONS --
----------------------------------------

absFloatExp :: F.Exp -> F.Exp
absFloatExp e = IfThenElse Inline (BinApp LessEq e (Constant (Real 0))) (F.Neg e) e

absExp :: F.Exp -> F.Exp
absExp e = IfThenElse Inline (BinApp LessEq e (Constant (Int 0))) (F.Neg e) e

maxExp :: F.Exp -> F.Exp -> F.Exp
maxExp e1 e2 = IfThenElse Inline (BinApp LessEq e1 e2) e2 e1

minExp e1 e2 = IfThenElse Inline (BinApp LessEq e1 e2) e1 e2

signdExp e = IfThenElse Indent (BinApp Less (Constant (Real 0)) e) (Constant (Int 1)) elseBranch
  where elseBranch = IfThenElse Indent (BinApp Eq (Constant (Real 0)) e) (Constant (Int 0)) (Constant (Int (-1)))

signiExp e = IfThenElse Indent (BinApp Less (Constant (Int 0)) e) (Constant (Int 1)) elseBranch
  where elseBranch = IfThenElse Indent (BinApp Eq (Constant (Int 0)) e) (Constant (Int 0)) (Constant (Int (-1)))

nandExp e1 e2 = F.FunCall "!" [BinApp F.LogicAnd e1 e2]

norExp e1 e2 = F.FunCall "!" [BinApp F.LogicOr e1 e2]

```

```
resiExp :: F.Exp -> F.Exp -> F.Exp
resiExp y x = F.IfThenElse F.Indent (y ‘eq‘ zero) x $ F.IfThenElse F.Indent cond (x % y) (x % y ‘plus‘ y)
  where cond = ((x % y) ‘eq‘ zero) ‘or‘ ((x ‘gr‘ zero) ‘and‘ (y ‘gr‘ zero)) ‘or‘ ((x ‘less‘ zero) ‘and‘ (y ‘less‘ zero))
        infix 1 %; (%) = F.BinApp F.Mod
        zero = Constant (Int 0)
        plus = F.BinApp F.Plus
        gr = F.BinApp F.Greater
        less = F.BinApp F.Less
        eq = F.BinApp F.Eq
        or = F.BinApp F.LogicOr
        and = F.BinApp F.LogicAnd

-- reshape1 --
-- create split part of reshape1 function --
mkSplit id1 id2 dims exp retExp = F.Let Inline (TouplePat [(Ident id1),(Ident id2)]) (F.FunCall2 "split" [dims] exp) retExp
makeLets ((id,exp) : rest) e = F.Let Indent (Ident id) exp (makeLets rest e)
makeLets [] e = e

reshape1Body :: F.Type -> F.Exp
reshape1Body tp = makeLets (zip ["roundUp","extend"] [length, reshapeCall]) split
  where split = mkSplit "v1" "_" (F.Var "l") (F.Var "extend") (F.Var "v1")
        length = (F.Var "l" ‘fplus‘ (size ‘fminus‘ Constant (Int 1)) ‘fdiv‘ size)
        reshapeCall = F.FunCall2 "reshape" [BinApp Mult size len] (F.FunCall "replicate" [len,F.Var "x"])
        size = F.FunCall "size" [Constant (Int 0),F.Var "x"]
        len = F.Var "roundUp"
        fdiv = BinApp Div
        fplus = BinApp Plus
        fminus = BinApp Minus

-- drop --
-- make body for drop1 function --
dropBody :: F.Type -> F.Exp
dropBody tp = IfThenElse Indent (size ‘less‘ absExp len) emptArr elseBranch
    where zero = Constant (Int 0)
          less = BinApp LessEq
          len = F.Var "l"
          size = F.FunCall "size" [zero, F.Var "x"]
          sum = BinApp Plus len size
          emptArr = F.Empty tp
          elseBranch = IfThenElse Indent (len ‘less‘ zero) negDrop posDrop
          negDrop = mkSplit "v1" "_" sum (F.Var "x") (F.Var "v1")
          posDrop = mkSplit "_" "v2" len (F.Var "x") (F.Var "v2")

-- take1 --
-- make body for take1 function --
takeBody :: F.Exp -> F.Exp
takeBody padElement = IfThenElse Indent (zero ‘less‘ len) posTake negTake
    where less = BinApp LessEq
          zero = Constant (Int 0)
          sum  = BinApp Plus len size
          len  = F.Var "l"
          size = F.FunCall "size" [zero, F.Var "x"]
```

```
          padRight = F.FunCall "concat" [F.Var "x", padding]
          padLeft = F.FunCall "concat" [padding, F.Var "x"]
          padding = F.FunCall "replicate" [(BinApp Minus len size), padElement]
          posTake = IfThenElse Indent (len `less` size) (mkSplit "v1" "_" (F.Var "l") (F.Var "x") (F.Var "v1")) padRight
          negTake = IfThenElse Indent (zero `less` sum) (mkSplit "_" "v2" sum (F.Var "x") (F.Var "v2")) padLeft


------------------------------------------
-- AUX FUNCTIONS FOR SPECIFIC FUNCTIONS --
------------------------------------------

-- AUX shape --
makeShape rank args
  | [e] <- args = map (\x -> FunCall "size" [Constant (Int x), compileExp e]) [0..rank-1]
  | otherwise = error "shape takes one argument"

-- AUX transp --
makeTransp r e = makeTransp2 (map (Constant . Int) (reverse [0..r-1])) e

-- AUX transp2 --
makeTransp2 dims exp = F.FunCall2 "rearrange" dims exp

---------------------------
-- GENERAL AUX FUNCTIONS --
---------------------------

-- make string representation of Futhark type --
showTp tp  = case baseType tp of
  F.IntT -> "int"
  F.RealT -> "real"
  F.BoolT -> "bool"
  F.CharT -> "char"

-- make Futhark basic type from string representation --
readBType tp = case tp of
  "int" -> F.IntT
  "real" -> F.RealT
  "bool" -> F.BoolT
  "char" -> F.CharT

