Bachelor thesis
Computing Science

Radboud University

A new semantics for array programming languages; how to introduce some laziness without being lazy

Author:
J.M. (Jordy) Aaldering, s1004292
`j.aaldering@student.ru.nl`

First supervisor/assessor:
Prof. dr. S.B. (Sven-Bodo) Scholz
`svenbodo.scholz@ru.nl`

Second assessor:
dr. P.W.M. (Pieter) Koopman
`pieter@cs.ru.nl`

March 17, 2021

Abstract

Array programming languages are often used in mathematical and engineering applications with very large and complicated equations. In these languages all values are represented as arrays, these arrays can usually be multi-dimensional. These multi-dimensional arrays have a shape vector which describes the length of each of its dimensions. The separation of knowledge of these arrays allows programs to potentially be rewritten according to the required level of information of an array, reducing the computational load by only requiring the shape or dimensionality of that array.

This paper defines a way to infer this required level of information and delivers rules for rewriting programs in a way that requires a lower level of information without losing strictness, finding a balance between lazy and strict evaluation. Along with this paper also comes a prototype language, created in a strict functional programming language, which implements these rules. This language will show how to implement these rules in the real world and will be used to validate and benchmark the results.

Contents

1 Introduction

In this paper we will be discussing a way of optimising array programming languages. In array programming languages all values are represented as, possibly multi-dimensional, arrays. These arrays also come with a shape vector, which gives the length of each dimension. Sometimes when executing a program we will not need the actual values of an array. Perhaps we only need the shape or dimensionality, or even nothing at all. Our goal then is to rewrite that program in such a way that any unused information is removed, avoiding unnecessary computations. If we apply this idea to a strictly evaluated language we get some of the benefits of lazy evaluation, without the added overhead it introduces. This idea of generating new programs by optimising them before evaluation is called partial evaluation[4][5][11].

We will start this paper by defining a small prototype language, to which we will apply our rewrite. This rewrite is a set of rules, one for each expression in the language, which defines how a certain expression has to be rewritten for each required level of information. We will also discuss what it means for these rules to be correct, which will then be proven using induction. To be able to define these rules we must also know the minimal required level of information, so we will first have to find a way to do this before we are able to define our rewrite rules. Previous works have also looked at possible ways of finding this required level of information[1][2][12].

The implementation of the prototype language will briefly be explained, after which we will use it to evaluate the impact of these rewrite rules on the runtime of an example program, which will show that this rewrite has a noticeable impact on the performance of a small but realistic program.

2 Background

2.1 Array programming

Array programming languages are a type of programming languages that are often used for scientific and engineering applications. They are languages in which the only data types are scalars and arrays. An array is a list of numbers along with some shape vector, this shape gives the length of each dimension of the array. These arrays can usually be of any dimension. Scalars are simply a decimal or integer value, which is internally represented as an array with a dimensionality of 0.

Because the only data types are arrays, array programming languages allow the programmer to apply operations to an entire set of values, without having to result to explicit loops over each individual element like in other languages. This also makes programs easier to read, as they more closely resemble the mathematical notation.

Two important terms in array programming are shape and dimensionality. The shape describes the length of each dimension of an array. A special case are scalars; they have an empty shape: [ ], note that an empty array would be have a shape of zeros: [0], and not an empty shape. The dimensionality describes the number of dimensions, or rank, of an array. It corresponds to the length of the shape vector.

2.2 Strict vs. lazy evaluation

The language we will be constructing will try to find a balance between strict and lazy evaluation. These are two different ways of evaluating a program. A strict program evaluates expressions immediately when it encounters them, and a lazy program only evaluates expressions when their resulting values are required. Both of these have their up- and downsides. Strict evaluation potentially has to do more calculations, if the resulting values are not required later, but for lazy evaluation to avoid this it has to keep track of the program in order to be able to backtrack and evaluate expressions only when required. One thing to note here, because we will encounter it later, is that a program that runs properly in a lazy evaluation might not run in a strict evaluation, as problematic expressions that, for instance, produce a runtime error might not have been evaluated by the lazy evaluation.

3 Syntax

We start this paper by creating a small array programming language, which we will use throughout this paper. This language will have the basic functionality which most array programming languages have, with which we will be able to create most simple programs. The syntax follows a lambda-calculus style, where a program is defined as a single expression, recursively containing sub-expressions. Every program must evaluate to a value.

3.1 Grammar

In this section we will use the Backus–Naur form[10] to explain the program's control structure, with the added symbol '`+`' which means 'one or more'. Anything between quotes are the literal characters of the syntax. We will also allow for comments in our language; comments start at a '`#`', and stop at the end of a line.

Our language consists of scalars, which are simply a decimal value, and arrays. These arrays can be made up of numbers, but they can also contain expressions. Additionally; arrays can also be empty, meaning they do not contain any values. We also have variable identifiers which are strings that point variable names to their values.

$$\begin{aligned}\langle \textit{value} \rangle \; &::= \langle \textit{scalar} \rangle \\ &\;\;|\;\; \langle \textit{array} \rangle \\ &\;\;|\;\; \langle \textit{var-id} \rangle\end{aligned}$$

$$\langle \textit{scalar} \rangle \; ::= \langle \textit{decimal} \rangle$$

$$\langle \textit{array} \rangle \; ::= \text{`}\texttt{[]}\text{'} \mid \text{`}\texttt{[}\text{'}\ \langle \textit{expr} \rangle\ (\text{`}\texttt{,}\text{'}\ \langle \textit{expr} \rangle)+\ \text{`}\texttt{]}\text{'}$$

Since all our expressions evaluate to values, we can easily define binary and unary operations to transform values. In the case of binary operations, the right expression must always be of the same shape as the left expression or it must be a scalar.

$$\langle \textit{binary} \rangle \; ::= \langle \textit{expr} \rangle\ \langle \textit{bop} \rangle\ \langle \textit{expr} \rangle$$

$$\langle \textit{unary} \rangle \; ::= \langle \textit{uop} \rangle\ \langle \textit{expr} \rangle$$

⟨*bop*⟩ ::= ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’ (maths)
| ‘`=`’ | ‘`!=`’ | ‘`>`’ | ‘`>=`’ | ‘`<`’ | ‘`<=`’ (equality)

⟨*uop*⟩ ::= ‘`-`’ | ‘`||`’ (maths)
| ‘`!`’ (equality)

We can assign values to a variable by using a let-expression. This expression takes the variable name and an expression to compute its value. This variable will then be in scope for the second expression, after the ‘in’. Similarly we can also assign a lambda expression to an identifier, which essentially makes the identifier the name of a function which can later be called with a number of arguments. These functions can be recursive.

⟨*let*⟩ ::= ‘`let`’ ⟨*var-id*⟩ ‘`=`’ ⟨*expr*⟩ ‘`in`’ ⟨*expr*⟩

⟨*fun-def*⟩ ::= ‘`let`’ ⟨*fun-id*⟩ ‘`=`’ ⟨*lambda*⟩ ‘`in`’ ⟨*expr*⟩

⟨*lambda*⟩ ::= (‘\’ ⟨*var-id*⟩ ‘`.`’)+ ⟨*expr*⟩

So now we need to be able to call these user-defined functions, as well as our built-in primitive functions, by passing them values which gives us some result. The first argument of selection is the array from which we want to select, the second argument is the index at which to select. The selected value can be a scalar, but it can also be an array; when the dimensionality of the selection index is lower than that of the array. We can also call a lambda function directly instead of assigning it to a function identifier first. Every function call takes at least one, and potentially multiple, arguments.

⟨*call*⟩ ::= ⟨*prf*⟩ (primitive function)
| ⟨*fun-id*⟩ ⟨*expr*⟩+ (user-defined function)
| ‘`(`’ ⟨*lambda*⟩ ‘`)`’ ⟨*expr*⟩+ (lambda application)

⟨*prf*⟩ ::= ⟨*expr*⟩ ‘`.(`’ ⟨*expr*⟩ ‘`)`’ (selection)
| ‘`shape`’ ⟨*expr*⟩ (shape)
| ‘`dim`’ ⟨*expr*⟩ (dimensionality)

We also define a conditional expression. Since our language only has scalars and arrays, and no boolean values, we define a scalar as being false if it is 0, else it is true. One thing to note is that there there must always be an else branch. This follows from the fact that an expression must always evaluate to a value, and thus the expression must evaluate to a value for both cases.

⟨*if*⟩ ::= ‘`if`’ ⟨*expr*⟩ ‘`then`’ ⟨*expr*⟩ ‘`else`’ ⟨*expr*⟩

Finally we have the with expression, which is arguably the most useful in array programming languages. We use this expression to generate arrays. This expression first takes a shape for the new value to generate, along with a default argument for the iteration. If the shape of this argument has size n, that shape should correspond with the last n arguments of the first expression. Then it takes two bounds, and a variable name that will iteratively be assigned all possible values within those bounds, the dimensionality of these bounds should be the dimensionality of the first expression minus n. Then the in-expression is expected to return a value of the same shape as the default value, which is then placed at the correct spot in the new array. We also allow a shorter case, which simply assigns the default value to every element.

$$
\begin{array}{lll}
\langle \textit{with} \rangle & ::= & \text{‘}\texttt{gen}\text{’}\ \langle \textit{expr} \rangle\ \langle \textit{expr} \rangle \\
 & & \text{‘}\texttt{with}\text{’}\ \langle \textit{expr} \rangle\ \text{‘}\texttt{<=}\text{’}\ \langle \textit{var-id} \rangle\ \text{‘}\texttt{<}\text{’}\ \langle \textit{expr} \rangle \\
 & & \text{‘}\texttt{in}\text{’}\ \langle \textit{expr} \rangle \\
 & | & \text{‘}\texttt{gen}\text{’}\ \langle \textit{expr} \rangle\ \langle \textit{expr} \rangle
\end{array}
$$

Combining these we get the core expression, which we use to generate our programs.

$$
\begin{array}{lll}
\langle \textit{program} \rangle & ::= & \langle \textit{expr} \rangle \\
 & & \\
\langle \textit{expr} \rangle & ::= & \langle \textit{fun-def} \rangle \\
 & | & \langle \textit{call} \rangle \\
 & | & \langle \textit{let} \rangle \\
 & | & \langle \textit{if} \rangle \\
 & | & \langle \textit{with} \rangle \\
 & | & \langle \textit{binary} \rangle \\
 & | & \langle \textit{unary} \rangle \\
 & | & \langle \textit{value} \rangle
\end{array}
$$

3.2 Example

We will now look at part of a program as an example to see how this grammar works in practise. This example will be used throughout the paper. In this example we define a function `shift n arr`, this function shifts the values of the given array `n` spots to the right if `n` is positive or `n` to the left otherwise. With zeros at all empty positions.

```
let shift = \n.\arr.
    let pad = gen (shape (take n arr)) 0 in
    let xs = drop (-n) arr in
    if n > 0 then pad ++ xs
        else xs ++ pad
in
```

We start by defining a function `shift n arr`, which takes two arguments. We then assign a value to the variable `pad` by using a with-expression. This with-expression creates a new array with a shape equal to the shape of the result of the function `take n arr`, which is user-defined. Then we assign a value to the variable xs with the other user-defined function `drop n arr`. Then we go to the conditional expression and, depending on the value of n, we append the values of the variables pad and xs. Here we have no more expressions so the result of this conditional expression is then the returned value when calling the function `shift n arr`.

3.3 Operational semantics

Now that we have a grammar, we know how a program is formed but not how it gives us a result. For this we will define semantics that tell us how we get the results from expressions. We will formalise the semantics of this language using big-step operational semantics. Here we will denote a value with dimensionality n as a pair of shape and data: $\langle [shp_1, \ldots, shp_n],\, [data_1, \ldots, data_m] \rangle$, where we have $m = \prod_{i=1}^{n} shp_i$.

This semantics will make use of a program environment: σ, which maps all variable names that are currently in scope to their corresponding values.

