Paper

Simon van Hus
6147879
s.vanhus@students.uu.nl

Abstract

Abstract.

1 Background

1.1 Functional Parallel Array Programming

A good starting point for functional parallel array programming was in 1992, with G. Belloch's paper on the parallel array programming language NESL [1]. The language was strongly-typed and had no support for side-effects, making it basically a functional language. The main way to add parallelism was through the inherently data-parallel "vectors" the language introduces in lieu of lists. These vectors could also be nested, and functions could run in nested parallel on these vectors. Another major inclusion was to allow user-defined functions to be run (in parallel) on these vectors, making it possible to write more complex nested data-parallel algorithms than before.

The functional language Haskell, saw the introduction of task-parallelism well before its first official release, through libraries like pH [20]. Some data-parallelism followed [12, 11, 7], but this was limited to applying a function over a flat array. However, in 2001 nested data-parallelism was introduced to Haskell by the NEPAL project by Chakravarty et al. [4]. The paper largely focusses on reimplementing NESL as a Haskell library, but creates a much more expressive data-parallel language doing so. This is because NESL was rather limited in scope, whereas Haskell was already a fully-fledged functional programming language. Two important concepts come to the forefront in the NEPAL paper, namely flattening and fusion. Both in NESL and in NEPAL, higher-dimensional nested parallelism is "flattened" to a single distributed parallel operation. In NESL, this meant that data-types had to be limited to tuples and the vectors it introduced, to make sure this flattening operation worked correctly. Since then however, Keller and Chakravarty had showed this flattening transformation could also be applied more generally [18, 15, 2]. This allowed them to apply the nested data parallelism of NESL to Haskell. Furthermore, they also showed that in combination with fusion it could produce efficient code for distributed machines [3]. Fusion is where multiple separate parallel operations are combined into a single parallel operation, which can greatly improve performance of complicated parallel programs. This is important because many

operations on arrays introduce the need for intermediate arrays to be computed. Doing this in parallel leads to more problems, as these implementations rely on gang parallelism, where the parallel threads remain in lockstep with each other [8]. Fusion helps us here, as we can reduce the number of intermediate arrays to be generated, as we can calculate the results of multiple operations at once [16, 6].

All this work culminated in 2007's Data Parallel Haskell (DPH) [22], by Peyton et al. Its main feature was the parallel array, that like NESL's vectors, were the main way of adding parallelism to a program. However, these parallel arrays could now hold any type, such as other arrays or functions, like Haskell's native (non-parallel) lists. Furthermore, DPH provides parallel variants of Haskell's native list functions, and a parallel alternative to Haskell's list comprehensions. The main difference between Haskell's native lists and DPH's parallel arrays (besides the parallelism) was that evaluating any value in a parallel array would require evaluation on all the array's elements, whereas Haskell as a lazy language would not normally do that. This is to be expected, as parallelism becomes meaningless if it is only applied to a single entry of an array.

Outside of Haskell, a functional array-programming dialect of C was developed: Single Assignment C (SAC) [23, 24, 9]. It would go on to distinguish itself as a functional array programming language in a style more familiar to programmers of imperative languages (like C). The main mechanic in SAC is the with-loop, which takes a generator that dictates a looping mechanism and an operation that dictates the return value. These operations can be functions like "fold" to reduce the rank of an array, or "genarray" to generate new (multidimensional) arrays. Besides the imperative style, the main draw of SAC is its performance comparable to Fortran and C, while its programs are generally more concise (for intensive numerical applications.)

In 2010, Keller, Chakravarty, et al. presented a new data-parallel approach for Haskell in "Regular, Shape-polymorphic, Parallel Arrays in Haskell" [17]. Previous approaches had focussed on irregular arrays, where on array could contain arrays of different lengths. The library Repa, introduced in this paper, was made for regular arrays where arrays of each nested rank are the same size. However, this allows the library to be purely functional, and allows support for shape polymorphism. In shape polymorphism, the type of a collection is fixed (unlike in type polymorphism), but the shape of the collection is not [14]. For instance, under shape polymorphism a function may be applied to either a flat array, or a 10-dimensional one. While shape polymorphism for functional arrays had been implemented before in SAC, Repa implemented it by embedding it into Haskell's type system, whereas the SAC implementation had required a purpose-built compiler. This also allowed programmers to more easily see and control the shapes of their multidimensional parallel arrays, and build their own shape polymorphic parallel functions.

In 2011, Repa was succeeded by the Accelerate project [5]. Accelerate is a library for Haskell, aimed specifically on bringing parallel array programming to modern GPUs. It mimicked many of Haskell's native list functions as parallel alternatives (run on the GPU), and used the typed shaped polymorphism from Repa. It also separated "collective" (array) computations and scalar computation by wrapping these in Haskell monads. Here, collective computations could include scalar computations, but not the other way around. This meant excluding nested and irregular data parallelism, which in turn allows Accelerate to efficiently run on GPUs (which are much more constrained than CPUs).