-- make Futhark type from string representation --
-- i.e., takes 2int and gives [[int]] --
getType :: [Char] -> Maybe F.Type
getType s
  | suffix `elem` ["int","real","bool","char"] = fmap (makeArrTp (readBType suffix)) $ rank
  | otherwise = Nothing
  where (prefix,suffix) = span isDigit s
        rank | [] <- prefix = Nothing
             | otherwise = Just (read prefix :: Integer)

-- make list of Futhark basic types --
btypes = map readBType ["int","real","bool","char"]

-- return zero expression of basic type --
zero :: F.Type -> F.Exp
zero F.IntT = Constant (Int 0)
```

```haskell
zero F.RealT = Constant (Real 0)
zero F.BoolT = Constant (Bool False)
zero F.CharT = Constant (Char ' ')
zero tp = error $ "take for type " ++ showTp tp ++ " not supported"

-- make Futhark function expression from ident
makeKernel ident
  | Just fun <- convertFun ident = F.Fun fun []
  | Just op  <- convertBinOp ident = F.Op op
  | otherwise = error $ "not supported operation " ++ ident

-- make Futhark basic type from Tail basic type --
makeBTp T.IntT = F.IntT
makeBTp T.DoubleT = F.RealT
makeBTp T.BoolT = F.BoolT
makeBTp T.CharT = F.CharT

-- make Futhark array type from Futhark basic type --
mkType (tp,rank) = makeArrTp (makeBTp tp) rank

-- aux for mkType --
makeArrTp :: F.Type -> Integer -> F.Type
makeArrTp btp 0 = btp
makeArrTp btp n = F.ArrayT (makeArrTp btp (n-1))

-- make curried Futhark function that have 1 as basic element and folds with times
multExp :: [F.Exp] -> F.Exp
multExp = foldr (BinApp Mult) (Constant (Int 1))

-- make Futhark kernel expression with type
compileKernel :: T.Exp -> F.Type -> Kernel
compileKernel (T.Var ident) rtp = makeKernel ident
compileKernel (T.Fn ident tp (T.Fn ident2 tp2 exp)) rtp = F.Fn rtp [(compileTp tp,"t_" ++ ident),(compileTp tp2,"t_" ++ ident2)] (compileExp exp)
compileKernel (T.Fn ident tp exp) rtp = F.Fn rtp [(compileTp tp,"t_" ++ ident)] (compileExp exp)

-- AUX for compileKernel --
compileTp (ArrT bt (R rank)) = makeArrTp (makeBTp bt) rank
compileTp (VecT bt (R rank)) = makeArrTp (makeBTp bt) 1
compileTp (SV bt (R rank)) = makeArrTp (makeBTp bt) 1
compileTp (S bt _) = makeBTp bt

-----------------------
-- LIBRARY FUNCTIONS --
-----------------------

-- list containing ompl of all library functions --
builtins :: [F.FunDecl]
builtins = [boolToInt,negi,negd,absi,absd,mini,mind,signd,signi,maxi,maxd,eqb,xorb,nandb,norb,neqi,neqd,resi]
        ++ reshapeFuns
        ++ takeFuns
        ++ dropFuns

boolToInt :: FunDecl
boolToInt = (F.IntT, "boolToInt", [(F.BoolT, "x")], F.IfThenElse Inline (F.Var "x") (Constant (Int 1)) (Constant (Int 0)))

negi :: FunDecl
```

```
negi = (F.IntT, "negi", [(F.IntT,"x")], F.Neg (F.Var "x"))

negd :: FunDecl
negd = (F.RealT, "negd", [(F.RealT,"x")], F.Neg (F.Var "x"))

absi :: FunDecl
absi = (F.IntT, "absi", [(F.IntT,"x")], absExp (F.Var "x"))

absd :: FunDecl
absd = (F.RealT, "absd", [(F.RealT,"x")], absFloatExp (F.Var "x"))

mini :: FunDecl
mini = (F.IntT, "mini", [(F.IntT, "x"), (F.IntT, "y")], minExp (F.Var "x") (F.Var "y"))
mind = (F.RealT, "mind", [(F.RealT, "x"), (F.RealT, "y")], minExp (F.Var "x") (F.Var "y"))

signd = (F.IntT, "signd", [(F.RealT, "x")], signdExp (F.Var "x"))

signi = (F.IntT, "signi", [(F.IntT, "x")], signiExp (F.Var "x"))

maxi :: FunDecl
maxi = (F.IntT, "maxi", [(F.IntT, "x"), (F.IntT, "y")], maxExp (F.Var "x") (F.Var "y"))

maxd :: FunDecl
maxd = (F.RealT, "maxd", [(F.RealT, "x"), (F.RealT, "y")], maxExp (F.Var "x") (F.Var "y"))

nandb :: FunDecl
nandb = (F.BoolT, "nandb", [(F.BoolT, "x"), (F.BoolT, "y")], nandExp (F.Var "x") (F.Var "y"))

norb :: FunDecl
norb = (F.BoolT, "norb", [(F.BoolT, "x"), (F.BoolT, "y")], norExp (F.Var "x") (F.Var "y"))

eqb = (F.BoolT, "eqb", [(F.BoolT, "x"), (F.BoolT, "y")], boolEquals (F.Var "x") (F.Var "y"))
  where boolEquals e1 e2 = BinApp F.LogicOr (norExp (F.Var "x") (F.Var "y")) (BinApp F.LogicAnd (F.Var "x") (F.Var "y"))

xorb = (F.BoolT, "xorb", [(F.BoolT, "x"), (F.BoolT, "y")], boolXor (F.Var "x") (F.Var "y"))
  where boolXor e1 e2 = BinApp F.LogicAnd (nandExp (F.Var "x")(F.Var "y")) (BinApp F.LogicOr (F.Var "x") (F.Var "y"))

neqi = (F.BoolT, "neqi", [(F.IntT, "x"), (F.IntT, "y")], notEq (F.Var "x") (F.Var "y"))

neqd = (F.BoolT, "neqd", [(F.RealT, "x"), (F.RealT, "y")], notEq (F.Var "x") (F.Var "y"))

notEq e1 e2 = FunCall "!" [BinApp F.Eq e1 e2]

resi = (F.IntT, "resi", [(F.IntT, "x"),(F.IntT, "y")], resiExp (F.Var "x") (F.Var "y"))