First we look at how scalars and vectors are transformed into their internal representation. A scalar is simply placed in an array with an empty shape. An array can be multi-dimensional and thus the elements can be arrays themselves, note here that all of these elements must evaluate to the same shape.

$$\text{scalar}\,\frac{}{(x,\, \sigma) \Downarrow \langle [\,],\, [x] \rangle}$$

$$\text{array}\,\frac{\forall_{i \in [1,n]} \colon (e_i,\, \sigma) \Downarrow \langle [shp_1, \ldots, shp_m],\, [data^i_1, \ldots, data^i_p] \rangle}{([e_1, \ldots, e_n],\, \sigma) \Downarrow \langle [n, shp_1, \ldots, shp_m],\, [data^1_1, \ldots, data^1_p, \ldots, data^n_1, \ldots, data^n_p] \rangle}$$

$$\text{where } p = \prod_{i=1}^{m} shp_i$$

We can also get a value from a variable name, which we find by looking that variable name up in the program environment σ.

$$\text{var}\,\frac{}{(s,\, \sigma) \Downarrow \sigma[s] = \langle [shp_1, \ldots, shp_n],\, [data_1, \ldots, data_m] \rangle}$$

Next we have the primitive functions `dim`, `shape`, and `sel`. We can get both the dimensionality and shape by looking at the shape vector of the value.

$$\text{dim}\,\frac{(e,\, \sigma) \Downarrow \langle [shp_1, \ldots, shp_n],\, \_ \rangle}{(\text{dim } e,\, \sigma) \Downarrow \langle [\,],\, [n] \rangle}$$

$$\text{shape}\,\frac{(e,\, \sigma) \Downarrow \langle [shp_1, \ldots, shp_n],\, \_ \rangle}{(\text{shape } e,\, \sigma) \Downarrow \langle [n],\, [shp_1, \ldots, shp_n] \rangle}$$

Selection can be done with an index vector with a lower dimensionality than that of the array we are selecting from, giving a sub-array of that array as a result. The dimensionality of this result is the difference of the dimensionality of the index vector and the array. If this difference were to give us $[shp_1, \ldots, shp_0]$ we will read that as an empty shape: $[\,]$.

We get the values of the result by first finding the start index. We find that index using a function `row_major`, it is not important how this function works exactly. Then from there we take the correct amount of values, which we get from the index vector.

$$\text{sel}\,\frac{\begin{array}{c}(iv,\,\sigma) \Downarrow \langle [n],\, [idx_1, \ldots, idx_n] \rangle \\ (e,\,\sigma) \Downarrow \langle [shp_1, \ldots, shp_p],\, [data_1, \ldots, data_q] \rangle\end{array}}{(\text{sel}\; iv\, e,\,\sigma) \Downarrow \langle [shp_1, \ldots, shp_{p-n}],\, [data_x, \ldots, data_{x+s-1}] \rangle}$$

$$\begin{array}{c}\text{where } x = row\_major\ iv\ e \\ \text{and } s = \prod_{i=1}^{p-n} shp_i\end{array}$$

Unary mathematical operators apply the operator to each element of the array. Binary mathematical operators work element-wise, meaning that the two arrays must be of the same shape. Additionally the right hand array may also be a scalar, which is internally broadcast to be of the same shape as the left hand array. Broadcasting means that if the right hand side is a scalar, it becomes an array of the same shape as the left hand argument filled with the original value of that scalar.

$$\text{unary}\,\frac{(e,\,\sigma) \Downarrow \langle [shp_1, \ldots, shp_n],\, [data_1, \ldots, data_m] \rangle}{(\texttt{uop}\; e,\,\sigma) \Downarrow \langle [shp_1, \ldots, shp_n],\, [\texttt{uop}\; data_1, \ldots, \texttt{uop}\; data_m] \rangle}$$

$$\text{binary}\,\frac{\begin{array}{c}(e_l,\,\sigma) \Downarrow \langle [shp_1, \ldots, shp_n],\, [data^l_1, \ldots, data^l_m] \rangle \\ (e_r,\,\sigma) \Downarrow \langle [shp_1, \ldots, shp_n],\, [data^r_1, \ldots, data^r_m] \rangle\end{array}}{(e_l\; \texttt{bop}\; e_r,\,\sigma) \Downarrow \langle [shp_1, \ldots, shp_n],\, [(data^l_1\; \texttt{bop}\; data^r_1), \ldots, (data^l_m\; \texttt{bop}\; data^r_m)] \rangle}$$

Unary and binary equality operators work slightly differently, as they will always return a boolean value, which is internally represented as a scalar of value 0 for false, and 1 for true. For the unary case the shape does not matter, but for the binary case we will have to make sure both sides are of the same shape.

$$\text{unary}\,\frac{(e,\,\sigma) \Downarrow \langle \_,\, [data_1, \ldots, data_m] \rangle}{(\texttt{uop}\; e,\,\sigma) \Downarrow \langle [\,],\, [\texttt{uop}\; data_1 \wedge \cdots \wedge \texttt{uop}\; data_m] \rangle}$$

$$\text{binary}\,\frac{\begin{array}{c}(e_l,\,\sigma) \Downarrow \langle [shp_1, \ldots, shp_n],\, [data^l_1, \ldots, data^l_m] \rangle \\ (e_r,\,\sigma) \Downarrow \langle [shp_1, \ldots, shp_n],\, [data^r_1, \ldots, data^r_m] \rangle\end{array}}{(e_l\; \texttt{bop}\; e_r,\,\sigma) \Downarrow \langle [\,],\, [(data^l_1\; \texttt{bop}\; data^r_1) \wedge \cdots \wedge (data^l_m\; \texttt{bop}\; data^r_m)] \rangle}$$

In the with expression we use the generator to get the shape of the result and the default argument. This default argument corresponds to some sub-list of the shape. We then find the range, to which we will apply the in-expression, from the lower and upper bounds. For each index in this range we add a variable x to the environment that maps to this index. The resulting value, which can be a scalar or an array, is then placed at the correct position in the result, replacing the default argument.

$$\text{with}\;\frac{\begin{array}{c}(e_{gen},\,\sigma) \Downarrow \langle [n],\, [shp_1,\ldots,shp_n] = shp\rangle \\ (e_{def},\,\sigma) \Downarrow \langle [shp_p,\ldots,shp_n],\, [def_1,\ldots,def_m] = def\rangle \\ (e_l,\,\sigma) \Downarrow \langle [p],\, [idx^l_1,\ldots,idx^l_p] = l\rangle \\ (e_u,\,\sigma) \Downarrow \langle [p],\, [idx^u_1,\ldots,idx^u_p] = u\rangle \\ \forall_{i\in[l,u)} : (e,\,\sigma\{\mathrm{x}\to i\}) \Downarrow \langle [shp_p,\ldots,shp_n],\, [data^i_1,\ldots,data^i_q] = d^i\rangle\end{array}}{\left(\begin{array}{l}\text{gen } e_{gen}\ e_{def} \\ \text{with } e_l \le x < e_u\end{array} \text{in}\, e,\,\sigma\right) \Downarrow \langle shp,\, [def,\ldots,def, d^l,\ldots,d^{u-1}, def,\ldots,def]\rangle}$$

In the conditional expression then, the conditional must evaluate to a scalar. A scalar internally represents false if it is 0 and true otherwise. Depending on this scalar the conditional expression then evaluates to either the true case or the false case. These cases do not have to be of the same shape.

$$\text{cond}\;\frac{\begin{array}{c}(e_c,\,\sigma) \Downarrow \langle [\,],\, [x]\rangle \\ (e_t,\,\sigma) \Downarrow \langle [shp^t_1,\ldots,shp^t_n],\, [data^t_1,\ldots,data^t_m]\rangle = v_t \\ (e_f,\,\sigma) \Downarrow \langle [shp^f_1,\ldots,shp^f_p],\, [data^f_1,\ldots,data^f_q]\rangle = v_f\end{array}}{(\text{if } e_c \text{ then } e_t \text{ else } e_f,\,\sigma) \Downarrow \text{if } x \neq 0 \text{ then } v_t \text{ else } v_f}$$

We add variables to the program environment using the let-expression. We get the result of the first expression and assign it to the variable name in the environment of the second expression, where this variable name can then be used to get that result.

$$\text{let}\;\frac{\begin{array}{c}(e_1,\,\sigma) \Downarrow \langle [shp^x_1,\ldots,shp^x_n],\, [data^x_1,\ldots,data^x_m]\rangle = v_1 \\ (e_2,\,\sigma\{\mathrm{x}\to v_1\}) \Downarrow \langle [shp_1,\ldots,shp_p],\, [data_1,\ldots,data_q]\rangle\end{array}}{(\text{let}\, x = e_1 \,\text{in}\, e_2,\,\sigma) \Downarrow \langle [shp_1,\ldots,shp_p],\, [data_1,\ldots,data_q]\rangle}$$

The let-expression can also be used to add functions to the environment. These functions must also be able to use variables defined before this function, so the function needs to keep track of its own environment. We call this a closure.

$$\text{fun-def}\;\frac{\begin{array}{c}(\lambda x.\, e_1,\,\sigma) \Downarrow \texttt{cls}(\lambda x.\, e_1,\,\sigma) \\ (e_2,\,\sigma\{\mathrm{f}\to \texttt{cls}(\lambda x.\, e_1,\,\sigma)\}) \Downarrow \langle [shp_1,\ldots,shp_n],\, [data_1,\ldots,data_m]\rangle\end{array}}{(\text{let}\, f = \lambda x.\, e_1 \,\text{in}\, e_2,\,\sigma) \Downarrow \langle [shp_1,\ldots,shp_n],\, [data_1,\ldots,data_m]\rangle}$$

Now that we can add lambda expressions to the environment, we can apply arguments to these expressions to get their results. These arguments must be values since we do not have a higher-order language. When the expression is another function, we use currying to apply multiple parameters. So an application can result in a value, or in a closure to which we can again apply an argument.

$$\text{apply}\;\frac{\begin{array}{r}(a,\,\sigma) \Downarrow \langle [shp^a_1,\ldots,shp^a_n],\, [data^a_1,\ldots,data^a_m]\rangle = v_a \\ (e,\,\sigma'\{\mathrm{x}\to v_a\}) \Downarrow \big(\text{if } e = \lambda y.\, e_2 \text{ then } \texttt{cls}(\lambda y.\, e_2,\, \sigma'[x \to v_a]) \\ \text{else } \langle [shp_1,\ldots,shp_p],\, [data_1,\ldots,data_q]\rangle\big) = v_e\end{array}}{((\texttt{cls}(\lambda x.\, e,\,\sigma'))\, a,\,\sigma) \Downarrow v_e}$$

4 Inference

Our goal now is to rewrite programs in this language in a way that minimises the level of required information. Lets go back to the `shift n arr` example.

```
let shift = \n.\arr.
    let pad = gen (shape (take n arr)) 0 in
    let xs = drop (-n) arr in
    if n > 0 then pad ++ xs
        else xs ++ pad
in
```

Here we see that we take the shape of the `take n arr` function. So perhaps we can rewrite `shape (take n arr)` to some new function `take_s n' arr'`, which simplifies the computations within. This is possible because we know that we do not need the actual values of this result, only its shape, which allows us to simplify all expressions computing these values.

We need a way to find out how far we can rewrite expressions, which we are going to do by finding a demand environment. Rewriting the `take` expression might also reduce the required level of information of the arguments `n` and `arr`, allowing us to also rewrite those. Here they are simply variables but keep in mind that these can also be complex expressions, which can benefit greatly from a rewrite.

To do so we need to know the demand of these arguments `n` and `arr` within the function `take_s`, we call this the propagation vector of `take_s`. This propagation vector is a list of demands, with a demand for each argument. For instance; the propagation vector of selection, which takes two arguments, is $[[0, 2, 2, 3], [0, 1, 2, 3]$. In this chapter we will see how we find these propagation vectors and demand environments.

4.1 Demand level

As discussed before. a value consists of three parts; the data, the shape of that data, and its corresponding dimensionality. Including the case where no information is required at all we get four levels of demands, from high to low: full, shape, dimensionality, and none. Which we will assign the values 3 through 0 respectively.

The required level of information, the demand, is represented as an array containing four values. Each index number corresponds with the requested level of information, and the number at that index describes the lowest information level we can rewrite the expression to in order to retain the requested information.

So if, for example, we have a demand $[0, 2, 2, 3]$ and we wish to rewrite according to a demand level of dimensionality (1), we take index 1, which is a 2, which tells us that we must rewrite this argument with a demand level of shape in order to retain the requested dimensionality information of the result.

4.2 Propagation vectors

The function $\mathcal{PV}(expr,\ \gamma)$ takes a primitive expression and returns the propagation vectors of that expression, which is a list of demands with a demand for each argument of the expression. The indices of these demands correspond with the indices of the arguments of the expression.

In case of the operands; mathematical operations are straightforward and return the identity demand. Equality always results in a 0 or 1 so we always already know its dimensionality and shape. The demands of the primitive functions can easily be inferred from the semantics.

$$\begin{aligned}
\mathcal{PV}(\texttt{bop},\ \gamma) &= \begin{cases} [[0,1,2,3],[0,1,2,3]] & \text{if } \texttt{bop} \in \mathit{maths} \\ [[0,0,0,3],[0,0,0,3]] & \text{if } \texttt{bop} \in \mathit{equality} \end{cases} \\
\mathcal{PV}(\texttt{uop},\ \gamma) &= \begin{cases} [[0,1,2,3]] & \text{if } \texttt{uop} \in \mathit{maths} \\ [[0,0,0,3]] & \text{if } \texttt{uop} \in \mathit{equality} \end{cases} \\
\mathcal{PV}(\mathit{sel},\ \gamma) &= [[0,2,2,3],[0,1,2,3]] \\
\mathcal{PV}(\mathit{shape},\ \gamma) &= [[0,0,1,2]] \\
\mathcal{PV}(\mathit{dim},\ \gamma) &= [[0,0,0,1]]
\end{aligned}$$

We can also get the propagation vector of a lambda-expression, we first get the demand environment of the inner expression. We discussed this demand environment in our example, its implementation will be explained in the next section. Then from that environment we take the demands of the variable names of the lambda.

$$\begin{aligned}
\mathcal{PV}(\lambda s_1, \ldots, \lambda s_n.\, e,\ \gamma) = [\mathit{env}[s_1], \ldots, \mathit{env}[s_n]] \\
\text{where } \mathit{env} = \mathcal{SD}(e,\ [0,1,2,3],\ \gamma)
\end{aligned}$$

4.3 Shape demand

To find an environment of these demands we define a function $\mathcal{SD}$. The function $\mathcal{SD}(expr,\, dem,\, \gamma)$ takes an expression, the current demand which we got from the previous expression, and the environment γ containing pre-computed propagation vectors for all user-defined functions. It returns an environment with a demand for each free variable of the given expression.