Another interesting example of a parallel array programming language is Remora by

Slepak et al. [25] The language, inspired by earlier array programming languages APL [13] and J implements rank-polymorphism. Rank polymorphism is similar to shape polymorphism, but it annotates functions and operators with a specific array rank they can operate on, and was also present in Repa and Accelerate. Remember that scalars are considered rank 0 arrays, a flat array is rank 1, a matrix is rank 2, et cetera. In rank polymorphism, arguments are transformed (re-ranked) such that they are the rank required for a specific function or operator. Specifically, an operator defined for a certain rank, is automatically defined for any higher rank. With Remora, Slepak et al. tried to shed some light on the more "murkier corners" of the array-computational model. They do this by generalizing the array-computational model, which then allows them to both address some of the shortcomings of APL, but also allows them to extend the model to allow arrays of functions and arrays of arguments, which in turn allows for the parallel MIMD (multiple instruction, multiple data) architecture, rather than only SIMD (single instruction, multiple data) parallelism.

Finally, a more recent parallel array programming language is Dex [19, 21]. Rather than avoiding loops and explicit indexing, like NESL, NEPAL, DPH, and Repa had all done, Dex suggests that these features might introduce more clarity, if only they were implemented correctly. The main idea is to treat index sets as types and arrays as functions. In reality this "index comprehension" can also be seen as functions that return arrays, and allow declaring iteration over multiple dimensions in a single line. The main advantage of this is that loops make some parallelism opportunities very explicit. Also when these index comprehensions are presented back-to-back, opportunities for fusion become fairly clear as well. In their paper, they also show that on some benchmark problems, Dex performs similarly to Futhark [10], a functional array programming language that was specifically designed to write performant parallel GPU code.

2 Tracing

In the broadest terms, when we trace a program, we track the most basic steps the program takes provided some input. This is relevant for many applications in Computer Science. For example, certain automatic differentiation (AD) effectively implement the forward-pass as tracing, and then perform the reverse pass on the trace[cite me]. Tracing is also used in Artificial Intelligence, where tracing applications can help determine how much memory needs to be allocated, which can speed up training if the model is run multiple times[cite me].

However, despite its ambivalence, tracing is rarely properly defined, or defined only for a specific use case. So, in this section we set out to create a more general definition of tracing.

To start, it will help us along to set clear expectations for what we expect a tracing function to do. In the simplest terms, we expect a tracing program to take an input program with a set of inputs, and output a "trace". This output trace is defined as a sequence of operations the input program performed on the inputs to get the expected output. A term often used for an output trace is a "single-line program"[cite me]: a program without control flow. Clearing control flow like if-then-else statements is only natural: after all, provided some input the program will only walk down one variation of this branching path.

Furthermore, it is also generally accepted that the trace consists of a subset of the syntax

of the input program. Because we are generally more interested in what happens to the data in our program, we can "trace away" functions and data structures. More precisely, say our input program has the types as defined in Equation 1, where we have sum-types as $\tau + \sigma$, product types as $\tau \times \sigma$, functions as $\tau \to \sigma$, literal real numbers, and literal Booleans.

$$\tau, \sigma \coloneqq \tau + \sigma \mid \tau \times \sigma \mid \tau \to \sigma \mid \mathbb{R} \mid \mathbb{B} \tag{1}$$

We can imagine our simplified language, in which we will express our trace – as a language with fewer type formers. By choosing a subset of the type formers in our program, we can indicate which data structures should be traced away. A common option is to keep only "ground types", where we defined a ground type as a type that is not constructed of other types. Looking at our example in Equation 1, a trace keeping only these ground types would keep only the real numbers and the Booleans as they are not built of other types. Another common option is to keep only continuous types, tracing away all unground and discrete types. Doing that on our type set in Equation 1 would leave us with only the real numbers. This is under the assumption that the discrete types are not actually used as data we are interested in tracing of course, but since tracing will remove all control flow from the program, keeping Booleans and operations on Booleans intact may be meaningless.

The main take-away here is that there is some freedom of choice in what to trace away. What parts we keep and what parts we trace away is very dependent on what information we want to keep in our trace, which in turn is dependent on what our exact goal is for the tracing in the first place.

We can also choose to keep some of our unground types, but then we run into a problem. Say we keep only functions ($\tau \to \sigma$) and real numbers, but our input program contains a function with type $\tau \to (\sigma_1 + \sigma_2)$. This typing is valid in our input program, but no longer valid in our trace, so we find ourselves in a bind. It will be impossible to trace away the sum-type in the output of the function without tracing away the function itself. This is because tracing something away basically means either deconstructing or ignoring it in the trace. For instance, tracing away a tuple, would mean tracing the individual components of that tuple to trace it away. Whereas keeping things in the trace means just keeping them untouched. Therefor, we cannot keep a type like a function $\tau \to (\sigma_1 + \sigma_2)$ in our trace, because we cannot access the sum type without tracing away the function. Of course we could define a subset $\tau', \sigma' \coloneqq \mathbb{R}$ and then redefine (or add a definition for) our function so that it becomes $\tau' \to \sigma'$ making it safe to trace. This then underlines the rule at work here: we can only keep types that do cannot be constructed of types that are traced away. This is why the ground types are a natural set of types to keep, as they are never constructed from other types.