-- AUX: make FunDecl by combining signature and body (aux function that create function body)
makeFun :: [F.Arg] -> F.Ident -> F.Exp -> F.Type -> FunDecl
```

```haskell
makeFun args name body tp = (ArrayT tp,name ++ "_" ++ showTp tp, args,body)
stdArgs tp = [(F.IntT,"l"),(ArrayT tp, "x")]

reshapeFun :: F.Type -> FunDecl
reshapeFun tp = makeFun (stdArgs tp) "reshape1" (reshape1Body tp) tp
takeFun :: F.Type -> F.FunDecl
takeFun tp = makeFun (stdArgs tp) "take1" (takeBody (zero tp)) tp
dropFun :: F.Type -> F.FunDecl
dropFun tp = makeFun (stdArgs tp) "drop1" (dropBody tp) tp

reshapeFuns = map reshapeFun btypes
takeFuns = map takeFun btypes
dropFuns = map dropFun btypes

-----------------
-- EXPRESSIONS --
-----------------

-- general expressions --
compileExp :: T.Exp -> F.Exp
compileExp (T.Var ident) | ident == "pi" = Constant(Real 3.14159265359) | otherwise = F.Var ("t_" ++ ident)
compileExp (I int) = Constant (Int int)
compileExp (D double) = Constant (Real double)
compileExp (C char)   = Constant (Char char)
compileExp (B bool)   = Constant (Bool bool)
compileExp Inf = Constant (Real (read "Infinity"))
compileExp (T.Neg exp) = F.Neg (compileExp exp)
compileExp (T.Let id _ e1 e2) = F.Let Indent (Ident ("t_" ++ id)) (compileExp e1) (compileExp e2) -- Let
compileExp (T.Op ident instDecl args) = compileOpExp ident instDecl args
compileExp (T.Fn _ _ _) = error "Fn not supported"
compileExp (Vc exps) = Array(map compileExp exps)

-- operators --
compileOpExp :: [Char] -> Maybe ([BType], [Integer]) -> [T.Exp] -> F.Exp
compileOpExp ident instDecl args = case ident of
  "reduce" -> compileReduce instDecl args
  "eachV"  -> compileEachV instDecl args
  "each"   -> compileEach instDecl args
  "firstV" -> compileFirstV instDecl args
  "first" -> compileFirst instDecl args
  "shapeV" -> F.Array $ makeShape 1 args
  "shape"  -> compileShape instDecl args
  "reshape" -> compileReshape instDecl args
  "take" -> compileTake instDecl args
  "takeV" -> compileTakeV instDecl args
  "zipWith" -> compileZipWith instDecl args
  "cat" -> compileCat instDecl args
  "reverse" -> compileReverse instDecl args
  "reverseV" -> compileVReverseV instDecl args
  "vreverse" -> compileVReverse instDecl args
  "vreverseV" -> compileVReverseV instDecl args
  "transp" -> compileTransp instDecl args
  "transp2" -> compileTransp2 instDecl args
  "drop" -> compileDrop instDecl args
  "dropV" -> compileDropV instDecl args
  "iota" -> compileIota instDecl args
  "iotaV" -> compileIota instDecl args
```

```
  "vrotate" -> compileVRotate instDecl args
  "rotate" -> compileRotate instDecl args
  "vrotateV" -> compileVRotateV instDecl args
  "rotateV" -> compileVRotateV instDecl args
  "snoc" -> compileSnoc instDecl args
  "snocV" -> compileSnocV instDecl args
  "cons" -> compileCons instDecl args
  "consV" -> compileConsV instDecl args
  "b2iV" | [T.Var "tt"] <- args -> (Constant (Int 1)) | [T.Var "
      ff"] <- args -> (Constant (Int 0)) -- | otherwise -> error "
      only bool literals supported in b2iV"
  _
    | [e1,e2]  <- args
    , Just op  <- convertBinOp ident
    -> F.BinApp op (compileExp e1) (compileExp e2)
    | Just fun <- convertFun ident
    -> F.FunCall fun $ map compileExp args
    | ident `elem` idFuns
    -> F.FunCall ident $ map compileExp args
    | otherwise       -> error $ ident ++ " not supported"

-- snocV --
compileSnocV :: Maybe InstDecl -> [T.Exp] -> F.Exp
compileSnocV (Just([tp],[r])) [a,e] = F.FunCall "concat" [
    compileExp a, F.Array [compileExp e]]
compileSnocV Nothing _ = error "snocV needs instance declaration"
compileSnocV _ _ = error "snocV take two aguments"

-- snoc --
compileSnoc :: Maybe InstDecl -> [T.Exp] -> F.Exp
compileSnoc (Just([tp],[r])) [a,e] = makeTransp2 (map (Constant .
     Int) (reverse [0..r])) (F.FunCall "concat" [arr,exp])
  where exp = F.Array [makeTransp r (compileExp e)]
        arr = makeTransp (r+1) (compileExp a)

-- consV --
compileConsV :: Maybe InstDecl -> [T.Exp] -> F.Exp
compileConsV (Just([tp],[r])) [e,a] = F.FunCall "concat" [F.Array
     [compileExp e], compileExp a]
compileConsV Nothing _ = error "consV needs instance declaration"
compileConsV _ _ = error "consV take two aguments"

-- cons --
compileCons :: Maybe InstDecl -> [T.Exp] -> F.Exp
compileCons (Just([tp],[r])) [e,a] = makeTransp2 (map (Constant .
     Int) (reverse [0..r])) (F.FunCall "concat" [exp, arr])
  where exp = F.Array [makeTransp r (compileExp e)]
        arr = makeTransp (r+1) (compileExp a)