We also define the operator $\oplus$ which takes the union of the given demand environments. If a variable exists in multiple environments the element wise maximum of its demand vectors is taken. As an example we will take $\{\mathrm{x}\colon[0,1,2,3],\ \mathrm{y}\colon[1,1,2,2]\}\oplus\{\mathrm{y}\colon[0,0,3,3]\}$. Here x only occurs on the left, so it will stay the same. But y occurs on both side, so we will look at both demands and take the maximum of the two at each index. This will then give us the new environment $\{\mathrm{x}\colon[0,1,2,3],\ \mathrm{y}\colon[1,1,3,3]\}$.

We have two non-recursive cases; *value* (a scalar or an array) and `VarId`. A value produces no demand, as we already have the value itself. A variable name adds that variable to the environment with the current demand.

$$\begin{aligned}\mathcal{SD}(value,\, dem,\, \gamma) &= \emptyset\\ \mathcal{SD}(\texttt{VarId},\, dem,\, \gamma) &= \{\texttt{VarId}\colon dem\}\end{aligned}$$

The let-expression combines two demand environments. The demand of the first expression depends on the demand of the second one, since the second expression uses the result of this expression. We get that demand by getting the propagation vector, and combining it with our current demand. Then we take the demand environment of the second expression with our original demand, but since we provide the variable s in this let-expression, we remove it from the demand environment.

$$\begin{aligned}\mathcal{SD}(\mathrm{let}\, s = e_1 \,\mathrm{in}\, e_2,\, dem,\, \gamma) &= \mathcal{SD}(e_1,\, dem_s,\, \gamma)\\ &\quad\oplus (\mathcal{SD}(e_2,\, dem,\, \gamma) \setminus \{s\})\\ &\quad\text{where } dem_s = \mathcal{PV}(\lambda s.\, e_2,\, \gamma)[0][dem]\end{aligned}$$

Since we always need the result of the condition if we want to know anything about the result, we must use a demand of $[0,3,3,3]$ for the condition of the conditional expression. Then we can simply combine it with the demand environments of the two branches since we have no way to know which branch will terminate at this point.

$$\begin{aligned}\mathcal{SD}(\mathrm{if}\, e_c \,\mathrm{then}\, e_t \,\mathrm{else}\, e_f,\, dem,\, \gamma) &= \mathcal{SD}(e_c,\, [0,3,3,3],\, \gamma)\\ &\quad\oplus \mathcal{SD}(e_t,\, dem,\, \gamma)\\ &\quad\oplus \mathcal{SD}(e_f,\, dem,\, \gamma)\end{aligned}$$

The with-expression is a bit more complicated. e_{gen} and e_{def} can be calculated as normal. But the bounds depend on the demand of the last expression, so like before we take the propagation vector of that expression and use it to get the demand environments of our bounds. Similar to in the let-expression we remove s from the demand environment of the last expression since it is provided by the with-expression itself.

$$\begin{aligned}
\mathcal{SD}\Big(\begin{matrix}\text{gen } e_{gen}\ e_{def} \\ \text{with } e_l \le s < e_u\end{matrix} \text{ in } e,\ dem,\ \gamma\Big) &= \mathcal{SD}(e_{gen},\ dem,\ \gamma) \\
&\oplus \mathcal{SD}(e_{def},\ dem,\ \gamma) \\
&\oplus \mathcal{SD}(e_l,\ dem_s,\ \gamma) \\
&\oplus \mathcal{SD}(e_u,\ dem_s,\ \gamma) \\
&\oplus (\mathcal{SD}(e,\ dem,\ \gamma) \setminus \{s\}) \\
&\text{where } dem_s = \mathcal{PV}(\lambda s.\, e,\ \gamma)\,[0][dem]
\end{aligned}$$

Finding the demand of a primitive expression is simple, since the propagation vectors are known to us, as we have already seen in section 4.2. So we can simply get this demand and combine it with our current demand, to then pass it through to the parameters of the primitive expression.

When calling a user-defined function we do something similar, but instead of finding the propagation vector we look up the demand of the function in the pre-computed function environment γ.

$$\begin{aligned}
\mathcal{SD}(\texttt{Prf}\, e_0\ \ldots\ e_n,\ dem,\ \gamma) &= \mathcal{SD}(e_0,\ dem_0,\ \gamma) \oplus \cdots \oplus \mathcal{SD}(e_n,\ dem_n,\ \gamma) \\
&\quad \text{where } dem_i = \mathcal{PV}(\texttt{prf},\ \gamma)\,[i][dem] \\
\mathcal{SD}(\texttt{FunId}\, e_0\ \ldots\ e_n,\ dem,\ \gamma) &= \mathcal{SD}(e_0,\ dem_0,\ \gamma) \oplus \cdots \oplus \mathcal{SD}(e_n,\ dem_n,\ \gamma) \\
&\quad \text{where } dem_i = \gamma[\texttt{FunId}][i][dem]
\end{aligned}$$

4.4 Example

As an example, let's take a look at a function '`take n arr`', which we have talked about before but not yet seen. This function takes the first n values from the list arr if n is positive, or the last n values of arr if n is negative. In our language this function looks as follows:

```
let take = \n.\arr.
    let ofs = if n > 0 then 0               # offset
        else (sel [0] (shape arr)) + n
    in
    gen |n| 0 with [n * 0] <= iv < [|n|] in
        sel (iv + ofs) arr
in
```

Say we want to find the propagation of this function, we then need to solve:

$$\begin{aligned}
\mathcal{PV}(\lambda n.\, \lambda arr.\, e_{body},\ \gamma) &= [env[n],\ env[arr]] \\
&\quad \text{where } env = \mathcal{SD}(e_{body},\ [0, 1, 2, 3],\ \gamma)
\end{aligned}$$

For readability purposes we will start at the end and work our way up. So lets first take a look at the inner expression of the with-loop; `sel (iv + offset) arr`, with the identity demand $[0,1,2,3]$. First we apply the rule for primitive functions. which will require us to get the propagation vector for `sel`:

$$\begin{aligned}
\mathcal{SD}(\text{sel}\ (iv + \mathit{ofs})\ arr,\ [0,1,2,3],\ \gamma) &= \mathcal{SD}(iv + \mathit{ofs},\ \mathcal{PV}(sel,\ \gamma)\,[0][0,1,2,3],\ \gamma) \\
&\quad \oplus \mathcal{SD}(arr,\ \mathcal{PV}(sel,\ \gamma)\,[1][0,1,2,3],\ \gamma) \\
\mathcal{PV}(sel,\ \gamma) &= [[0,2,2,3],\ [0,1,2,3]]
\end{aligned}$$

This gives us two cases we need to solve. One for the binary operator for addition, and one for the variable named `arr`.

$$\begin{aligned}
\mathcal{SD}(iv + \mathit{ofs},\ pv[0][0,1,2,3],\ \gamma) &= \mathcal{SD}(iv + \mathit{ofs},\ [0,2,2,3],\ \gamma) \\
&= \mathcal{SD}(iv,\ \mathcal{PV}(+,\ \gamma)\,[0][0,2,2,3],\ \gamma) \\
&\quad \oplus \mathcal{SD}(\mathit{ofs},\ \mathcal{PV}(+,\ \gamma)\,[1][0,2,2,3],\ \gamma) \\
&= \mathcal{SD}(iv,\ [0,1,2,3][0,2,2,3],\ \gamma) \\
&\quad \oplus \mathcal{SD}(\mathit{ofs},\ [0,1,2,3][0,2,2,3],\ \gamma) \\
&= \mathcal{SD}(iv,\ [0,2,2,3],\ \gamma) \oplus \mathcal{SD}(\mathit{ofs},\ [0,2,2,3],\ \gamma) \\
&= \{\text{iv}: [0,2,2,3]\} \oplus \{\text{ofs}: [0,2,2,3]\} \\
&= \{\text{iv}: [0,2,2,3],\ \text{ofs}: [0,2,2,3]\} \\
\mathcal{SD}(arr,\ pv[1][0,1,2,3],\ \gamma) &= \mathcal{SD}(arr,\ [0,1,2,3],\ \gamma) \\
&= \{\text{arr}: [0,1,2,3]\}
\end{aligned}$$

Combining these results, we get the following demand environment for `sel`.

$$\begin{aligned}
\mathcal{SD}(\text{sel}\ (iv + \mathit{ofs})\ arr,\ [0,1,2,3],\ \gamma) &= \{\text{iv}: [0,2,2,3],\ \text{ofs}: [0,2,2,3]\} \oplus \{\text{arr}: [0,1,2,3]\} \\
&= \{\text{iv}: [0,2,2,3],\ \text{ofs}: [0,2,2,3],\ \text{arr}: [0,1,2,3]\}
\end{aligned}$$

We can now use this to get the environment for the entire with-expression. This requires us to find the five inner environments of that expression.

$$\begin{aligned}
\mathcal{SD}\left(\begin{matrix}\text{gen}\ |n|\ 0 \\ \text{with}\ [n * 0] \le iv < [|n|]\end{matrix}\ \text{in}\, e,\ [0,1,2,3],\ \gamma\right) &= \mathcal{SD}(|n|,\ [0,1,2,3],\ \gamma) \\
&\quad \oplus \mathcal{SD}(0,\ [0,1,2,3],\ \gamma) \\
&\quad \oplus \mathcal{SD}([n * 0],\ dem_{iv},\ \gamma) \\
&\quad \oplus \mathcal{SD}([|n|],\ dem_{iv},\ \gamma) \\
&\quad \oplus (\mathcal{SD}(e,\ [0,1,2,3],\ \gamma) \setminus \{iv\}) \\
&\text{where } dem_{iv} = \mathcal{PV}(\lambda iv.\, e,\ \gamma)\,[dem]
\end{aligned}$$

Going from top to bottom, we get the first environment by finding the propagation vector of the primitive function for the absolute value '||'. The second environment is just a scalar value, so this gives the empty set. The third and fourth case are similar to the first one, usually we would have to keep in mind that these use the newly found demand, instead of the original one, but in this case we will not have to do this computation since both expressions are just arrays and will simply evaluate to an empty environment, whatever the demand is. This allows us to skip some computations and leaves us with only the following two cases:

$$\begin{aligned}
\mathcal{SD}(|n|,\ [0,1,2,3],\ \gamma) &= \mathcal{SD}(n,\ \mathcal{PV}(||,\ \gamma)\,[0,1,2,3],\ \gamma) \\
&= \mathcal{SD}(n,\ [[0,1,2,3]][0,1,2,3],\ \gamma) \\
&= \{\mathrm{n}\colon[0,1,2,3]\} \\
\mathcal{SD}(e,\ [0,1,2,3],\ \gamma)\setminus\{iv\} &= \{\mathrm{iv}\colon[0,2,2,3],\ \mathrm{ofs}\colon[0,2,2,3],\ \mathrm{arr}\colon[0,1,2,3]\}\setminus\{iv\} \\
&= \{\mathrm{ofs}\colon[0,2,2,3],\ \mathrm{arr}\colon[0,1,2,3]\}
\end{aligned}$$

We have now found that the demand environment of the with-expression is $\{\mathrm{n}\colon[0,1,2,3]\} \oplus \{\mathrm{ofs}\colon[0,2,2,3],\ \mathrm{arr}\colon[0,1,2,3]\} = \{\mathrm{n}\colon[0,1,2,3],\ \mathrm{ofs}\colon[0,2,2,3],\ \mathrm{arr}\colon[0,1,2,3]\}$. Now let us move another step up and take a look at the let-expression that encapsulates the with-expression we just solved.

$$\begin{aligned}
\mathcal{SD}(\mathrm{let}\ \mathit{ofs} = e_{\mathit{cond}}\,\mathrm{in}\,e_{\mathit{with}},\ [0,1,2,3],\ \gamma) &= \mathcal{SD}(e_{\mathit{cond}},\ \mathit{dem}_{\mathit{ofs}},\ \gamma) \\
&\quad \oplus (\mathcal{SD}(e_{\mathit{with}},\ [0,1,2,3],\ \gamma)\setminus\{\mathit{ofs}\}) \\
&\text{where } \mathit{dem}_{\mathit{ofs}} = \mathcal{PV}(\lambda \mathit{ofs}.\, e_{\mathit{with}},\ \gamma)\,[0][0,1,2,3]
\end{aligned}$$