In a similar vein, we may also encounter operators in our trace that take in or produce types that are not allowed in our trace. For operators that produce a type that is not in our trace, tracing them away is no problem. Since we know we will not be interested in whatever output they produce for our trace, we can simply omit them from the trace altogether. For instance, if we keep only real numbers in our trace like before, an operator returning a Boolean value is of no interest for the trace. However, this is not a simple for operations that take in a type we wish to trace away, yet produce a type we wish to keep in our trace. A simple example of this is the "switch" operator, which takes in a Boolean value and two values of another type, of which it returns one

depending on the Boolean value (see Equation 2).

$$\begin{aligned} \texttt{switch}(\top, a, b) &= a \\ \texttt{switch}(\bot, a, b) &= b \end{aligned} \tag{2}$$

While the switch operator looks like it mimics if-then-else statements, it is generally accepted that it does so in a non-lazy way[cite me], where both a and b are evaluated before returning either. The main problem here is that we wish to keep operators that produce types we keep in our trace, yet we do not wish (or are not even able to) express the Boolean value in our trace. Now, due to switch statement's likeness to if-then-else, the solution here is pretty clear: only trace the value that gets returned. However, it is not always that easy: as we introduce arrays and array operations in Section 2.4, we will see how operations like mapping on an array need a special solution.

This all is to say that the while we can either ignore or homomorphically copy basic operations for our trace, sometimes we need a special solution. This is mainly because we do not want to lose the information that is needed to execute the trace as a single-line program, even if that means fudging our operations a little. This also means that, while the operations in our trace language might be a subset of the operations in the original expression language, they might contain modified operations

It seems that our tracing definition comes down to a function that takes in a program and an input to that program, and outputs the steps taken by the program run on the input. Where the input program takes uses some set of types, of which only a subset is kept in the trace, where the types in this subset may not be constructed using types from outside of the subset. What now remains is a concrete definition of the output of the tracing program. We have already stated that it should somehow contain the steps done in by the input program. The steps we wish to record are generally basic operations like arithmetic operations. But other operations, such as operations on arrays, can also be added depending on the ultimate goal of the tracing. More importantly, as we expect our trace to be akin to a single-line program, we may consider our trace as a series of let-bindings, akin to A-normal form[cite me]. This means storing each operation as a pair of a unique name or id and the operation performed (like the name and value of the declarations in a let-binding).

2.1 Tracing Correctness

Before going into specifics on how to implement tracing, it would also be a good idea to formalize when a trace is actually correct. Like we posed before, we start with some program formed from some expression language S, and some input I that is valid for that program. If we would wish to resolve a program S on input I, then we would need some evaluation function that produces the expected output O. Now, given some trace language T we can write a tracing function that gives us the trace and output of a specific program and input combination. We can write this out as the two functions eval and trace in Equation 3.

$$\begin{aligned} \text{eval} &: S \times I \to O \\ \text{trace} &: S \times I \to T \times O \end{aligned} \tag{3}$$

With this we can formalize two criteria for our trace. First, the trace, as a single line program $t \in T$ produced by the trace function needs to produce the correct output.

Now, as mentioned before, t might contain transformed operations, that are not present in S. Therefor we either need to look at traces $t \in S \cap T$, or use a different evaluation function. For now we will use the former, to assert the output criterium in Equation 4. Here we state that for any program s with any input i: if the trace t is also a valid program in S, that the evaluation of t on i should be the same as the evaluation of s on i or the output o we got out of the tracing function.

$$\begin{aligned} &\forall s \in S \\ &\forall i \in I \\ &\text{trace}(s, i) = (t \in T, o \in O) \\ &(t \in S \cap T) \rightarrow (\text{eval}(s, i) = \text{eval}(t, i) = o) \end{aligned} \tag{4}$$

Furthermore, tracing a trace t should also return that trace t. This is because we want to find the minimal straight-line program using tracing, and if tracing the trace we found reduces it somehow to a more minimal program, we know that the original trace was incomplete. This is expressed in Equation 5, where we assert that for some program $s \in S$ and some input $i \in I$, the trace t (produced by tracing s on i), is the same as the trace obtained from tracing t itself.

$$\begin{aligned} &\forall s \in S \\ &\forall i \in I \\ &\text{trace}(s, i) = (t \in T, o \in O) \\ &\text{trace}(t, i) = (t, o) \end{aligned} \tag{5}$$

The above statements, assert that a trace should produce the correct output value as expected from the input program, and that a trace should be its own trace. While these assertions do not say a lot about the nature of the actual trace, they do set some baseline requirements for the trace, and proving the correctness of a trace. This vagueness on the contents of the trace is partly because we cannot really say anything about a trace without dissecting the source program as well, which would bring us to a point very close to actual tracing itself. In another part however, this is because we do not want to make any assumptions what can or cannot be in our trace. While it is likely that some there is significant overlap between S and T, as mentioned, we might need some additions to T to actually be able to trace everything in S correctly. Also, whilst in practice it might be meaningless, a trace where $T = \emptyset$ is in itself not incorrect: any trace would simply be empty. In a similar vein a trace where $S \subseteq T$ would also be meaningless in practice, it is also not wrong: any trace would simply be the same as the source program.

As an additional note, Equation 4 also implies something interesting. If we want our trace to output the same value as the original program, we cannot trace away the type of the original programs output. Say we trace away Boolean values when we are tracing a program that returns a Boolean value, then we find ourselves stuck, because we trace away all operations that produce Boolean values. And of course, if our trace is not allowed to produce any Boolean values, we cannot produce the required output either. Therefor we must assure that the type of the output is valid in our trace as well.