-- first --
compileFirst (Just(_,[r])) [a] = F.Let Inline (Ident "x") (
    compileExp a) $ F.Index (F.Var "x") (replicate rInt (F.
    Constant (F.Int 0)))
  where rInt = fromInteger r :: Int
compileFirst Nothing _ = error "first needs instance declaration"
compileFirst _ _ = error "first take one argument"

-- iota --
compileIota _ [a] = Map (F.Fn F.IntT [(F.IntT, "x")] (F.BinApp
    Plus (F.Var "x") (Constant (F.Int 1)))) (FunCall "iota" [
    compileExp a])
compileIota _ _ = error "Iota take one argument"

```

```
-- vreverse --
compileVReverse (Just([tp],[r])) [a] = makeVReverse tp r (compileExp a)
compileReverse :: Maybe InstDecl -> [T.Exp] -> F.Exp
compileReverse (Just([tp],[r])) [a] = makeTransp r $ makeVReverse tp r $ makeTransp r $ compileExp a
compileVReverseV (Just([tp],[l])) [a] = makeVReverse tp 1 (compileExp a)

makeVReverse tp r a = F.Let Inline (Ident "a") a $ Map kernelExp (FunCall "iota" [FunCall "size" [F.Constant (F.Int 0), a]])
  where
    kernelExp = F.Fn (mkType (tp,r-1)) [(F.IntT,"x")] (F.Index (F.Var "a") [F.BinApp F.Minus minusIndex one])
    sizeCall = F.FunCall "size" [zero, a]
    minusIndex = F.BinApp F.Minus sizeCall (F.Var "x")
    zero = F.Constant (F.Int 0)
    one = F.Constant (F.Int 1)
    mkType (tp,rank) = makeArrTp (makeBTp tp) rank

-- rotate --
compileVRotate (Just([tp],[r])) [i,a] = makeVRotate tp r i (compileExp a)
compileVRotate Nothing _ = error "Need instance declaration for vrotate"
compileVRotate _ _ = error "vrotate needs 2 arguments"

compileRotate (Just([tp],[r])) [i,a] = makeTransp r $ makeVRotate tp r i $ makeTransp r $ compileExp a
compileRotate Nothing _ = error "Need instance declaration for rotate"
compileRotate _ _ = error "rotate needs 2 arguments"

-- vrotateV --
compileVRotateV (Just([tp],[r])) [i,a] = makeVRotate tp 1 i (compileExp a)
compileVRotateV Nothing _ = error "Need instance declaration for vrotateV"
compileVRotateV _ _ = error "vrotateV needs 2 arguments"

-- vrotate --
makeVRotate tp r i a = F.Let Inline (Ident "a") a $ Map kernelExp (FunCall "iota" [size])
  where
    kernelExp = F.Fn (mkType (tp, r-1)) [(F.IntT, "x")] (F.Index (F.Var "a") [F.BinApp F.Mod sum size])
    sum = F.BinApp F.Plus (F.Var "x") (compileExp i)
    size = FunCall "size" [F.Constant (F.Int 0), a]

-- cat --
compileCat (Just([tp],[r])) [a1,a2] = makeCat tp r (compileExp a1) (compileExp a2)
  where
    makeCat tp 1 a1 a2 = FunCall "concat" [a1, a2]
    makeCat tp r a1 a2 = Map kernelExp (FunCall "zip" [a1, a2])
      where
        kernelExp = F.Fn (mkType (tp,r-1)) [(mkType (tp,r-1),"x"), (mkType(tp,r-1),"y")] recursiveCall
        recursiveCall = makeCat tp (r-1) (F.Var "x") (F.Var "y")
    mkType (tp,rank) = makeArrTp (makeBTp tp) rank

-- takeV --
compileTakeV :: Maybe InstDecl -> [T.Exp] -> F.Exp
```

```
compileTakeV (Just([tp],_)) [len,exp] = F.FunCall fname [compileExp len,compileExp exp]
    where fname = "take1_" ++ showTp (makeBTp tp)
compileTakeV Nothing _ = error "Need instance declaration for takeV"
compileTakeV _ _ = error "TakeV needs 2 arguments"

-- dropV --
compileDropV :: Maybe InstDecl -> [T.Exp] -> F.Exp
compileDropV (Just([tp],_)) [len,exp] = F.FunCall fname [compileExp len,compileExp exp]
    where fname = "drop1_" ++ showTp (makeBTp tp)
compileDropV Nothing _ = error "Need instance declaration for dropV"
compileDropV _ _ = error "DropV needs 2 arguments"

-- take --
compileTake :: Maybe InstDecl -> [T.Exp] -> F.Exp
compileTake (Just([tp],[r])) [len,exp] = F.FunCall2 "reshape" dims $ F.FunCall fname [sizeProd,resh]
    where dims = absExp (compileExp len) : tail shape
          sizeProd = multExp $ compileExp len : tail shape
          fname = "take1_" ++ showTp (makeBTp tp)
          resh = F.FunCall2 "reshape" [multExp shape] (compileExp exp)
          shape = makeShape r [exp]
compileTake Nothing args = error "Need instance declaration for take"
compileTake _ _ = error "Take needs 2 arguments"

-- drop --
compileDrop (Just([tp],[r])) [len,exp] = F.FunCall2 "reshape" dims $ F.FunCall fname [sizeProd,resh]
    where dims = maxExp (Constant (Int 0)) (F.BinApp F.Minus (F.FunCall "size" [Constant (Int 0), compileExp exp])  (absExp (compileExp len))) : tail shape
          resh = F.FunCall2 "reshape" [multExp shape] (compileExp exp)
          sizeProd = multExp $ compileExp len : tail shape
          fname = "drop1_" ++ showTp (makeBTp tp)
          shape = makeShape r [exp]