Before we can find the demand environment of our conditional expression, we need to know what demand its resulting value, `ofs`, will have in the with-expression. This is easy to find, as we have already found the demand environment of the with-expression.

$$\begin{aligned}
\mathcal{PV}(\lambda \mathit{ofs}.\, e_{\mathit{with}},\ \gamma) &= [\mathcal{SD}(e_{\mathit{with}},\ [0,1,2,3],\ \gamma)\,[\mathit{ofs}]] \\
&= [\{\mathrm{ofs}\colon[0,2,2,3],\ \mathrm{arr}\colon[0,1,2,3]\}[\mathit{ofs}]] \\
&= [[0,2,2,3]]
\end{aligned}$$

Now, using this newly found demand, let us find the demand environment of the conditional expression. We have looked at quite a few examples already, so I will gloss over this one. As an exercise you could try to solve this one yourself.

$$\begin{aligned}
\mathcal{SD}(\mathrm{if}\ n > 0\,\mathrm{then}\,[0]\,\mathrm{else}\,e_f,\ [0,2,2,3],\ \gamma) &= \mathcal{SD}(n > 0,\ [0,3,3,3],\ \gamma) \\
&\quad \oplus \mathcal{SD}([0],\ [0,2,2,3],\ \gamma) \\
&\quad \oplus \mathcal{SD}(e_f,\ [0,2,2,3],\ \gamma) \\
&= \{\mathrm{n}\colon[0,3,3,3]\} \\
&\quad \oplus \{\mathrm{n}\colon[0,2,2,3],\ \mathrm{arr}\colon[0,0,1,2]\} \\
&= \{\mathrm{n}\colon[0,3,3,3],\ \mathrm{arr}\colon[0,0,1,2]\}
\end{aligned}$$

Using what we now know we can solve the let-expression.

$$\begin{aligned}\mathcal{SD}(\text{let } \mathit{ofs} = e_{cond} \text{ in } e_{with},\ [0,1,2,3],\ \gamma) &= \mathcal{SD}(e_{cond},\ [0,2,2,3],\ \gamma)\\ &\quad \oplus (\mathcal{SD}(e_{with},\ [0,1,2,3],\ \gamma) \setminus \{\mathit{ofs}\})\\ &= \{\text{n}: [0,3,3,3],\ \text{arr}: [0,0,1,2]\}\\ &\quad \oplus \{\text{n}: [0,1,2,3],\ \text{arr}: [0,1,2,3]\}\\ &= \{\text{n}: [0,3,3,3],\ \text{arr}: [0,1,2,3]\}\end{aligned}$$

Finally we can use this to find the propagation vector of the lambda expression using the calculation from the start.

$$\begin{aligned}\mathcal{PV}(\lambda n.\, \lambda arr.\, e_{body},\ \gamma) &= [\mathit{env}[n],\ \mathit{env}[arr]]\\ &\quad \text{where } \mathit{env} = \{\text{n}: [0,3,3,3],\ \text{arr}: [0,1,2,3]\}\\ &= [[0,3,3,3],[0,1,2,3]]\end{aligned}$$

Which shows us that the propagation vector of the user-defined function `take n arr` is $[[0,3,3,3],[0,1,2,3]]$. If we now, for example, want the shape of the result of this function, we see that the demand of `n` is 3 and the demand of `arr` is 2. This tells us that to get the shape of the `take` function we must have the full value of `n` and only the shape of `arr`.

5 Rewrite

The symbols $\mathcal{F}$, $\mathcal{S}$, $\mathcal{D}$, and $\mathcal{N}$ explain the different levels of information of a value. They correspond to the demand values 3, 2, 1, and 0 from the propagation vectors respectively.

The recursive functions $\mathcal{F}(expr,\, \varepsilon)$, $\mathcal{S}(expr,\, \varepsilon)$, $\mathcal{D}(expr,\, \varepsilon)$, and $\mathcal{N}(expr,\, \varepsilon)$ define the rewrite rules of the expression for the corresponding demand values. The rewrite rule $\mathcal{N}(expr,\, \varepsilon)$ always returns 0. Here the environment ε indicates to which level the corresponding variable at runtime will have been evaluated at that point.

5.1 Rules

We will start with the cases for values and variables. The *value* rule is very straightforward, we just rewrite the value to the required level. In the case of `VarId`, we first check if the variable has already been rewritten previously, and then only rewrite it again if necessary. For example, in the $\mathcal{S}$ case, if the variable is currently still in its full version, we take the shape of that variable. But when the variable has already been rewritten to the shape level, no more rewrites are necessary and we thus leave it as is.

$$\begin{aligned}
\mathcal{F}(value,\, \varepsilon) &= value \\
\mathcal{S}(value,\, \varepsilon) &= \text{shape } value \\
\mathcal{D}(value,\, \varepsilon) &= \text{dim } value
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}(\texttt{VarId},\, \varepsilon) &= \texttt{VarId} \\
\mathcal{S}(\texttt{VarId},\, \varepsilon) &= \begin{cases} \text{shape } \texttt{VarId} & \text{if } \varepsilon[\texttt{VarId}] = \mathcal{F} \\ \texttt{VarId} & \text{if } \varepsilon[\texttt{VarId}] = \mathcal{S} \end{cases} \\
\mathcal{D}(\texttt{VarId},\, \varepsilon) &= \begin{cases} \text{dim } \texttt{VarId} & \text{if } \varepsilon[\texttt{VarId}] = \mathcal{F} \\ (\text{shape } \texttt{VarId})[0] & \text{if } \varepsilon[\texttt{VarId}] = \mathcal{S} \\ \texttt{VarId} & \text{if } \varepsilon[\texttt{VarId}] = \mathcal{D} \end{cases}
\end{aligned}$$

Next up we will look at lambda-expressions and -applications. Here we see how the variable from the base case `VarId` can be rewritten before being evaluated. First we get the demand of the free variable by looking at its propagation vector. We then rewrite the variable to that level. This rewritten variable is then passed to the rewritten lambda function, where the environment is updated with the new level of the free variable to avoid incorrectly reducing the level multiple times.

$$\mathcal{F}(\lambda x.\, e,\, \varepsilon) = \begin{cases} \lambda x.\, \mathcal{F}(e,\, \varepsilon\{\text{x}\to\mathcal{X}\}) & \text{if } pv_x[3] = \mathcal{X} \in \{\mathcal{F}, \mathcal{S}, \mathcal{N}\} \\ \mathcal{F}(e,\, \varepsilon) & \text{if } pv_x[3] = \mathcal{N} \end{cases}$$
$$\mathcal{S}(\lambda x.\, e,\, \varepsilon) = \begin{cases} \lambda x.\, \mathcal{S}(e,\, \varepsilon\{\text{x}\to\mathcal{X}\}) & \text{if } pv_x[2] = \mathcal{X} \in \{\mathcal{F}, \mathcal{S}, \mathcal{N}\} \\ \mathcal{S}(e,\, \varepsilon) & \text{if } pv_x[2] = \mathcal{N} \end{cases}$$
$$\mathcal{D}(\lambda x.\, e,\, \varepsilon) = \begin{cases} \lambda x.\, \mathcal{D}(e,\, \varepsilon\{\text{x}\to\mathcal{X}\}) & \text{if } pv_x[1] = \mathcal{X} \in \{\mathcal{F}, \mathcal{S}, \mathcal{N}\} \\ \mathcal{D}(e,\, \varepsilon) & \text{if } pv_x[1] = \mathcal{N} \end{cases}$$
$$\text{where } pv_x = \mathcal{PV}(\lambda x.\, e,\, \gamma)\,[0]$$

$$\mathcal{F}((\lambda x.\, e)\ a,\, \varepsilon) = \begin{cases} (\lambda x.\, \mathcal{F}(e,\, \varepsilon\{\text{x}\to\mathcal{X}\}))\ \mathcal{X}(a,\, \varepsilon) & \text{if } pv_x[3] = \mathcal{X} \in \{\mathcal{F}, \mathcal{S}, \mathcal{N}\} \\ \mathcal{F}(e,\, \varepsilon) & \text{if } pv_x[3] = \mathcal{N} \end{cases}$$
$$\mathcal{S}((\lambda x.\, e)\ a,\, \varepsilon) = \begin{cases} (\lambda x.\, \mathcal{S}(e,\, \varepsilon\{\text{x}\to\mathcal{X}\}))\ \mathcal{X}(a,\, \varepsilon) & \text{if } pv_x[2] = \mathcal{X} \in \{\mathcal{F}, \mathcal{S}, \mathcal{N}\} \\ \mathcal{S}(e,\, \varepsilon) & \text{if } pv_x[2] = \mathcal{N} \end{cases}$$
$$\mathcal{D}((\lambda x.\, e)\ a,\, \varepsilon) = \begin{cases} (\lambda x.\, \mathcal{D}(e,\, \varepsilon\{\text{x}\to\mathcal{X}\}))\ \mathcal{X}(a,\, \varepsilon) & \text{if } pv_x[1] = \mathcal{X} \in \{\mathcal{F}, \mathcal{S}, \mathcal{N}\} \\ \mathcal{D}(e,\, \varepsilon) & \text{if } pv_x[1] = \mathcal{N} \end{cases}$$
$$\text{where } pv_x = \mathcal{PV}(\lambda x.\, e,\, \gamma)\,[0]$$

The let-expression works similarly to the lambda cases, where we find the best level to rewrite the variable x to and pass that on to the inner expression.

$$\mathcal{F}(\text{let}\, x = e_1 \,\text{in}\, e_2,\, \varepsilon) = \begin{cases} \text{let}\, x = \mathcal{X}(e_1,\, \varepsilon) \,\text{in}\, \mathcal{F}(e_2,\, \varepsilon\{\text{x}\to\mathcal{X}\}) & \text{if } pv_x[3] = \mathcal{X} \in \{\mathcal{F}, \mathcal{S}, \mathcal{N}\} \\ \mathcal{F}(e_2,\, \varepsilon) & \text{if } pv_x[3] = \mathcal{N} \end{cases}$$
$$\mathcal{S}(\text{let}\, x = e_1 \,\text{in}\, e_2,\, \varepsilon) = \begin{cases} \text{let}\, x = \mathcal{X}(e_1,\, \varepsilon) \,\text{in}\, \mathcal{S}(e_2,\, \varepsilon\{\text{x}\to\mathcal{X}\}) & \text{if } pv_x[2] = \mathcal{X} \in \{\mathcal{F}, \mathcal{S}, \mathcal{N}\} \\ \mathcal{S}(e_2,\, \varepsilon) & \text{if } pv_x[2] = \mathcal{N} \end{cases}$$
$$\mathcal{D}(\text{let}\, x = e_1 \,\text{in}\, e_2,\, \varepsilon) = \begin{cases} \text{let}\, x = \mathcal{X}(e_1,\, \varepsilon) \,\text{in}\, \mathcal{D}(e_2,\, \varepsilon\{\text{x}\to\mathcal{X}\}) & \text{if } pv_x[1] = \mathcal{X} \in \{\mathcal{F}, \mathcal{S}, \mathcal{N}\} \\ \mathcal{D}(e_2,\, \varepsilon) & \text{if } pv_x[1] = \mathcal{N} \end{cases}$$
$$\text{where } pv_x = \mathcal{PV}(\lambda x.\, e_2,\, \gamma)\,[0]$$

The conditional expression is also very simple. We always need the full rewrite of the condition, since even the dimensionality of the two branches can differ. Then we rewrite the two branches according to the requested rewrite level.

$$\mathcal{F}(\text{if}\ e_c\,\text{then}\ e_t\,\text{else}\ e_f,\, \varepsilon) = \text{if}\ \mathcal{F}(e_c,\, \varepsilon)\ \text{then}\ \mathcal{F}(e_t,\, \varepsilon)\ \text{else}\ \mathcal{F}(e_f,\, \varepsilon)$$
$$\mathcal{S}(\text{if}\ e_c\,\text{then}\ e_t\,\text{else}\ e_f,\, \varepsilon) = \text{if}\ \mathcal{F}(e_c,\, \varepsilon)\ \text{then}\ \mathcal{S}(e_t,\, \varepsilon)\ \text{else}\ \mathcal{S}(e_f,\, \varepsilon)$$
$$\mathcal{D}(\text{if}\ e_c\,\text{then}\ e_t\,\text{else}\ e_f,\, \varepsilon) = \text{if}\ \mathcal{F}(e_c,\, \varepsilon)\ \text{then}\ \mathcal{D}(e_t,\, \varepsilon)\ \text{else}\ \mathcal{D}(e_f,\, \varepsilon)$$

Since the with-expression we use expects the shape of the resulting value as an argument, the rewrites can simply use that value to find the shape and dimensionality. Making the rewrite of the with-expression very simple.

$$\begin{aligned}
\mathcal{F}\left(\begin{matrix}\text{gen } e_{gen}\ e_{def} \\ \text{with } e_l \leq x < e_u\end{matrix} \text{in}\, e,\ \varepsilon\right) &= \begin{matrix}\text{gen } \mathcal{F}(e_{gen},\ \varepsilon)\ \mathcal{F}(e_{def},\ \varepsilon) \\ \text{with } \mathcal{F}(e_l,\ \varepsilon) \leq x < \mathcal{F}(e_u,\ \varepsilon)\end{matrix} \text{in}\, \mathcal{F}(e,\ \varepsilon) \\
\mathcal{S}\left(\begin{matrix}\text{gen } e_{gen}\ e_{def} \\ \text{with } e_l \leq x < e_u\end{matrix} \text{in}\, e,\ \varepsilon\right) &= \mathcal{F}(e_{gen},\ \varepsilon) \\
\mathcal{D}\left(\begin{matrix}\text{gen } e_{gen}\ e_{def} \\ \text{with } e_l \leq x < e_u\end{matrix} \text{in}\, e,\ \varepsilon\right) &= \mathcal{S}(e_{gen},\ \varepsilon)\,[0]
\end{aligned}$$

Next we have all primitive functions and operators. Selection is quite difficult as the dimensionality of the index vector can be smaller than that of the expression, because we allow selecting sub-arrays and not just individual scalars.