2.2 Basic Tracing

We now define some basic tracing steps for some arbitrary language. For clarity's sake, we will do this is with Haskell[cite me]code. To do this we first define a language and

values on which we will operate. We do this in Listing 1, where we define a basic lambda calculus. Here the value types are represented as the algebraic data type (ADT) `Value`, where we find constructors for Booleans (`VBool`), real numbers (`VReal`), and functions (`VFunc`). Then we define the four terms of a basic lambda calculus in the `Expression` ADT: application (`EApply`), abstraction (`ELambda`), loose values (`ELift`), and variable reference (`ERef`). To make tracing a little more interesting we also add in if-then-else statements (`EIf`) and binary operators (`EOp2`). For those binary operators, we define four operations in the separate `Op2` ADT: addition (`Add`), equality (`Equ`), multiplication (`Mul`), and inequality (`Neq`). Finally, to make use of variable references, we define an environment as a mapping of strings to values. We interact with this environment in two ways: by inserting values into them, and getting values from them (indexing). The function signatures for these interactions, respectively `insert` and `(!)`, have been included in Listing 1 as well. We can use this language and evaluate it, an example of this has been provided in Appendix A.

```haskell
data Value = VBool Bool | VReal Float | VFunc (Value -> Value)

data Expression
    = EApply  Expression Expression
    | EIf     Expression Expression Expression
    | ELambda String     Expression
    | ELift   Value
    | EOp2    Op2        Expression Expression
    | ERef    String

data Op2 = Add | Equ | Mul | Neq

type Environment = Map String Value

-- Operations on maps:
-- (where Map a b is a mapping from keys of type a to values of type b)
insert :: a -> b -> Map a b -> Map a b
(!) :: Map a b -> a -> b
```

Listing 1: Minimal lambda calculus with added if-then-else and binary operators

With our language in Listing 1, we can almost start tracing. However, we must first decide which parts of the language we keep, and which parts we wish to trace away. In the previous section, we talked about how we can do this by selecting which type formers we wish to keep. In Listing 1, we have practically defined the types of our values by the data constructors present in the `Value` ADT as Booleans, real numbers, and functions. Let us now choose to keep only real numbers in the trace.

We now define a new ADT for traced values in Listing 2. This is only so we can incorporate a name into the values we wish to keep in our trace. These names will help us read the trace, and can be incrementing numbers or something entirely random, as long as they are unique. The basic idea is here to feed the `trace` function a number with which to generate the steps' names from, and increment the number every time we do. However, since this clutters the code while not being very interesting, we will assume we have some function `getName` that provides us with a unique name. Furthermore, it is important to see that we still have Boolean values and functions in our `TValue` ADT, even though we only wish to keep real numbers in our trace. This is because we might still need these values to resolve expressions, even if they never end up in the trace. We might also achieve this by extending our original `Value` ADT (from Listing 1) with

traced variants of values, but this is merely a point of preference. Finally, we have also changed the signature of the function value to return a trace as well, as we move on to functions we will see how this works.

```
1  data TValue = TBool Bool
2              | TReal String Float
3              | TFunc (TValue -> (TValue, Trace))
4
5  data Traced = TLift TValue | TOp2 Op2 String String
6
7  type TEnvironment = Map String TValue
8
9  type Trace = [(String, Traced)]
10
11 getName :: String
```

Listing 2: Basic trace building blocks

First however, with our basic building blocks for tracing set up, lets trace away these boolean values. We do this with the trace function in Listing 3. For now, we will leave out abstraction and application, as it might be easier to talk about tracing away Booleans first.

When we trace away Booleans, like in Listing 3, it is useful to think about where these Boolean values actually come up. In our minimal language from Listing 1, there are only three points: when they are included as literal values, as the input or output to basic operations, or as the conditional in if-then-else statements.

Let us start with the easiest first: literal Boolean values. When we encounter literal values during tracing, and they are of a type we wish to keep for our trace, we simply add their instantiation to the trace (as `TLift` in Listings 2 and 3). This is extremely straightforward: those values might be used by the operations we wish to trace, so they should be included in the trace themselves as well. For values of types we wish to trace away, we simply do not include them in the trace. After all, our trace should be fine without them, as we do not include any operations that require them in our trace, right? For now this seems obvious: if we look at the language in Listing 1, we see that there are no other uses for Booleans values than the use in the equality and inequality operators, and as the conditional in the if-then-else statement. Since we plan to trace these away, we do not appear to need these value instantiations in our trace either. However, at the end of Section 2.1 we already discussed what would happen if our program were to return a Boolean value. And in Section 2.4, we will see how this might not be entirely true when we talk about arrays and operations on arrays like mapping a function.

Tracing (away) simple operations like addition and equality (`EOp2`) are done in a similar vein. If the operation returns a value of a type we wish to keep in our trace, we include the operation in our trace as well. Similarly, if the operation returns a value that we do not wish to keep, we simply do not trace it. Again, if there was an operation that took in a value of a type we do not wish to trace, and returned one that we do wish to trace we run into a problem. Luckily, these operations are not included in our current example.