-- reshape --
compileReshape (Just([tp],[r1,r2])) [dims,array] = F.FunCall2 "reshape" dimsList $ F.FunCall fname [dimProd, resh]
    where dimsList | F.Array dimsList <- dimsExp = dimsList
                   | F.Var dimsVar <- dimsExp = map (\i -> F.Index (F.Var dimsVar) [Constant (Int i)]) [0..r2-1]
                   | otherwise = error "reshape needs literal or variable as shape argument"
          dimsExp = compileExp dims
          fname = "reshape1_" ++ showTp (makeBTp tp)
          dimProd = multExp dimsList
          resh = F.FunCall2 "reshape" [shapeProd] (compileExp array)
          shapeProd = multExp (makeShape r1 [array])
compileReshape Nothing args = error "Need instance declaration for reshape"
compileReshape _ _ = error "Reshape needs 2 arguments"

-- transp --
```

```haskell
compileTransp (Just(_,[r])) [exp] = makeTransp2 (map (Constant . Int) (reverse [0..r-1])) (compileExp exp)
compileTransp Nothing args = error "Need instance declaration for transp"
compileTransp _ _ = error "Transpose takes 1 argument"

-- transp2 --
compileTransp2 _ [Vc dims,e] = makeTransp2 (map compileExp dimsExps) (compileExp e)
    where dimsExps = map (I . (\x -> x - 1) . getInt) dims
          getInt (I i) = i
          getInt _ = error "transp2 expects number literals in it's first argument"
compileTransp2 _ e = case e of [_,_] -> error "transp2 needs litaral as first argument"
                               _     -> error "transp2 takes 2 arguments"

-- shape --
compileShape (Just(_,[len])) args = F.Array $ makeShape len args
compileShape Nothing args = error "Need instance declaration for shape"

-- firstV --
compileFirstV _ args
  | [e] <- args = F.Let Inline (Ident "x") (compileExp e) $ F.Index (F.Var "x")[F.Constant (F.Int 0)]
  | otherwise = error "firstV takes one argument"

-- eachV --
compileEachV :: Maybe InstDecl -> [T.Exp] -> F.Exp
compileEachV Nothing _ = error "Need instance declaration for eachV"
compileEachV (Just ([intp,outtp],[len])) [kernel,array] = Map kernelExp (compileExp array)
   where kernelExp = compileKernel kernel (makeBTp outtp)

-- each --
compileEach :: Maybe InstDecl -> [T.Exp] -> F.Exp
compileEach (Just ([intp,outtp],[rank])) [kernel,array] = makeEach intp outtp rank kernel (compileExp array)
  where makeEach tp1 tp2 r kernel array
          | r == 1 = Map (compileKernel kernel (makeBTp tp2)) array
          | otherwise = Map (F.Fn (mkType (tp2,r-1)) [(mkType (tp1,r-1),"x")] (makeEach tp1 tp2 (r-1) kernel (F.Var "x"))) array
compileEach Nothing _ = error "Need instance declaration for each"
compileEach _ _ = error "each takes two arguments"

-- zipWith --
compileZipWith :: Maybe InstDecl -> [T.Exp] -> F.Exp
compileZipWith (Just([tp1,tp2,rtp],[rk])) [kernel,a1,a2] = makeZipWith rk kernel (compileExp a1) (compileExp a2)
  where
  makeZipWith r kernel a1 a2
    | r == 1 = Map (compileKernel kernel (makeBTp rtp)) (FunCall "zip" [a1,a2])
    | otherwise = Map (F.Fn (mkType (rtp,r-1)) [(mkType(tp1,r-1),"x"),(mkType(tp2,r-1),"y")] (makeZipWith (r-1) kernel (F.Var "x") (F.Var "y"))) (FunCall "zip" [a1, a2])
```

```
    --Map kernelExp $ F.FunCall "zip" [(compileExp a1),(compileExp a2)] -- F.Map kernelExp $ F.FunCall "zip" [a1,a2]
compileZipWith Nothing _ = error "Need instance declaration for zipWith"
compileZipWith _ _ = error "zipWith takes 3 arguments"

-- reduce --
compileReduce :: Maybe InstDecl -> [T.Exp] -> F.Exp
compileReduce Nothing _ = error "Need instance declaration for reduce"
compileReduce (Just ([tp],[rank]))[kernel,id,array] = makeReduce tp rank kernelExp (compileExp id) (compileExp array)
  where
  mkType (tp,rank) = makeArrTp (makeBTp tp) rank
  kernelExp = compileKernel kernel (makeBTp tp)
  makeReduce :: BType -> Integer -> Kernel -> F.Exp -> F.Exp -> F.Exp
  makeReduce tp rank kernel idExp arrayExp
    | rank == 0 = Reduce kernel idExp arrayExp
    | otherwise = Map (F.Fn (mkType(tp,rank-1)) [(mkType(tp,rank),"x")] (makeReduce tp (rank-1) kernel idExp (F.Var "x"))) arrayExp
compileReduce _ _ = error "reduce needs 3 arguments"


-- operators that are 1:1  --
-- (library functions) --
idFuns = ["negi",
          "negd",
          "absi",
          "absd",
          "mini",
          "mind",
          "signd",
          "signi",
          "maxi",
          "maxd",
          "eqb",
          "xorb",
          "nandb",
          "norb",
          "neqi",
          "neqd",
          "resi"]

-- operators that are 1:1 with Futhark functions --
convertFun fun = case fun of
  "i2d"    -> Just "toFloat"
  "catV"   -> Just "concat"
  "b2i"    -> Just "boolToInt"
  "b2iV"   -> Just "boolToInt"
  "ln"     -> Just "log"
  "expd"   -> Just "exp"
  "notb"   -> Just "!"
  "floor"  -> Just "trunc"
  _         | fun ‘elem‘ idFuns -> Just fun
            | otherwise -> Nothing