When taking the shape we select the first few values from the shape of the expression, depending on the dimensionality of the index vector. When taking the dimensionality we know that it must be the difference of the expression and index vector.

$$\begin{aligned}
\mathcal{F}(\text{sel } iv\, e,\ \varepsilon) &= \text{sel } \mathcal{F}(iv,\ \varepsilon)\ \mathcal{F}(e,\ \varepsilon) \\
\mathcal{S}(\text{sel } iv\, e,\ \varepsilon) &= \mathcal{S}(e,\ \varepsilon)\,[:\mathcal{D}(iv,\ \varepsilon)] \\
\mathcal{D}(\text{sel } iv\, e,\ \varepsilon) &= \mathcal{D}(e,\ \varepsilon) - \mathcal{D}(iv,\ \varepsilon)
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}(\text{shape } e,\ \varepsilon) &= \mathcal{S}(e,\ \varepsilon) \\
\mathcal{S}(\text{shape } e,\ \varepsilon) &= [\mathcal{D}(e,\ \varepsilon)] \\
\mathcal{D}(\text{shape } e,\ \varepsilon) &= 1
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}(\text{dim } e,\ \varepsilon) &= \mathcal{D}(e,\ \varepsilon) \\
\mathcal{S}(\text{dim } e,\ \varepsilon) &= [] \\
\mathcal{D}(\text{dim } e,\ \varepsilon) &= 0
\end{aligned}$$

And finally we get to binary and unary operations. These contain two separate cases for maths and equality operations. This is because we know that an equality operation will always return a scalar, specifically always 0 or 1, so we will always know exactly what the shape and dimensionality will be. Maths operations are applied element-wise, where the right hand argument might have been broadcast, so maths operations will not change the shape and dimensionality of the left hand argument.

$$\mathcal{F}(e_l \;\texttt{bop}\; e_r,\, \varepsilon) = \mathcal{F}(e_l,\, \varepsilon) \;\texttt{bop}\; \mathcal{F}(e_r,\, \varepsilon)$$

$$\mathcal{S}(e_l \;\texttt{bop}\; e_r,\, \varepsilon) = \begin{cases} \mathcal{S}(e_l,\, \varepsilon) & \text{if } \texttt{bop} \in \mathit{maths} \\ [] & \text{if } \texttt{bop} \in \mathit{equality} \end{cases}$$

$$\mathcal{D}(e_l \;\texttt{bop}\; e_r,\, \varepsilon) = \begin{cases} \mathcal{D}(e_l,\, \varepsilon) & \text{if } \texttt{bop} \in \mathit{maths} \\ 0 & \text{if } \texttt{bop} \in \mathit{equality} \end{cases}$$

$$\mathcal{F}(\texttt{uop}\; e,\, \varepsilon) = \texttt{uop}\; \mathcal{F}(e,\, \varepsilon)$$

$$\mathcal{S}(\texttt{uop}\; e,\, \varepsilon) = \begin{cases} \mathcal{S}(e,\, \varepsilon) & \text{if } \texttt{uop} \in \mathit{maths} \\ [] & \text{if } \texttt{uop} \in \mathit{equality} \end{cases}$$

$$\mathcal{D}(\texttt{uop}\; e,\, \varepsilon) = \begin{cases} \mathcal{D}(e,\, \varepsilon) & \text{if } \texttt{uop} \in \mathit{maths} \\ 0 & \text{if } \texttt{uop} \in \mathit{equality} \end{cases}$$

5.2 Example

Lets now take a look at an example of a rewrite. We will be extending the example `take n xs` from the inference. We will use this function to implement a function that shifts the values of an array left or right by some amount `n`.

This program is shown below, along with the pre-computed demands for each user-defined function, these are computed like we did in the inference section (4). The program creates a list ranging from 0 to 99 and shifts those values 20 indices to the right, placing zeros at the now empty indices.

```
# demand: [[0,3,3,3], [0,1,2,3]]
let take = \n.\arr.
    let ofs = if n > 0 then 0
        else (shape arr).([0]) + n
    in
    gen [|n|] 0 with [n * 0] <= iv < [|n|] in
        arr.(iv + ofs)
in

# demand: [[0,3,3,3], [0,1,2,3]]
let drop = \n.\arr.
    if n > 0 then
        take (n - (shape arr).([0])) arr
    else
        take ((shape arr).([0]) + n) arr
in
```

```
# demand: [[0,3,3,3], [0,1,2,3]]
let shift = \n.\arr.
    let pad = gen (shape (take n arr)) 0 in     # (1)
    let xs = drop (-n) arr in
    if n > 0 then pad ++ xs
        else xs ++ pad
in

let size = 20000 in
let arr = gen size 0
    with 0 <= iv < size in iv in
shift 5000 arr
```

As can be seen in the first line of the shift function (1), we compute the entire result of `take`, but then only use its shape. So instead of looking at rewriting the entire program we will look at how this line would be rewritten, as it is the most interesting.

Lets assume we want the full result of this expression and that the parameters have not been rewritten to a lower level, we would then have to apply the following rewrite rule to the with-expression:

$$\begin{aligned}\mathcal{F}(\text{gen}\,(\text{shape}\;(\mathit{take}\;n\;\mathit{arr}))\;0,\,\varepsilon) &= \text{gen}\,\mathcal{F}(\text{shape}\;(\mathit{take}\;n\;\mathit{arr}),\,\varepsilon)\;\mathcal{F}(0,\,\varepsilon)\\ &= \text{gen}\,\mathcal{F}(\text{shape}\;(\mathit{take}\;n\;\mathit{arr}),\,\varepsilon)\;0\end{aligned}$$

We know that if we want the full value of a shape, we can rewrite the inner expression to the shape level. We would officially use currying currying here, but we won't worry about those extra steps in this example and just combine them.

$$\begin{aligned}\mathcal{F}(\text{gen}\,(\text{shape}\;(\mathit{take}\;n\;\mathit{arr}))\;0,\,\varepsilon) &= \text{gen}\,\mathcal{S}((\mathit{take}\;n)\;\mathit{arr},\,\varepsilon)\;0\\ &= \text{gen}\,\mathcal{S}(\mathit{take}\;n\;\mathit{arr},\,\varepsilon)\;0\end{aligned}$$

Now we are going to need a new version of the `take` function, which we will call `take_s`. Usually this would already been done by this point but for readability we'll change the order up a bit. First we can the let-expression by removing the variable *ofs* (offset), this is possible because the demand of *ofs* in the with-expression is $\mathcal{N}$. The rewrite of the with-expression is also very simple as we only need its shape, which is given by the programmer.

$$\begin{aligned}\mathcal{S}(\text{let}\;\mathit{ofs} = e_{\mathit{cond}}\;\text{in}\;e_{\mathit{with}},\,\varepsilon) &= \mathcal{S}(e_{\mathit{with}},\,\varepsilon)\\ &= \mathcal{F}([|n]\!],\,\varepsilon)\\ &= [\mathcal{F}(|n|,\,\varepsilon)]\\ &= [|\mathcal{F}(n,\,\varepsilon)]\!]\\ &= [|n]\!]\end{aligned}$$

So our `take n arr` function is now a lot simpler.

$$\text{let } \mathit{take\_s} = \lambda n.\, \lambda \mathit{arr}.\, [\![n]\!]$$

We now get the application rule, but since `take` is a pointer to a lambda expression, we will point to the rewritten function of the required level by changing the name of this pointer. We also need to know to what level we need to rewrite the variables, for this we use the demand we have found earlier. Again, in the rules we would use currying, but here we will just do both parameters at once.

$$\begin{aligned}
\mathcal{F}(\text{gen}\,(\text{shape}\,(\mathit{take}\ n\ \mathit{arr}))\ 0,\ \varepsilon) &= \text{gen}\,(\mathit{take\_s}\ \mathcal{X}(n,\ \varepsilon)\ \mathcal{Y}(\mathit{arr},\ \varepsilon))\ 0 \\
&\quad \text{where } \mathcal{X} = \mathcal{PV}(\texttt{take},\ \gamma)\,[0][2] \\
&\quad\qquad = [[0,3,3,3],[0,1,2,3]][0][2] \\
&\quad\qquad = [0,3,3,3][2] \\
&\quad\qquad = 3 \\
&\quad \text{and } \mathcal{Y} = \mathcal{PV}(\texttt{take},\ \gamma)\,[1][2] \\
&\quad\qquad = [[0,3,3,3],[0,1,2,3]][1][2] \\
&\quad\qquad = [0,1,2,3][2] \\
&\quad\qquad = 2 \\
&= \text{gen}\,(\mathit{take\_s}\ \mathcal{F}(n,\ \varepsilon)\ \mathcal{S}(\mathit{arr},\ \varepsilon))\ 0
\end{aligned}$$

Which finally results in the program below. Especially if we expand `take_s` we see that this computation is a lot faster and skips a lot of steps that have become unnecessary when computing only the shape.

$$\begin{aligned}
\mathcal{F}(\text{gen}\,(\text{shape}\,(\mathit{take}\ n\ \mathit{arr}))\ 0,\ \varepsilon) &= \text{gen}\,(\mathit{take\_s}\ n\ (\text{shape}\ \mathit{arr}))\ 0 \\
&= \text{gen}\,((\lambda n.\, \lambda \mathit{arr}.\, [\![n]\!])\ n\ (\text{shape}\ \mathit{arr}))\ 0
\end{aligned}$$

6 Correctness

One important consideration is whether the rewritten program is semantically equivalent to the original program, e.g. whether for every input of the original program, the rewritten program produces the same result. It turns out that this is not always the case. Consider the very simple expression shape (3/0), which rewrites to:

$$\begin{aligned}\mathcal{F}(\text{shape } (3/0),\, \varepsilon) &= \mathcal{S}(3/0,\, \varepsilon)\\ &= \mathcal{S}(3,\, \varepsilon)\\ &= \text{shape } 3\\ &= [\,]\end{aligned}$$

In the original program the expression 3/0 would result in a zero-division error, thus not evaluating to a value. But the rewritten expression removes unnecessary computations, causing this rewritten program to evaluate to the value [ ] instead of an error.

To avoid this we will create a new semantics, we get this new semantics by applying our rewrite $\mathcal{F}$ to our current rewrite rules, defined in section 3.3. We would like this semantics to be an extension of the original semantics, where the set of valid programs and their evaluations for the original semantics are a subset of the set of valid programs of our new semantics, where these evaluated values are the same.

6.1 Proof

The property we just discussed can be defined formally as the following theorem.

$$\forall_{prg}\colon ((prg,\, \emptyset)\ \Downarrow val) \rightarrow ((\mathcal{F}(prg,\, \emptyset),\, \emptyset)\ \Downarrow val)$$

Which states that for every possible program; if that program, initially with the empty program environment, evaluates to a value val, then the full rewrite of that program must also evaluate to the value val.

To prove this theorem we need to show something similar for the shape and dimensionality cases. But since these cases change the rewrite level of variables, we will need to use that rewrite level environment ε to also update the program environment.

To do so we will write $\varepsilon.\sigma$, which takes keys x from the program environment σ and updates their values v into (shape v) if $\varepsilon[x] = \mathcal{S}$, and into (dim v) if $\varepsilon[x] = \mathcal{D}$.