When we trace an if-then-else statement, we know we have to deal with a Boolean regardless. Luckily for us, we know we only need to trace one of the branches. This means quite simply, that we can ignore the if-then-else statement, and act like the

```
trace :: TEnvironment -> Expression -> (TValue, Trace)
trace n (EIf e1 e2 e3) =
    -- Since we e1 should resolve in a Boolean value, we do not need to trace it.
    let v1 = eval n e1
    in  case v1 of
        -- We can check for the type of v1 and its value in one go
        -- We trace only the relevant branch
        (VBool True)  -> trace n e2
        (VBool False) -> trace n e3
        _             -> error "Type mismatch in trace/EIf"

trace n (ELift v) =
    -- Check if v is a value we would like to trace
    case v of
        -- If yes return the transformed value with its simple trace
        (VReal v) ->
            -- Generate a name for this step and make the TValue
            let s  = getName
                v' = TReal s v
            -- Combine the TValue with a trace of its instantiation
            in  (v', [(s, v')])
        -- If we do not wish to trace something, we can just return the value
        -- with an empty trace.
        (VBool v) -> (TBool v, [])
        -- Instantiation is not allowed for functions, they need to be
        -- abstracted using ELambda
        _         -> error "Type mismatch in trace/ELift"

trace n (EOp2 op e1 e2) =
    -- We again first trace e1 and e2
    let (v1, t1) = trace n e1
        (v2, t2) = trace n e2
        -- We get a ready name in case we need it
        s = getName
    -- This case syntax allows us to select for the right operator with the
    -- right value types at the same time.
    in  case (op, v1, v2) of
        -- Since add and mul take in reals and produce one too, we trace both
        -- the operation and the origins of v1 and v2
        (Add, TReal s1 a, TReal s2 b) -> (TFloat s (a + b),
            (TOp2 op s1 s2) : t1 ++ t2)
        (Mul, TReal s1 a, TReal s2 b) -> (TFloat s (a * b),
            (TOp2 op s1 s2) : t1 ++ t2)
        -- For operations producing Bools we only return the result, but they
        -- are not traced, and therefor return an empty trace
        (Equ, TBool _  a, TBool _  b) -> (TBool (a == b), [])
        (Equ, TReal _  a, TReal _  b) -> (TBool (a == b), [])
        (Neq, TBool _  a, TBool _  b) -> (TBool (a /= b), [])
        (Neq, TReal _  a, TReal _  b) -> (TBool (a /= b), [])
        _                             -> error "Type mismatch in trace/EOp2"

-- There is nothing to trace when fetching a variable, but we still need to
-- actually get the value
trace n (ERef s1) = (n ! s1, [])
```

Listing 3: Tracing away Boolean values

program continued at the branch that is chosen. Since the input is provided, we can resolve the conditional immediately, and then just trace the appropriate branch.

Finally, tracing variable references are simple as well. Currently the only named variables that occur are those created in lambda abstractions or those that are provided as inputs. But no matter how they are created, variable reference does not require tracing. This is because the trace will reference the values regardless of whether they are instantiated on the spot or somewhere previously. And if they were defined previously, that definition is already in the trace somewhere.

2.3 Function Tracing

With our basic tracing established, we can now talk about tracing functions, which are more complicated. It is the tracing of abstracted functions that is the first issue here. The issue is that when we perform an abstraction (as with `ELambda`), there is nothing to trace. In fact, we can see this as an instantiation of a function literal, and when functions are not in our set of types to keep in the trace, this abstraction creates an empty trace. However, leaving it at that would mean we never actually trace the body of the function. Yet, at the time of the abstraction, we also do not yet know the input to the function either, meaning we cannot trace the body at that time. We must instead consider how we delay tracing until the function is actually applied. This is where our notation for `TFunc` (as in Listing 2) comes up. We wish that functions while tracing perform tracing themselves, thus return a `Trace` together with the return value. This is then what we do in the abstraction step: we set the trace on the body of the function as the body of the function we return. Similarly, we also give this tracing function call the environment at the time of abstraction, allowing the function body to access any free variables that were defined at that time. This makes application also very simple: we apply the function, and then just combine the trace of the functions instantiation, with that of the argument, and that of the functions execution. We also trace the functions instantiation, since at the time of application we do not now if the expression that leads to the function does anything else that we might need to trace as well. Finally, this is results in what we see in Listing 4, where we left out any patterns of trace that were already present in Listing 3.

2.3.1 Tracing let bindings

As an additional structure present in functional languages that we might wish to trace, there are let-bindings. Recall that let-bindings are effectively the same a lambda abstractions that are resolved immediately. This makes it extremely easy to resolve them, because we can just trace the let-side of the binding and add it to the environment for the tracing of the righthand-side.
Adding let-bindings and tracing them is done in Listing 5.

2.4 Array Tracing

Tracing on data structures like arrays provides us with a new problem that revolves around whether or not we wish to trace arrays away or not. We can see arrays as either structures that contain the data we are really after, which would require us to trace

```haskell
trace :: TEnvironment -> Expression -> (Value, Trace)
trace n (EApply e1 e2) =
    -- First trace e1 and e2
    let (v1, t1) = trace n e1
        (v2, t2) = trace n e2
    -- Check if v1 actually returns a function
    in  case v1 of
        -- Do the application, return the result and the combined trace
        TFunc f -> let (vf, tf) = f v2
                   in  (vf, tf ++ t2 ++ t1)
        _       -> error "Type mismatch in trace/EApply"

trace n (ELambda s e1) =
    -- Define the function, insert value x as variable s into the environment that is currently
    -- present, and trace the body
    let f = TFunc (\x -> trace (insert s x) e1)
    -- Return the function as abstracted function as a value, and no trace
    in  (f, [])
```

Listing 4: Tracing away functions

```haskell
data Expression
    = ...
    -- The string here is the name of the bound variable
    | ELet String Expression Expression

trace :: Environment -> Expression -> (TValue, Trace)
trace n (ELet s1 e1 e2) =
    -- Evaluate e1 first, then e2 with e1 in its environment
    let (v1, t1) = trace n e1
        (v2, t2) = trace (insert s1 v1 n) e2
    -- Return the value of e2 and the combined trace
    in  (v2, t1 ++ t2)
```

Listing 5: Tracing let bindings

them away, or as data in their own right which we wish to keep in the trace. Both scenarios provide us with interesting challenges.