-- binary operators --
convertBinOp op = case op of
  "addi" -> Just F.Plus
```

```
    "addd" -> Just F.Plus
    "subi" -> Just F.Minus
    "subd" -> Just F.Minus
    "muli" -> Just F.Mult
    "muld" -> Just F.Mult
    "ltei" -> Just F.LessEq
    "lted" -> Just F.LessEq
    "eqi"  -> Just F.Eq
    "eqd"  -> Just F.Eq
    "gti"  -> Just F.Greater
    "gtd"  -> Just F.Greater
    "gtei" -> Just F.GreaterEq
    "gted" -> Just F.GreaterEq
    "andb" -> Just F.LogicAnd
    "orb"  -> Just F.LogicOr
    "divi" -> Just F.Div
    "divd" -> Just F.Div
    "powd" -> Just F.Pow
    "powi" -> Just F.Pow
    "lti"  -> Just F.Less
    "ltd"  -> Just F.Less
    "andi" -> Just F.And
    "andd" -> Just F.And
    "ori"  -> Just F.Or
    "shli" -> Just F.Shl
    "shri" -> Just F.Shr
    _      -> Nothing
```

C Pretty printer source code

This appendix contains the source code of the pretty printer used to print the Futhark AST. The pretty printer are located on the following path in the project: `src/Tail2Futhark/Futhark/Pretty.hs`

```
module Tail2Futhark.Futhark.Pretty (prettyPrint)  where

import Text.PrettyPrint
import Tail2Futhark.Futhark.AST

prettyPrint :: Program -> String
prettyPrint = render . vcat . map ppFun

ppFun :: FunDecl -> Doc
ppFun (tp, ident, args, exp) =
  text "fun"
  <+> ppType tp
  <+> text ident
  <> (commaList . map ppArg) args
  <+> equals $+$ nest 2 (ppExp exp)

commaList = parens . hcat . punctuate comma
commaExps = commaList . map ppExp
brackList = brackets . hcat . punctuate comma
brackExps = brackList . map ppExp

ppType :: Type -> Doc
ppType IntT = text "int"
ppType RealT = text "real"
ppType BoolT = text "bool"
ppType CharT = text "char"
ppType (ArrayT at) = brackets (ppType at)

ppExp (Var ident) = text ident
ppExp (Let Indent pat exp1 exp2) = text "let" <+> ppPat pat <+> equals <+> ppExp exp1 <+> text "in" $+$ ppExp exp2
ppExp (Let Inline pat exp1 exp2) = text "let" <+> ppPat pat <+> equals <+> ppExp exp1 <+> text "in" <+> ppExp exp2
ppExp (IfThenElse Indent e1 e2 e3) = text "if" <+> ppExp e1 $+$ text "then" <+> ppExp e2 $+$ text "else" <+> ppExp e3
ppExp (IfThenElse Inline e1 e2 e3) = text "if" <+> ppExp e1 <+> text "then" <+> ppExp e2 <+> text "else" <+> ppExp e3
ppExp (Constant c) = ppConstant c
ppExp (Neg exp)    = text "-" <> ppExp exp
ppExp (Index exp exps) = ppExp exp <> brackExps exps
ppExp (Array exps) = brackExps exps
ppExp (BinApp op e1 e2) = parens $ ppExp e1 <+> ppOp op <+> ppExp e2
ppExp (FunCall ident exps) = text ident <> commaExps exps
ppExp (FunCall2 ident exps exp) = text ident <> parens (commaExps exps <> comma <> ppExp exp)
--ppExp (Reshape exps exp) = text "reshape" <> parens (commaExps exps <> comma <> ppExp exp)
ppExp (Empty tp) = text "empty" <> parens (ppType tp)
ppExp e = case e of
  Map k e         -> pp1 "map" k e
  Filter k e      -> pp1 "filter" k e
  Scan k e1 e2    -> pp2 "scan" k e1 e2
  Reduce k e1 e2  -> pp2 "reduce" k e1 e2
  where pp1 id k e     = text id <> parens ((ppKernel k) <> comma <> ppExp e)
        pp2 id k e1 e2 = text id <> parens ((ppKernel k) <> comma <> ppExp e1 <> comma <> ppExp e2)

```

```
ppKernel (Fn tp args exp) = text "fn" <+> ppType tp <+> (commaList . map ppArg $ args) <+> text "=>" <+> ppExp exp
ppKernel (Fun ident []) = text ident
ppKernel (Fun ident exps) = text ident <+> (commaList . map ppExp $ exps)
ppKernel (Op op) = ppOp op

ppOp op = text $ case op of
  Plus -> "+"
  Minus -> "-"
  LessEq -> "<="
  Mult -> "*"
  Div -> "/"
  Eq -> "=="
  Mod -> "%"
  Greater -> ">"
  Less -> "<"
  GreaterEq -> ">="
  LogicAnd -> "&&"
  LogicOr -> "||"
  Pow -> "pow"
  Or -> "|"
  And -> "&"
  Shl -> ">>"
  Shr -> "<<"
  --XOr -> "^"

ppConstant (Int int) = integer int
ppConstant (Real f) = double f
ppConstant (Char c) = quotes $ char c
ppConstant (Bool b) = text (if b then "True" else "False")
ppConstant (ArrayConstant arr) = braces . hcat . punctuate comma . map ppConstant $ arr

-- Arguments --
ppArg (tp,ident) = ppType tp <+> text ident

-- Pattern --
ppPat :: Pattern -> Doc
ppPat (Ident ident) = text ident
ppPat (TouplePat pat) = braces . hcat . punctuate comma . map ppPat $ pat
```

D Complete Futhark primes code