For example lets take a program environment that contains a value v for the variable x, then: $\sigma = \{\mathrm{x}\colon v\}$, and a rewrite level environment where x has been rewritten to level $\mathcal{S}$, then: $\varepsilon = \{\mathrm{x}\colon \mathcal{S}\}$. Then $\varepsilon.\sigma$ takes key x and finds that its rewrite level is $\mathcal{S}$, which means that the new program environment σ' becomes $\{\mathrm{x}\colon \mathcal{S}(v)\} = \{\mathrm{x}\colon \text{shape } v\}$.

Using this we can now create a lemma that includes cases for shape and dimensionality. For those cases we update their program environments. We will now also look at separate expressions, and not the entire program. So here we have the environments σ and ε instead of empty sets.

$$\begin{aligned}\forall_{expr}\colon ((expr,\, \sigma) \Downarrow val) \rightarrow\; & ((\mathcal{F}(expr,\, \varepsilon),\, \sigma) \Downarrow val)\\ \wedge\; & ((\mathcal{S}(expr,\, \varepsilon),\, \varepsilon.\sigma) \Downarrow \text{shape } val)\\ \wedge\; & ((\mathcal{D}(expr,\, \varepsilon),\, \varepsilon.\sigma) \Downarrow \text{dim } val)\end{aligned}$$

Here we can not take any rewrite level environment ε, but luckily we know all possible states for this environment. We find these by first applying our $\mathcal{SD}$ rule to the expression, which gives us an environment mapping the free variables of the expression to their corresponding demands. Then for each of these demands we take one specific index, depending on the rewrite rule this environment is in. These being 3 for $\mathcal{F}$, 2 for $\mathcal{S}$, and 1 for $\mathcal{D}$. The environment ε is then a combination of all possible assignments of variables to rewrite levels, where this rewrite level is at least the one we just found.

For example, lets say that applying $\mathcal{SD}(expr,\, [0,1,2,3],\, \gamma)$ resulted to the environment $\{\mathrm{x}\colon [0,2,2,3],\ \mathrm{y}\colon 0,1,2,3\}$. Then in the case of $\mathcal{S}(expr,\, \varepsilon)$ we take index 2 of both `x` and `y`, giving us: $\{\mathrm{x}\colon 2,\ \mathrm{y}\colon 2\}$. Now both `x` and `y` must be at least $\mathcal{S}$. Getting all possible combinations with at least this demand then gets us the following possible environments: $\varepsilon \in \{\, \{\mathrm{x}\colon\mathcal{S}, \mathrm{y}\colon\mathcal{S}\},\ \{\mathrm{x}\colon\mathcal{S}, \mathrm{y}\colon\mathcal{F}\},\ \{\mathrm{x}\colon\mathcal{F}, \mathrm{y}\colon\mathcal{S}\},\ \{\mathrm{x}\colon\mathcal{F}, \mathrm{y}\colon\mathcal{F}\}\, \}$.

6.2 Induction

We will show using structural induction that this theorem holds, which will prove that that these rewrite rules do not change the result of a valid expression with a valid input. In some cases we will be able to prove this by showing that the rewrite evaluates to the same expression. In other cases this will not be possible, in which case we will show that the rewrite evaluates to the same value as the original program.

6.2.1 Base cases

Our base cases are two non-recursive expressions *value* and `VarId`. The case *value* trivially holds, as it just gets the data, shape, or dimensionality of the value as requested by the rewrite.

Next we have the `VarId` case, where we will take a variable x with corresponding value v from the program environment. We have to show that the rewrite rules hold for different cases of the current rewrite level of x. We do this by showing that the expressions evaluate to the correct values, these being: v for $\mathcal{F}$, (shape v) for $\mathcal{S}$, and (dim v) for $\mathcal{D}$. In this case we will also see why our choice for possible ε's is correct.

The $\mathcal{F}$ case is easy, we can always simply lookup the value of variable x in the program environment without any extra work. So we will take this opportunity to show how these prove rules will be formulated.

$$\begin{aligned}
(\mathcal{F}(x,\, \varepsilon),\, \sigma) &= (x,\, \sigma) \\
&\Downarrow \sigma[x] \\
&= v
\end{aligned}$$

In the first line we use our definition of the rewrite rules from section 5 to rewrite the expression, here with level $\mathcal{F}$. This then gives us an expression we can evaluate. In the second line we then use our semantic rules from section 3.3 to get the evaluated value from this expression.

For the shape and dimensionality cases we need to make use of the $\varepsilon.\sigma$ rule defined above to update the program environment before we can evaluate these expressions. This new environment can then be used to find the (potentially rewritten) value of variable x, which we can then use to evaluate the expression using our semantics.

$$\begin{aligned}
(\mathcal{S}(x,\, \varepsilon),\, \varepsilon.\sigma) &= (\text{shape } x,\, \varepsilon.\sigma) && \text{if } \varepsilon[x] = \mathcal{F} \\
&= (\text{shape } x,\, \sigma\{\text{x}{\rightarrow}\mathcal{F}(v,\, \varepsilon)\}) \\
&= (\text{shape } x,\, \sigma\{\text{x}{\rightarrow}v\}) \\
&\Downarrow \text{shape } (\sigma'[x]) \\
&= \text{shape } v \\
(\mathcal{S}(x,\, \varepsilon),\, \varepsilon.\sigma) &= (x,\, \varepsilon.\sigma) && \text{if } \varepsilon[x] = \mathcal{S} \\
&= (x,\, \sigma\{\text{x}{\rightarrow}\mathcal{S}(v,\, \varepsilon)\}) \\
&= (x,\, \sigma\{\text{x}{\rightarrow}\text{shape } v\}) \\
&\Downarrow \sigma'[x] \\
&= \text{shape } v
\end{aligned}$$

For both cases, in the second line we find this new environment by mapping our variable `x` to either $\mathcal{F}(v,\, \varepsilon)$ or $\mathcal{S}(e,\, \varepsilon)$, depending on `x`'s current rewrite level. In the third line we can then evaluate this expression using our rewrite rules, after which we can use this new environment to lookup the value of the variable `x`.

We apply the same steps to the $\mathcal{D}$ cases.

$$\begin{aligned}
(\mathcal{D}(x,\, \varepsilon),\, \varepsilon.\sigma) &= (\text{dim } x,\, \varepsilon.\sigma) && \text{if } \varepsilon[x] = \mathcal{F}\\
&= (\text{dim } x,\, \sigma\{\text{x}{\to}\mathcal{F}(v,\, \varepsilon)\})\\
&= (\text{dim } x,\, \sigma\{\text{x}{\to}v\})\\
&\Downarrow \text{dim } (\sigma'[x])\\
&= \text{dim } v\\
(\mathcal{D}(x,\, \varepsilon),\, \varepsilon.\sigma) &= ((\text{shape } x)[0],\, \varepsilon.\sigma) && \text{if } \varepsilon[x] = \mathcal{S}\\
&= ((\text{shape } x)[0],\, \sigma\{\text{x}{\to}\mathcal{S}(v,\, \varepsilon)\})\\
&= ((\text{shape } x)[0],\, \sigma\{\text{x}{\to}\text{shape } v\})\\
&\Downarrow (\text{shape } (\sigma'[x]))[0]\\
&= (\text{shape } (\text{shape } v))[0]\\
&= \text{dim } v\\
(\mathcal{D}(x,\, \varepsilon),\, \varepsilon.\sigma) &= (x,\, \varepsilon.\sigma) && \text{if } \varepsilon[x] = \mathcal{D}\\
&= (x,\, \sigma\{\text{x}{\to}\mathcal{D}(v,\, \varepsilon)\})\\
&= (x,\, \sigma\{\text{x}{\to}\text{dim } v\})\\
&\Downarrow \sigma'[x]\\
&= \text{dim } v
\end{aligned}$$

In the case of $\mathcal{F}$ the expression evaluates to just v. For $\mathcal{S}$ it evaluates to shape v, and for $\mathcal{D}$ it evaluates to dim v. This is exactly what we want to show, so this case holds.

6.2.2 Inductive cases

For the inductive cases we use the lemma defined above as our induction hypothesis. Lets start simple and begin with binary and unary operations. First we take a look at only the mathematical operations. Since we know that mathematical operations do their operations element-wise, potentially using broadcasting on the right hand side if it is a scalar; it follows that the result has the same shape as the array on the left. Because of this potential broadcasting we must take the left hand side and not the right hand side.

$$\begin{aligned}
(\mathcal{F}(e_1 \texttt{ bop } e_2,\, \varepsilon),\, \sigma) &= (\mathcal{F}(e_1,\, \varepsilon) \texttt{ bop } \mathcal{F}(e_2,\, \varepsilon),\, \sigma)\\
&\Downarrow e_1 \texttt{ bop } e_2\\
(\mathcal{S}(e_1 \texttt{ bop } e_2,\, \varepsilon),\, \sigma) &= (\mathcal{S}(e_1,\, \varepsilon),\, \sigma)\\
&\Downarrow \text{shape } e_1\\
&= \text{shape } (e_1 \texttt{ bop } e_2)\\
(\mathcal{D}(e_1 \texttt{ bop } e_2,\, \varepsilon),\, \sigma) &= (\mathcal{S}(e_1,\, \varepsilon),\, \sigma)\\
&\Downarrow \text{dim } e_1\\
&= \text{dim } (e_1 \texttt{ bop } e_2)
\end{aligned}$$

$$\begin{aligned}
(\mathcal{F}(\texttt{uop}\ e,\ \varepsilon),\ \sigma) &= (\texttt{uop}\ \mathcal{F}(e,\ \varepsilon),\ \sigma)\\
&\Downarrow \texttt{uop}\ e\\
(\mathcal{S}(\texttt{uop}\ e,\ \varepsilon),\ \sigma) &= (\texttt{uop}\ \mathcal{S}(e,\ \varepsilon),\ \sigma)\\
&\Downarrow \texttt{uop}\ (\text{shape}\ e)\\
&= \text{shape}\ (\texttt{uop}\ e)\\
(\mathcal{D}(\texttt{uop}\ e,\ \varepsilon),\ \sigma) &= (\texttt{uop}\ \mathcal{D}(e,\ \varepsilon),\ \sigma)\\
&\Downarrow \texttt{uop}\ (\text{dim}\ e)\\
&= \text{dim}\ (\texttt{uop}\ e)
\end{aligned}$$

Next come the equality operations, which always return the same default value for the shape and dimensionality cases. Here we make use of the fact that equality operators always return a scalar, namely 0 for false or 1 for true cases.

$$\begin{aligned}
(\mathcal{F}(e_l\ \texttt{bop}\ e_r,\ \varepsilon),\ \sigma) &= (\mathcal{F}(e_l,\ \varepsilon)\ \texttt{bop}\ \mathcal{F}(e_r,\ \varepsilon),\ \sigma)\\
&\Downarrow e_l\ \texttt{bop}\ e_r\\
(\mathcal{S}(e_l\ \texttt{bop}\ e_r,\ \varepsilon),\ \sigma) &\Downarrow [\,]\\
&= \text{shape}\ \textit{scalar}\\
&= \text{shape}\ (e_l\ \texttt{bop}\ e_r)\\
(\mathcal{D}(e_l\ \texttt{bop}\ e_r,\ \varepsilon),\ \sigma) &\Downarrow 0\\
&= \text{dim}\ \textit{scalar}\\
&= \text{dim}\ (e_l\ \texttt{bop}\ e_r)
\end{aligned}$$

$$\begin{aligned}
(\mathcal{F}(\texttt{uop}\ e,\ \varepsilon),\ \sigma) &= (\texttt{uop}\ \mathcal{F}(e,\ \varepsilon),\ \sigma)\\
&\Downarrow \texttt{uop}\ e\\
(\mathcal{S}(\texttt{uop}\ e,\ \varepsilon),\ \sigma) &\Downarrow [\,]\\
&= \text{shape}\ \textit{scalar}\\
&= \text{shape}\ (\texttt{uop}\ e)\\
(\mathcal{D}(\texttt{uop}\ e,\ \varepsilon),\ \sigma) &\Downarrow 0\\
&= \text{dim}\ \textit{scalar}\\
&= \text{dim}\ (\texttt{uop}\ e)
\end{aligned}$$

Now we will take a look at the primitive functions. The shape and dimensionality cases are not very hard, though they use some common guaranteed information we get from these operations. We know that the shape always is a 1-dimensional array (which we will call a list), from which it follows that the shape of a shape is the shape of a list, and thus is a scalar.

$$
\begin{aligned}
(\mathcal{F}(\text{shape } e,\, \varepsilon),\, \sigma) &= (\mathcal{S}(e,\, \varepsilon),\, \sigma) \\
&\Downarrow \text{shape } e \\
(\mathcal{S}(\text{shape } e,\, \varepsilon),\, \sigma) &= ([\mathcal{D}(e,\, \varepsilon)],\, \sigma) \\
&\Downarrow [\text{dim } e] \\
&= \text{shape (shape } e) \\
(\mathcal{D}(\text{shape } e,\, \varepsilon),\, \sigma) &\Downarrow 1 \\
&= \text{dim } \mathit{list} \\
&= \text{dim (shape } e)
\end{aligned}
$$

$$
\begin{aligned}
(\mathcal{F}(\text{dim } e,\, \varepsilon),\, \sigma) &= (\mathcal{D}(e,\, \varepsilon),\, \sigma) \\
&\Downarrow \text{dim } e \\
(\mathcal{S}(\text{dim } e,\, \varepsilon),\, \sigma) &\Downarrow [\,] \\
&= \text{shape } \mathit{scalar} \\
&= \text{shape (dim } e) \\
(\mathcal{D}(\text{dim } e,\, \varepsilon),\, \sigma) &\Downarrow 0 \\
&= \text{dim } \mathit{scalar} \\
&= \text{dim (dim } e)
\end{aligned}
$$

Our third primitive function is selection. The full rewrite of selection is very simple, but the other two cases are quite complicated.