Let us first talk about tracing arrays away. When we simply view arrays as another computational structure, they are not too complicated to trace away. When initializing an array, we just initialize all the individual values in the array. And when performing operations on items in the array, we instead perform those operations on the individual items again. That is, we do the operation like normal, but denote them as operations on separate items in the trace.

In Listing 6 we first add a arrays and array operations. While we said earlier that constructors in the `TValue` ADT only needed strings for names if they are traced, we need to make an exception for arrays. This is because when working with arrays our expression will never refer to individual values in arrays, only to the array itself (and using its individual values from there). This means that to consistently refer to values that were in arrays in the original expression, we need to give a little more structure to the naming scheme. We do this by taking the name of the array, and adding the index of the item to create a name that is unique yet identifiable. Furthermore, we add in array operations: iota (or range operation) (`Iota`), indexing (`Idx`), sum (`Sum`), and map (`Map`). It should be noted that iota and indexing take an integer argument as part of their operator. For iota this is the size of the array to create, and for indexing this is naturally the index to get. While we could allow our language with integers (or by casting floats) to allow using in-language numbers, this really is not all that interesting. If those arguments were part of our trace, it would just mean tracing them like any other number.

```
1  data Value = ... | VArray [Float]
2
3  data TValue = ... | TArray String [Float]
4
5  data Expression
6      = ...
7      | EOp0 Op0
8      | EOp1 Op1 Expression
9
10 data Op0 = Iota Int
11
12 data Op1 = Idx Int | Sum
13
14 data Op2 = ... | Map
```

Listing 6: Adding arrays

Now with arrays added to our language, we can actually trace them. This is done in Listing 7, where we again extend the trace function, leaving out any patterns that remain unchanged.

First off, when encountering a literal array, or creating one with the iota operator, we need to initialize every individual value. This is fairly simple, it just requires us to walk through the array and initialize every value like when we were initializing literal real values.

Indexing, in this mode, is equal to variable reference due to our naming scheme. This means then that we do not need to trace anything here.

```
trace :: TEnvironment -> Expression -> (TValue, Trace)
trace n (ELift v) =
    case v of
        -- Tracing for reals and Booleans remain unchanged
        (VReal  v) -> ...
        (VBool  v) -> ...
        (VArray v) -> let s = getName
                      -- For traceArrayLift see Listing 8
                      in  (TArray s v, traceArrayLift s v 0)

-- With only iota, we could write this a little more curtly, but for clarity we leave it like this
trace n (EOp0 op) =
    case op of
        (Iota r) ->
            let s = getName
                -- Define an array of size r, then lift is using traceArrayLift again
                v = [0.0 .. (r - 1)]
                -- For traceArrayLift see Listing 8
            in  (TArray s v, traceArrayLift s v 0)

trace n (EOp1 op e1) =
    -- Again we first trace e1, and we get a name ready as well
    let (v1, t1) = trace n e1
        s = getName
    in  case (op, v1) of
        -- Indexing is like variable reference, we do not need to add to the trace,
        -- but we need to create the name to be consistent
        (Idx i, TArray s1 v) =
            -- Get the actual item using indexing (!!)
            let x  = v !! i
                s' = s1 ++ '!' : show i
            in  (TReal s' x, t1)
        -- For traceArraySum see Listing 9
        (Sum, TArray s1 v) =
            let (vs, ts) = traceArraySum s1 v 0
            -- We must not forget to add the trace of e1 to our trace here
            in  (vs, ts ++ t1)

trace n (EOp2 op e1 e2) =
    -- Again we first trace e1 and e2, and we get a name ready as well
    let (v1, t1) = trace n e1
        (v2, t2) = trace n e2
        s = getName
    in  case (op, v1, v2) of
        ...
        (Map, TFunc f, TArray sa va) ->
            -- For traceArrayMap see Listing 8
            let (vm, tm) = traceArrayMap f sa va 0
            -- Combine the traces
            in  (vm, tm ++ t1 ++ t2)
```

Listing 7: Tracing away arrays

```haskell
-- traceArrayLift takes the name of the array, the contents, and the current index
traceArrayLift :: String -> [Float] -> Int -> Trace
-- Empty lists get no trace
traceArrayLift _ []     _ = []
traceArrayLift s (x:xs) i =
    -- Create the name for this item from the array's name and the current index
    let s' = s ++ '!' : show i
    -- Trace x as a single real number
        tx = TLift (TReal s' x)
    -- Trace the rest of the array
        txs = traceArrayLift s xs (i + 1)
    -- Return the combined trace
    in  tx : txs

-- traceArrayMap takes the function to map, the name of the old array, the name of the new array,
-- the contents of the old array, and the current index
traceArrayMap :: (TValue -> (TValue, Trace)) -> String -> String -> [Float]
    -> Int -> (TValue, Trace)
traceArrayMap _ _  sn []     _ = (TArray sn [], [])

traceArrayMap f so sn (x:xs) i =
    -- Get the current value from the array with the right name
    let current  = TReal (so ++ '!' : show i) x
    -- Get the result from the function application
        (fv, ft) = f current
    -- Get the results from the rest of the array
        (xsv, xst) = traceArraymap f so sn xs (i + 1)
    -- To add to the TArray and to rename fv we use this case-of statement
    in  case (fv, xsv) of
        (TReal s' v, TArray _ xsv') ->
            -- Add this item to the new array
            let vn = TArray sn (v : xsv')
            -- Rename fv in the function trace to the correct name
                ft' = rename s' (sn ++ '!' : show i) ft
            -- Finally return the new array and the combined trace
            in  (vn, ft' ++ xst)

rename :: String -> String -> Trace -> Trace
```