```
fun int boolToInt(bool x) =
  if x then 1 else 0
fun int negi(int x) =
  -x
fun real negd(real x) =
  -x
fun int absi(int x) =
  if (x <= 0) then -x else x
fun real absd(real x) =
  if (x <= 0.0) then -x else x
fun int mini(int x,int y) =
  if (x <= y) then x else y
fun real mind(real x,real y) =
  if (x <= y) then x else y
fun int signd(real x) =
  if (0.0 < x)
  then 1
  else if (0.0 == x)
       then 0
       else -1
fun int signi(int x) =
  if (0 < x)
  then 1
  else if (0 == x)
       then 0
       else -1
fun int maxi(int x,int y) =
  if (x <= y) then y else x
fun real maxd(real x,real y) =
  if (x <= y) then y else x
fun bool eqb(bool x,bool y) =
  (!((x || y)) || (x && y))
fun bool xorb(bool x,bool y) =
  (!((x && y)) && (x || y))
fun bool nandb(bool x,bool y) =
  !((x && y))
fun bool norb(bool x,bool y) =
  !((x || y))
fun bool neqi(int x,int y) =
  !((x == y))
fun bool neqd(real x,real y) =
  !((x == y))
fun int resi(int x,int y) =
  if (x == 0)
  then y
else if ((((y % x) == 0) || ((y > 0) && (x > 0))) || ((y < 0) &&
    (x < 0)))
       then (y % x)
       else (y % (x + x))
fun [int] reshape1_int(int l,[int] x) =
  let roundUp = ((l + (size(0,x) - 1)) / size(0,x)) in
  let extend = reshape(((size(0,x) * roundUp)),replicate(roundUp,
      x)) in
  let {v1,_} = split((l),extend) in v1
fun [real] reshape1_real(int l,[real] x) =
  let roundUp = ((l + (size(0,x) - 1)) / size(0,x)) in
  let extend = reshape(((size(0,x) * roundUp)),replicate(roundUp,
      x)) in
  let {v1,_} = split((l),extend) in v1
fun [bool] reshape1_bool(int l,[bool] x) =
  let roundUp = ((l + (size(0,x) - 1)) / size(0,x)) in
```

```
  let extend = reshape(((size(0,x) * roundUp)),replicate(roundUp,x)) in
  let {v1,_} = split((l),extend) in v1
fun [char] reshape1_char(int l,[char] x) =
  let roundUp = ((l + (size(0,x) - 1)) / size(0,x)) in
  let extend = reshape(((size(0,x) * roundUp)),replicate(roundUp,x)) in
  let {v1,_} = split((l),extend) in v1
fun [int] take1_int(int l,[int] x) =
  if (0 <= l)
  then if (l <= size(0,x))
       then let {v1,_} = split((l),x) in v1
       else concat(x,replicate((l - size(0,x)),0))
  else if (0 <= (l + size(0,x)))
       then let {_,v2} = split(((l + size(0,x))),x) in v2
       else concat(replicate((l - size(0,x)),0),x)
fun [real] take1_real(int l,[real] x) =
  if (0 <= l)
  then if (l <= size(0,x))
       then let {v1,_} = split((l),x) in v1
       else concat(x,replicate((l - size(0,x)),0.0))
  else if (0 <= (l + size(0,x)))
       then let {_,v2} = split(((l + size(0,x))),x) in v2
       else concat(replicate((l - size(0,x)),0.0),x)
fun [bool] take1_bool(int l,[bool] x) =
  if (0 <= l)
  then if (l <= size(0,x))
       then let {v1,_} = split((l),x) in v1
       else concat(x,replicate((l - size(0,x)),False))
  else if (0 <= (l + size(0,x)))
       then let {_,v2} = split(((l + size(0,x))),x) in v2
       else concat(replicate((l - size(0,x)),False),x)
       fun [char] take1_char(int l,[char] x) =
  if (0 <= l)
  then if (l <= size(0,x))
       then let {v1,_} = split((l),x) in v1
       else concat(x,replicate((l - size(0,x)),' '))
  else if (0 <= (l + size(0,x)))
       then let {_,v2} = split(((l + size(0,x))),x) in v2
       else concat(replicate((l - size(0,x)),' '),x)
fun [int] drop1_int(int l,[int] x) =
  if (size(0,x) <= if (l <= 0) then -l else l)
  then empty(int)
  else if (l <= 0)
       then let {v1,_} = split(((l + size(0,x))),x) in v1
       else let {_,v2} = split((l),x) in v2
fun [real] drop1_real(int l,[real] x) =
  if (size(0,x) <= if (l <= 0) then -l else l)
  then empty(real)
  else if (l <= 0)
       then let {v1,_} = split(((l + size(0,x))),x) in v1
       else let {_,v2} = split((l),x) in v2
fun [bool] drop1_bool(int l,[bool] x) =
  if (size(0,x) <= if (l <= 0) then -l else l)
  then empty(bool)
  else if (l <= 0)
       then let {v1,_} = split(((l + size(0,x))),x) in v1
       else let {_,v2} = split((l),x) in v2
fun [char] drop1_char(int l,[char] x) =
  if (size(0,x) <= if (l <= 0) then -l else l)
  then empty(char)
  else if (l <= 0)
       then let {v1,_} = split(((l + size(0,x))),x) in v1
```