In the shape case we know that the dimensionality of *iv* is a scalar, whose length is shorter or equal to the length of the shape of *e*, so in the rule after the induction hypothesis we take the first few values from this shape. But we can now also do it the other way around by first selecting the correct values and then taking the shape, which then gives us exactly the rule we want.

In the dimensionality case, we can move the dimensionality operations outward to combine them in one. To do this we must also somehow subtract *iv* from *e*, since we are going to only take its dimensionality we can do this be selecting *iv* in *e*.

$$
\begin{aligned}
(\mathcal{F}(\text{sel } iv\, e,\, \varepsilon),\, \sigma) &= (\text{sel } \mathcal{F}(iv,\, \varepsilon)\; \mathcal{F}(e,\, \varepsilon),\, \sigma) \\
&\Downarrow \text{sel } iv\, e \\
(\mathcal{S}(\text{sel } iv\, e,\, \varepsilon),\, \sigma) &= (\mathcal{S}(e,\, \varepsilon)[:\mathcal{D}(iv,\, \varepsilon)],\, \sigma) \\
&\Downarrow (\text{shape } e)[:\text{dim } iv] \\
&= \text{shape (sel } iv\, e) \\
(\mathcal{D}(\text{sel } iv\, e,\, \varepsilon),\, \sigma) &= (\mathcal{D}(e,\, \varepsilon) - \mathcal{D}(iv,\, \varepsilon),\, \sigma) \\
&\Downarrow \text{dim } e - \text{dim } iv \\
&= \text{dim (sel } iv\, e)
\end{aligned}
$$

The conditional expression is fairly straightforward, here we can move the shape or demand expression outside of the conditional if both branches apply it.

$$
\begin{aligned}
\big(\mathcal{F}(\text{if } e_c \text{ then } e_t \text{ else } e_f,\, \varepsilon),\, \sigma\big) &= \big(\text{if } \mathcal{F}(e_c,\, \varepsilon) \text{ then } \mathcal{F}(e_t,\, \varepsilon) \text{ else } \mathcal{F}(e_f,\, \varepsilon),\, \sigma\big)\\
&\Downarrow \text{if } e_c \text{ then } e_t \text{ else } e_f\\
\big(\mathcal{S}(\text{if } e_c \text{ then } e_t \text{ else } e_f,\, \varepsilon),\, \sigma\big) &= \big(\text{if } \mathcal{F}(e_c,\, \varepsilon) \text{ then } \mathcal{S}(e_t,\, \varepsilon) \text{ else } \mathcal{S}(e_f,\, \varepsilon),\, \sigma\big)\\
&\Downarrow \text{if } e_c \text{ then (shape } e_t) \text{ else (shape } e_f)\\
&= \text{shape (if } e_c \text{ then } e_t \text{ else } e_f)\\
\big(\mathcal{D}(\text{if } e_c \text{ then } e_t \text{ else } e_f,\, \varepsilon),\, \sigma\big) &= \big(\text{if } \mathcal{F}(e_c,\, \varepsilon) \text{ then } \mathcal{D}(e_t,\, \varepsilon) \text{ else } \mathcal{D}(e_f,\, \varepsilon),\, \sigma\big)\\
&\Downarrow \text{if } e_c \text{ then (dim } e_t) \text{ else (dim } e_f)\\
&= \text{dim (if } e_c \text{ then } e_t \text{ else } e_f)
\end{aligned}
$$

The next expression is the with-expression. Here we require the programmer to give us the resulting shape with e_{gen}. The $\mathcal{F}$ case is trivial, in the other two cases we can simply use this e_gen inside of a new with-expression of which we take the shape of demand, which will discard the computed values.

$$
\begin{aligned}
\left(\mathcal{F}\left(\begin{matrix}\text{gen } e_{gen}\ e_{def}\\ \text{with } e_l \le x < e_u\end{matrix} \text{ in } e,\, \varepsilon\right),\, \sigma\right) &= \left(\begin{matrix}\text{gen } \mathcal{F}(e_{gen},\, \varepsilon)\ \mathcal{F}(e_{def},\, \varepsilon)\\ \text{with } \mathcal{F}(e_l,\, \varepsilon) \le x < \mathcal{F}(e_u,\, \varepsilon)\end{matrix} \text{ in } \mathcal{F}(e,\, \varepsilon),\, \sigma\right)\\
&\Downarrow \begin{matrix}\text{gen } e_{gen}\ e_{def}\\ \text{with } e_l \le x < e_u\end{matrix} \text{ in } e\\
\left(\mathcal{S}\left(\begin{matrix}\text{gen } e_{gen}\ e_{def}\\ \text{with } e_l \le x < e_u\end{matrix} \text{ in } e,\, \varepsilon\right),\, \sigma\right) &= \big(\mathcal{F}(e_{gen},\, \varepsilon),\, \sigma\big)\\
&\Downarrow e_{gen}\\
&= \text{shape } \left(\begin{matrix}\text{gen } e_{gen}\ e_{def}\\ \text{with } e_l \le x < e_u\end{matrix} \text{ in } e\right)\\
\left(\mathcal{D}\left(\begin{matrix}\text{gen } e_{gen}\ e_{def}\\ \text{with } e_l \le x < e_u\end{matrix} \text{ in } e,\, \varepsilon\right),\, \sigma\right) &= \big(\mathcal{S}(e_{gen},\, \varepsilon),\, \sigma\big)\\
&\Downarrow (\text{shape } e_{gen})[0]\\
&= \text{dim } \left(\begin{matrix}\text{gen } e_{gen}\ e_{def}\\ \text{with } e_l \le x < e_u\end{matrix} \text{ in } e\right)
\end{aligned}
$$

Next we get the let-expression. From our semantics we know that; if the let-expression evaluates to a value v_2, then there exists an evaluation v_1 of e_1 such that expression e_2 with x mapped to v_1 evaluates to v_2.

$$((\mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2,\, \sigma) \Downarrow v_2) \rightarrow \exists_{v_1}\colon ((e_1,\, \sigma) \Downarrow v_1) \wedge ((e_2,\, \sigma\{\mathrm{x}\rightarrow v_1\}) \Downarrow v_2)$$

Like with `VarId`, we have separate cases within each rewrite rule, depending on the demand of the variable x; dem_x, which we find by computing $\mathcal{PV}(\lambda x.\, e_2,\, \gamma)[0]$. Here these cases overlap nicely, so we can combine them in a single case where the demand of x is $\mathcal{X}$.

if $dem_x[3] = \mathcal{X}$:

$$\begin{aligned}
(\mathcal{F}(\mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2,\, \varepsilon),\, \sigma) &= (\mathrm{let}\, x = \mathcal{X}(e_1,\, \varepsilon) \,\mathrm{in}\, \mathcal{F}(e_2,\, \varepsilon),\, \sigma) \\
&\Downarrow \mathrm{let}\, x = \mathcal{X}(v_1) \,\mathrm{in}\, (\mathcal{F}(e_2,\, \varepsilon\{\mathrm{x}\rightarrow\mathcal{X}\}),\, \varepsilon.\sigma) \\
&= \mathrm{let}\, x = \mathcal{X}(v_1) \,\mathrm{in}\, (\mathcal{F}(e_2,\, \varepsilon\{\mathrm{x}\rightarrow\mathcal{X}\}),\, \sigma\{\mathrm{x}\rightarrow\mathcal{X}(v_1)\}) \\
&= (\mathcal{F}(e_2,\, \varepsilon\{\mathrm{x}\rightarrow\mathcal{X}\}),\, \sigma\{\mathrm{x}\rightarrow\mathcal{X}(v_1)\}) \\
&\Downarrow v_2
\end{aligned}$$

if $dem_x[2] = \mathcal{X}$:

$$\begin{aligned}
(\mathcal{S}(\mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2,\, \varepsilon),\, \varepsilon.\sigma) &= (\mathrm{let}\, x = \mathcal{X}(e_1,\, \varepsilon) \,\mathrm{in}\, \mathcal{S}(e_2,\, \varepsilon),\, \sigma) \\
&\Downarrow \mathrm{let}\, x = \mathcal{X}(v_1) \,\mathrm{in}\, (\mathcal{S}(e_2,\, \varepsilon\{\mathrm{x}\rightarrow\mathcal{X}\}),\, \varepsilon.\sigma) \\
&= \mathrm{let}\, x = \mathcal{X}(v_1) \,\mathrm{in}\, (\mathcal{S}(e_2,\, \varepsilon\{\mathrm{x}\rightarrow\mathcal{X}\}),\, \sigma\{\mathrm{x}\rightarrow\mathcal{X}(v_1)\}) \\
&= (\mathcal{S}(e_2,\, \varepsilon\{\mathrm{x}\rightarrow\mathcal{X}\}),\, \sigma\{\mathrm{x}\rightarrow\mathcal{X}(v_1)\}) \\
&\Downarrow \mathrm{shape}\ v_2
\end{aligned}$$

if $dem_x[2] = \mathcal{X}$:

$$\begin{aligned}
(\mathcal{D}(\mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2,\, \varepsilon),\, \varepsilon.\sigma) &= (\mathrm{let}\, x = \mathcal{X}(e_1,\, \varepsilon) \,\mathrm{in}\, \mathcal{D}(e_2,\, \varepsilon),\, \sigma) \\
&\Downarrow \mathrm{let}\, x = \mathcal{X}(v_1) \,\mathrm{in}\, (\mathcal{D}(e_2,\, \varepsilon\{\mathrm{x}\rightarrow\mathcal{X}\}),\, \varepsilon.\sigma) \\
&= \mathrm{let}\, x = \mathcal{X}(v_1) \,\mathrm{in}\, (\mathcal{D}(e_2,\, \varepsilon\{\mathrm{x}\rightarrow\mathcal{X}\}),\, \sigma\{\mathrm{x}\rightarrow\mathcal{X}(v_1)\}) \\
&= (\mathcal{D}(e_2,\, \varepsilon\{\mathrm{x}\rightarrow\mathcal{X}\}),\, \sigma\{\mathrm{x}\rightarrow\mathcal{X}(v_1)\}) \\
&\Downarrow \mathrm{dim}\ v_2
\end{aligned}$$

And finally we have function application. From our semantics we know that if; the application-expression evaluates to a value v_e, then there exists an evaluation v_a of a such that expression e with x mapped to v_a evaluates to v_e.

$$(((\lambda x.\, e)\ a,\ \sigma)\ \Downarrow v_e) \rightarrow \exists_{v_a}\!: ((a,\ \sigma)\ \Downarrow v_a) \wedge ((e,\ \sigma\{\mathrm{x}{\rightarrow}v_a\})\ \Downarrow v_e)$$

We again have separate cases within each rewrite rule, depending on the demand of the variable x. Here we find this demand with: $dem_x = \mathcal{PV}(\lambda x.\, e,\ \gamma)\,[0]$. Again there is a lot of overlap within these cases, so like before we combine them using $\mathcal{X}$ for the demand we find for x.