Listing 8: Tracing array instantiation and array mapping

The sum operator is a little more in-depth, as shown in Listing 9. However, this is a lot of code for a very simple principle, and a couple edge cases. The principle is, add the first two values in the array together, and then every following item to that result and so on. And we have edge cases for singleton and empty arrays. It is worth explicitly stating that every addition done whilst summing the array gets its own unique name and step in the trace. This means that an operation that is single step the original program, explodes to a bunch of steps (the length of the array minus one) in the trace. This is because we decided to trace away arrays, and we will see later on how we save ourselves from this by not tracing away arrays.

The map operator is funky in a way similar to sum. In essence, we take each item in the array and apply it to the function as expected. However, we run into a little problem with our naming scheme. For map, the items, once mapped on, are placed back into a new array. This means that, according to the scheme we laid out, the items in this array should be named in reference to the new array, however this is not something the call to trace in the function body considers. Luckily we can resolve this by renaming the returned value from that function, and changing the name in the trace. The signature of a function that does this is also included at the end of Listing 8, but its exact implementation is not of importance here.

While the concepts behind tracing away arrays are hopefully not too difficult to understand, it should be obvious from Listings 8 and 9 that the implementation becomes more complex. Now while that is not really a problem, we should really note that the trace becomes messier as well. This is especially problematic if we actually want to read the trace to see what is going on: not impossible, but also not pleasant, especially with large arrays. So perhaps we are tempted to keep arrays in the trace instead, or perhaps we are interested in the trace of arrays specifically.

Luckily for us, in large parts tracing while keeping arrays is fairly easy. This is because we can treat most operations like how we treated operations for real numbers. This has been done in Listing 10, except for map, where replace the tracing patterns from Listing 7.

In our current language, the main point of difficulty and interest is the map operation. Map takes in a function, which is not a type we wish to keep in our trace, however it produces an array which we wish to keep in our trace. While we might be tempted to just discard the function component, we cannot do that because it provides the trace from the original array to the new array. Without that information our trace is no longer a functional (straight-line) program.

The basic way to solve this, the naïve method, would be to attach an array of traces to the map operator, so they can be followed to derive the correct results. To easily do this we extend our `Traced` ADT with a special map constructor (`TMap`). We show this in Listing 11. The traces in the `TMap` constructor correspond with the application of the function to be mapped to the individual item, for each item. The string references the array the map is performed on.

Now, while the naïve way is fine in functionality, it does again create some overhead (a trace for each item in the array) by splitting the trace into multiple smaller traces. And if the function is the same for every item in the array, we may find ourselves saving a lot of redundant data. Now, this may be necessary: at the time we map a function over an array, we do not know if it will act the same for every input. Perhaps there is some control flow in the function body that checks if a number is even, or a factor

```haskell
-- traceArraySum starts the trace, and traceArraySum' completes it
-- This is necessary because we do not know the number of items in the array
-- traceArraySum takes only the array to sum
traceArraySum (TArray _ []) =
    let s = getName
        v = TReal s 0
    -- The sum of an empty array means just lifting the value 0
    in  (v, [(s, TLift v)])

traceArraySum (TArray _ [x]) =
    let s = getName
        v = TReal s x
    -- The sum of a singleton array is just that one value
    in  (v, [(s, TLift v)])

traceArraySum (TArray sa (x:y:z)) =
    -- When summing on a larger array, the first sum is of the first two items
    let sx = sa ++ ``!0''
        sy = sa ++ ``!1''
        s  = getName
        v  = TReal s (x + y)
        -- Get the result, and the trace of the rest of the array with traceArraySum'
        (rv, rt) = traceArraySum' (TArray sa z) 2 v
    -- Return the final result, but do not forget the trace of the first sum
    in  (rv, (s, TOp2 Add sx sy) : rt)

-- traceArraySum' takes the array we sum over, the current index, and the last calculated value
traceArraySum' :: TValue -> Int -> TValue -> (TValue, Trace)
-- When we are done, return the value
traceArraySum' (TArray _ []) _ v = (v, [])

traceArraySum' (TArray sa (x:xs)) i (TReal sr r) =
    -- Get the name for this item
    let sx = sa ++ '!' : show i
        -- Get the name for this addition step
        s  = getName
        -- Get the result of the rest of the array
        (v, t) = traceArraySum' (TArray sa xs) i (TReal s (x + r))
    -- Return the final result, and add this steps addition to the trace
    in  (v, (s, TOp2 Add sx sr) : t)
```