```
       else let {_,v2} = split((1),x) in v2
fun real main() =
  let t_v1 = drop1_int(1,map(fn int (int x) => (x + 1),iota(9999)
    )) in
  let t_v7 = rearrange((1,0),reshape((9998,9998),reshape1_int
    ((9998 * (9998 * 1)),reshape(((size(0,t_v1) * 1)),t_v1))))
    in
  let t_v8 = reshape((9998,9998),reshape1_int((9998 * (9998 * 1))
    ,reshape(((size(0,t_v1) * 1)),t_v1))) in
  let t_v11 = map(fn [int] ([int] x,[int] y) => map(resi,zip(x,y)
    ),zip(t_v7,t_v8)) in
  let t_v13 = map(fn [bool] ([int] x) => map(fn bool (int t_v12)
    => (0 == t_v12),x),t_v11) in
  let t_v18 = rearrange((0),map(fn int ([int] x) => reduce(+,0,x)
    ,map(fn [int] ([bool] x) => map(boolToInt,x),rearrange((1,0)
    ,t_v13)))) in
  let t_v20 = map(fn bool (int t_v19) => (1 == t_v19),t_v18) in
  let t_v24 = reduce(+,0,map(boolToInt,t_v20)) in
  let t_v25 = reshape((2,2),reshape1_int((2 * (2 * 1)),reshape(((
    size(0,[2,3,4,5]) * 1)),[2,3,4,5]))) in
  let t_v28 = map(fn int ([int] x) => reduce(*,1,x),t_v25) in
  toFloat(reduce(+,0,t_v28))
```

E Test results

The tests can be found in the `test/basic_tests/` directory in our project. The expected result of the test can be found in the .ok version of the file. The results in the table below is OK if the result of the compilation (found in the .out file) matches the expected one or FAIL if it does not.

Function	File name	Describtion of test	Result
reshape	reshape.tail	reshapes vector of int with padding	OK
reshape	reshape2.tail	reshape vector into array of rank	OK
vreverse	rev2.tail	reverse of matrix of ints	OK
vreverseV	rev.tail	reverse of vector of ints	OK
vrotate	rotateRank2.tail	rotate array of rank 3 of ints	OK
vrotateV	rotateRank1.tail	rotate vektoof ints	OK
transp	transp2.tail	transpose vector of ints	OK
transp	transp3.tail	transpose array of rank 3 of ints	OK
transp	transpAPL.tail	transpose matrix of ints	OK
transp2	dyadic_transp.tail	transpose of matrix of ints	OK
takeV	take1.tail	positive int on vector with enough elements	OK
takeV	take1neg.tail	negative int on vector with enough elements	OK
take	take2.tail	positive int on matrix with enough elements	OK
dropV	drop2.tail	positive int on vector with enough elements	OK
drop	drop2Dim.tail	drops row in array rank 2 with enough elements	OK
drop	drop2DimNeg.tail	drops with a negative number on array of rank 2	OK
drop	drop2DimtoMuch.tail	drops more rows than there are in the array	OK
drop	drop3Dim.tail	positive int in a 3 dim array	OK
consV	-		
cons	cons1.tail	vector of ints on array of rank 2 of ints	OK
snocV	snocRank1.tail	set scalar on vector	OK
snoc	snocRank2.tail	set vector on matrix	OK
firstV	firstV2.tail	first element of vektor with only one element	OK
firstV	firstV3.tail	first element of vektor of ints	OK
first	first2.tail	first on a matrix	OK
zipWith	zipwith.tail	zip addi over two vectors	OK
zipWith	zipwith2.tail	zip addi over two matrices	OK
zipWith	zipwith3.tail	zip addi over two arrays of rank 3 of ints	OK
catV	-		
cat	cat.tail	cat of arrays of rank 2 of ints	OK
cat	catV.tail	cat of vectors of ints	OK
cat	concat.tail	cat of arrays of rank 2 af ints	OK
reshape	reshape.tail	reshape vector to matrix	OK
reshape	reshape2.tail	reshape with extending the vector	OK

Function	File name	Describtion of test	Result
negi	negi.tail	negtes ints	OK
negd	negd.tail	negate double	OK
ln	blacksholes.tail	-	blacksholes evaluates to correct result
absi	blacksholes.tail	-	blacksholes evaluates to correct result
expd	blacksholes.tail	-	blacksholes evaluates to correct result
mini	mini.tail	min on 2 positive ints	OK
signd	signd.tail	sign of double	OK
notb	not1.tail	not on true	OK
notb	not0.tail	not on false	OK
maxi	maxi.tail	max on 2 positive ints	OK
maxd	maxd.tail	max on 2 positive doubles	OK
ori		NOT TESTET	
subi	subi.tail	substract positive ints	OK
subd	subd.tail	substract positive doubles	OK
muli	muli.tail	multiply 2 positive ints	OK
muld	muld.tail	multiply 2 positive doubles	OK
ltei	ltei.tail	$4 \leq 3$	OK
ltei	lteiTrue.tail	$4 \leq 5$	OK
lted	lted.tail	$2.3 \geq 3.3$	OK
eqi	eqiTrue.tail	$4 = 4$	OK
eqi	eqiFalse.tail	$4 = 5$	OK
eqd	eqdFalse.tail	$3.4 = 1.2$	OK
gti	gtiTrue.apl	$5 > 4$	OK
gtd	gtdTrue.apl	$3.4 > 1.2$	OK
gtei	gteiTrue.tail	$3 \geq 2$	OK
gted	gtedFalse.tail	$3.0 \geq 3.2$	OK
andb	andbTrue.tail	$(2 = 2) \wedge (3 = 3)$	OK
orb	orFalse.tail	$(2 = 1) \vee (3 = 2)$	OK
orb	orTrue.tail	$(2 = 1) \vee (2 = 2)$	OK
divi	divi.tail	$(4 / 2) + 4$	OK
divd	divd.tail	$(4.0 / 2.0) + 4$	OK
powd	powd.tail	FORKERT tester ints	
powi	powi.tail	power on 2 positive ints	OK
lti	ltiTrue.tail	$3 < 5$	OK
ltd	ltdTrue.tail	$3.2 < 5.1$	OK
andi	-		
xorb	xorb.tail	$(3{=}3) \neq (4{=}1)$	OK
i2d	i2d.tail	integer to double	OK
addi	addi.tail	addition of 2 positive ints	OK
addd	addd.tail	$2.3 + 4.5$	OK
iotaV	iotaV.tail	iota in positive integer	OK
iota	iotaV.tail	iota in positive integer	OK
eachV	eachV.tail	add int on vector of ints	OK
each	each.tail	add int on matrix af ints	OK
reduceV	-		
reduce	reduceRank0.tail	reduce on vektor of int	OK
reduce	reduce2.tail	reduce on matrix of int	OK
reduce	reduce3.tail	reduce on array of rank 3	OK
shapeV	firstV2.tail	shape of vector	OK
shape	take2.tail	shape of matrix of ints	OK