$$\begin{aligned}
&\text{if } dem_x[3] = \mathcal{X}\text{:}\\
&(\mathcal{F}((\lambda x.\, e)\ a,\ \varepsilon),\ \sigma) = ((\lambda x.\, \mathcal{F}(e,\ \varepsilon\{\mathrm{x}{\rightarrow}X\}))\ \mathcal{X}(a,\ \varepsilon),\ \sigma)\\
&\qquad\Downarrow (\lambda x.\ (\mathcal{F}(e,\ \varepsilon\{\mathrm{x}{\rightarrow}X\}),\ \varepsilon.\sigma))\ \mathcal{X}(v_a)\\
&\qquad= (\lambda x.\ (\mathcal{F}(e,\ \varepsilon\{\mathrm{x}{\rightarrow}X\}),\ \sigma\{\mathrm{x}{\rightarrow}\mathcal{X}(v_a)\}))\ \mathcal{X}(v_a)\\
&\qquad= (\mathcal{F}(e,\ \varepsilon\{\mathrm{x}{\rightarrow}X\}),\ \sigma\{\mathrm{x}{\rightarrow}\mathcal{X}(v_a)\})\\
&\qquad\Downarrow v_e\\
&\text{if } dem_x[3] = \mathcal{X}\text{:}\\
&(\mathcal{S}((\lambda x.\, e)\ a,\ \varepsilon),\ \varepsilon.\sigma) = ((\lambda x.\, \mathcal{S}(e,\ \varepsilon\{\mathrm{x}{\rightarrow}X\}))\ \mathcal{X}(a,\ \varepsilon),\ \sigma)\\
&\qquad\Downarrow (\lambda x.\ (\mathcal{F}(e,\ \varepsilon\{\mathrm{x}{\rightarrow}X\}),\ \varepsilon.\sigma))\ \mathcal{X}(v_a)\\
&\qquad= (\lambda x.\ (\mathcal{F}(e,\ \varepsilon\{\mathrm{x}{\rightarrow}X\}),\ \sigma\{\mathrm{x}{\rightarrow}\mathcal{X}(v_a)\}))\ \mathcal{X}(v_a)\\
&\qquad= (\mathcal{S}(e,\ \varepsilon\{\mathrm{x}{\rightarrow}X\}),\ \sigma\{\mathrm{x}{\rightarrow}\mathcal{X}(v_a)\})\\
&\qquad\Downarrow \text{shape } v_e\\
&\text{if } dem_x[3] = \mathcal{X}\text{:}\\
&(\mathcal{D}((\lambda x.\, e)\ a,\ \varepsilon),\ \varepsilon.\sigma) = ((\lambda x.\, \mathcal{D}(e,\ \varepsilon\{\mathrm{x}{\rightarrow}X\}))\ \mathcal{X}(a,\ \varepsilon),\ \sigma)\\
&\qquad\Downarrow (\lambda x.\ (\mathcal{D}(e,\ \varepsilon\{\mathrm{x}{\rightarrow}X\}),\ \varepsilon.\sigma))\ \mathcal{X}(v_a)\\
&\qquad= (\lambda x.\ (\mathcal{D}(e,\ \varepsilon\{\mathrm{x}{\rightarrow}X\}),\ \sigma\{\mathrm{x}{\rightarrow}\mathcal{X}(v_a)\}))\ \mathcal{X}(v_a)\\
&\qquad= (\mathcal{D}(e,\ \varepsilon\{\mathrm{x}{\rightarrow}X\}),\ \sigma\{\mathrm{x}{\rightarrow}\mathcal{X}(v_a)\})\\
&\qquad\Downarrow \text{dim } v_e
\end{aligned}$$

We have now shown that the theorem holds for each of our expressions, which concludes this proof and shows that; for all valid programs prg in the original semantics, the rewrite $\mathcal{F}(prg,\ \emptyset)$ of this program is part of our new semantics and evaluates to the same value as the original program.

7 Implementation

Accompanying this paper is a prototypical implementation of the grammar and rules discussed above. The code is based on the languages `SaC`[6] and `heh`[3], and can be found on GitHub[1]. The language is implemented in the strict programming language `OCaml`[8], throughout this chapter we will see the benefits of choosing this language.

The language consists of three main stages; the parser, the rewrite, and the evaluator. In the parser the user's input is processed and transformed into a tree. The rewrite then applies the rules discussed in this paper to rewrite that program, which is then evaluated in the evaluator, producing a result and printing that to the screen.

$$\begin{tikzcd}
\text{input} \arrow[r] & \boxed{\text{Parser}} \arrow[r] & \boxed{\text{Rewrite}} \arrow[r] & \boxed{\text{Evaluator}} \arrow[r] & \text{output}
\end{tikzcd}$$

Figure 7.1: Program pipeline

7.1 Parser

The first step in processing our input is parsing the program. The parser transforms the text file that contains the program to a something that is easily readable by our compiler, we will do this using an abstract syntax tree. This tree consists of nodes of expressions, where each node has subtrees containing the sub-expressions. Leaf nodes of the abstract syntax tree must always be strings (variables) or values. Before parsing we must transform the input program such that it is represented as tokens. This is easily done in `OCaml` using its built-in lexer[7].

Lets look a look at our shift example again.

```
let shift = \n.\arr.
    let pad = gen (shape (take n arr)) 0 in
    let xs = drop (-n) arr in
    if n > 0 then pad ++ xs
        else xs ++ pad
in
```

[1]`https://github.com/JordyAaldering/Bachelor-Thesis`

7.1.1 Abstract Syntax Tree

The root of the abstract syntax tree is then the let expression that defines the function. The first child must be a string, so we get a leaf containing 'shift', the second child is then the lambda expression that describes this function which has children itself, and the final expression is the rest of our program, which we have omitted here. This produces the syntax tree shown in figure 7.2.

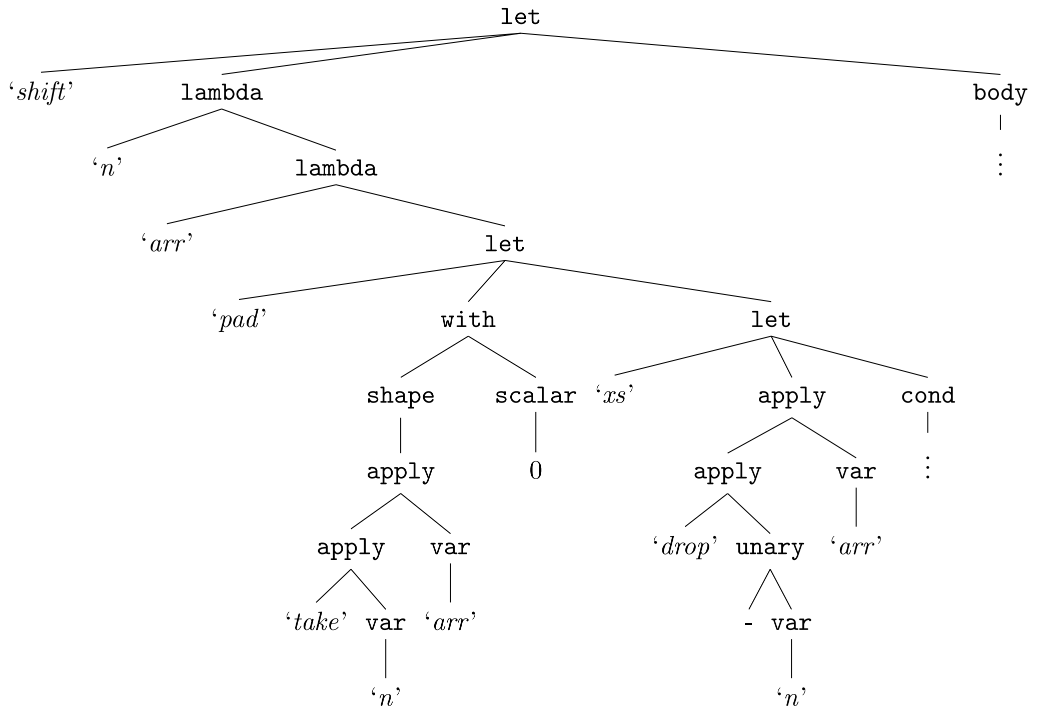


Figure 7.2: Abstract Syntax Tree

7.2 Rewrite

This syntax tree is then passed on to the rewrite, where the tree will be rewritten using our rewrite rules. Doing this is now very easy because our rules work very well on this syntax tree. Lets say we want to rewrite the outer let-expression.

$$\mathcal{F}(\text{let}\, x = e_1 \,\text{in}\, e_2,\, \varepsilon) \to \mathcal{F}(\text{let}\, shift = \texttt{lambda} \,\text{in}\, \texttt{body},\, \varepsilon)$$

We can then easily get the three arguments of the expression by getting the three child nodes and continuing the rewrite at each of those children. Implementing this in code is also very easy, as we can simply use pattern matching[9].

7.3 Evaluator

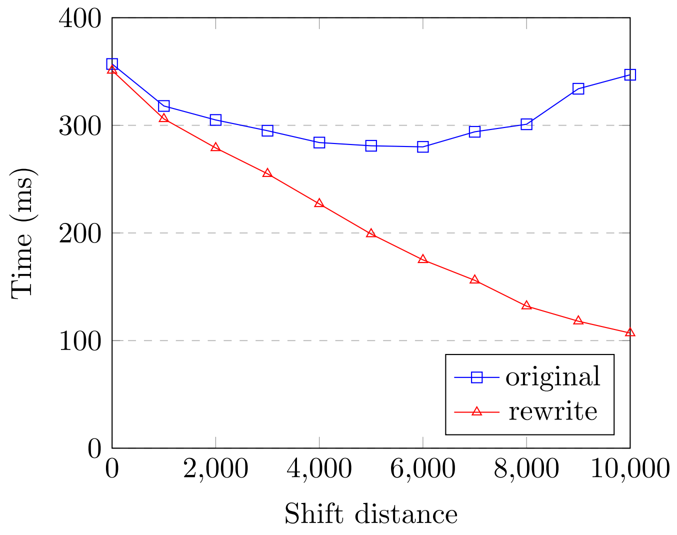
After rewriting the abstract syntax tree we pass it on to the evaluator. The evaluator passes over this tree to compute the resulting value. The evaluator also has to keep track of an environment that maps variables to their corresponding values or functions. For instance; when we encounter a let-expression we map the variable to the result of the first expression, and than add that to the environment of the second expression. Using pattern matching again we do certain operations and computations for each expression, which will eventually get us the result after having evaluated the entire tree.

8 Performance

There exist very many programs, some of which will profit greatly from our rewrite rules and some of which might not differ at all. This makes evaluating the performance of our rewrite rules in a general case very hard, so instead we will again look at the shift example discussed earlier and see how it fares, to get an idea of the possible performance improvements. This way we can still get an idea of the potential benefits of the rewrite rules, using a realistic example.

Figure 8.1 below shows the performance of both the original and rewritten program when shifting a list of length 20000 by some amount to the right.

Figure 8.1: Running time of shifting a list of length 20000



From our examples we know that most of the performance is gained by rewriting the `take n arr` expression to a version that only gets its shape. The farther we shift, the longer the original program will spend in the with-expression of this function. This is not necessary in our rewritten program, which is why we see greatly improved performance with an increasing shift distance.

9 Conclusion

In this paper we have seen how programs written in array programming languages can be rewritten in a way that finds a balance between strict and lazy evaluation. To do so we have created a small language along with rules which define how to rewrite programs written in this language by rewriting its expressions. This makes use of inference rules which find the required level of information for these expressions. Using a small example we have seen how to apply the knowledge provided by this paper, along with a quick look at the performance benefits of using these rewrite rules.

Along with the paper also comes a prototype, which implements the grammar and rules discussed in this paper. This prototype has shown that these rules can also be used in practise. Using our example program this prototype clearly shows the benefits of this rewrite by showing that the evaluation time can be greatly reduced.

9.1 Future work

Although this paper has clearly shown how these rewrite rules can be implemented to improve the performance of programs, the language discussed in this paper is not very extensive, making it hard to use it in actual applications. Instead, this paper has shown a new way of rewriting programs that, with further research, could be used for languages larger than the array programming language shown here. For example, it could be applied to the objects in an object oriented language using the techniques shown in this paper to remove unused properties and computations on these properties, decreasing the computational and memory load of these objects.

One big problem in the language discussed here is that these rewrite rules do not allow for higher order functions. Possible future research could look into extending the results from this paper by allowing rewrites on higher order functions to create a larger, more complete, array programming language.

10 Bibliography

[1] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. A binding scope analysis for generic programs on arrays. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages*, pages 212–230, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[2] Rogardt Heldal and John Hughes. Binding-time analysis for polymorphic types. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 191–204. Springer, 2001.

[3] Stephan Andreas Herhut. Auxiliary computations: a framework for a step-wise, non-disruptive introduction of static guarantees to untyped programs using partial evaluation techniques. *University of Hertfordshire*, 2010.

[4] Neil Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28:480–503, 09 1996.

[5] Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1 1993.

[6] Dietmar Kreye. A compiler backend for generic programming with arrays. *Faculty of Engineering at Christian-Albrechts-Universität*, 01 2004.

[7] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *Ocamllex and Ocamlyacc*, pages 193–211. Apress, Berkeley, CA, 2007.

[8] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 3.12. *Institut National de Recherche en Informatique et en Automatique*, 2011.

[9] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, page 35–46, New York, NY, USA, 2008. Association for Computing Machinery.

[10] Daniel D McCracken and Edwin D Reilly. Backus-naur form (bnf). In *Encyclopedia of Computer Science*, pages 129–131. John Wiley and Sons Ltd., Baffins Lane Chichester West Sussex PO19 1UD, United Kingdom, 2003.

[11] Sven-Bodo Scholz and Artjoms Šinkarovs. Tensor comprehensions in sac. In *Proceedings of International Symposium on Implementation and Application of Functional Languages*, pages 1–12, USA, New York, 2020. ACM.

[12] Kai Trojahner, Clemens Grelck, and Sven-Bodo Scholz. On optimising shape-generic array programs using symbolic structural information. In Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors, *Implementation and Application of Functional Languages*, pages 1–18, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.