Listing 9: Tracing the sum operator

```
trace :: TEnvironment -> Expression -> (TValue, Trace)
trace n (ELift v) =
    case v of
        (VReal  v) -> ...
        (VBool  v) -> ...
        (VArray v) ->
            let s = getName
            -- Literal lifting of arrays becomes real simple
            in  (TArray s v, [(s, TLift (TArray s v))])

trace n (EOp0 op) =
    case op of
        (Iota r) ->
            let s = getName
                v = [0.0 .. (r - 1)]
                -- Iota again becomes very similar to literal array lifting
            in  (TArray s v, [s, TLift (TArray s v)])

trace n (EOp1 op e1) =
    -- We trace e1 first, and create a name just in case
    let (v1, t1) = trace n e1
        s = getName
    in  case (op, v1) of
        (Idx i, TArray s1 v) =
            let x  = v !! i
                s' = s1 ++ '!' : show i
            -- Now we trace arrays, indexing becomes more relevant to add to our trace,
            -- as the individual item has not been defined before
            in  (TReal s' x, (s', TOp1 op s1) : t1)
        -- Sum becomes very simple, just apply it to the array
        (Sum, TArray s1 v) = (TReal s (sum v), (s, TOp1 Sum s1) : t1)
```

Listing 10: Tracing whilst keeping arrays

```haskell
data Traced
    = ...
    | TMap [Trace] String

trace :: TEnvironment -> Expression -> (TValue, Trace)
trace n (EOp2 op e1 e2) =
    -- We first trace e1 and e2, and generate a name
    let (v1, t1) = trace n e1
        (v2, t2) = trace n e2
        s = getName
    in  case (op, v1, v2) of
        ...
        (Map, TFunc f, TArray sa va) ->
            let (vs, ts) = traceMapNaive f sa s va 0
            -- The trace becomes TMap, the collection of traces ts, on the old array v2 (with
            -- name sa)
            in  (vs, [(s, TMap ts sa)])

-- traceMapNaive takes in the function to be mapped, the name of the old array, the name of the
-- new array, the contents of the old array, and the current index
traceMapNaive :: (TValue -> (TValue, Trace)) -> String -> String -> [Float]
    -> Int -> (TValue, [Trace])
-- A map over an empty array returns the empty array and no traces
traceMapNaive _ _  sn []     _ = (TArray sn [], [])

traceMapNaive f so sn (x:xs) i =
    -- Create specific names for the old and new value
    let old = so ++ '!' : show i
        new = sn ++ '!' : show i
        -- Apply the function, getting the value for x and its trace
        (xv, xt) = f (TReal old x)
        -- Apply the function for the rest of the map
        (xsv, xst) = traceMapNaive f so sn xs (i + 1)
        -- We use a case-of statement to append xv to xsv and to rename xv in xt
    in  case (xv, xsv) of
        (TReal s' v, TArray _ vs) ->
            let xt' = rename s' new xt
            in  (TArray sn (v : vs), xt' : xst)
```

Listing 11: Tracing map while keeping arrays, naïvely

of three, or something else entirely. In such a case, having a trace for each item may be strictly necessary. However, it also highlights for which functions it may not be: functions without control flow or branching. After all, these functions are little straight-line programs, and should act the same no matter on what input they are applied. Writing a function that checks if a the body of a lambda abstraction contains branching is very simple for this language: currently the only expression term that can introduce branching is the if-then-else statement. Unfortunately, we cannot check that at the moment the we trace a map operator. This is because any function here would already have been abstracted to a `TArray` value. So, we would need to check for branching when we are abstracting the function and we also need a way to convey if a specific instance of `TFunc` contains branching or not. We write branch-checking into functions in Listing 12. For most terms we can just commute the branch checking to the arguments of that term, but there are a couple exceptions. If-then-else statements are the definition of branching in our language, so they return 'true', and no branching can occur in literal instantiation (`ELift`) or nullary operators (`Op0`) (literal instantiation can also be rewritten as a nullary operator), so they always return 'false'. Only for variable reference, which may return a value without actually providing a code to check, we need to see if the value is a function, and whether it has the branching flag set or not. This works because we set the branching flag when functions are defined using abstraction, and because functions may not be entered as literals.

```
1  data TValue
2      = ...
3      -- Add a branching flag to TFunc
4      | TFunc Bool (TValue -> (TValue, Trace))
5
6  branchCheck :: TEnvironment -> Expression -> Bool
7  -- Encountering an if-else-statement means a encountering a branch
8  branchCheck _ (EIf _ _ _) = True
9
10 branchCheck n (EApply e1 e2) = branchCheck n e1 || branchCheck n e2
11 branchCheck n (ELambda _ e1) = branchCheck n e1
12 branchCheck n (ELet _ e1 e2) = branchCheck n e1 || branchCheck n e2
13 -- ELift is always false, because lifting functions is not allowed
14 branchCheck _ (ELift) = False
15 branchCheck _ (EOp0 _) = False
16 branchCheck n (EOp1 _ e1) = branchCheck n e1
17 branchCheck n (EOp2 _ e1 e2) = branchCheck n e1 || branchCheck n e2
18
19 -- If our variable contains a function we need to check what it has the branching flag set to
20 branchCheck n (ERef s1) = case n ! s1 of
21     (TFunc b _) -> b
22     _           -> False
```

Listing 12: Checking for branches

With our branch checking defined we still need to talk about how we actually apply that and make a trace for map that requires less information. The basic idea here is that we can essentially perform vectorization of our function on the array in our trace: we rewrite the trace such that the function is "applied" to the whole array, rather than its individual items. Now without support for this in our language, this basically amounts to syntactic sugar in our trace, however it will provide us with a much clearer trace. This has been done in Listing 13, where we again add a map operator to our `Traced` ADT. This is because we may need to use the naïve method if a function contains branching

and we cannot vectorize it. In Listing 13 we still use `traceMapNaive` to actually map over our array. This is because we need to get the value of the array regardless, and our function value (`TFunc`) will return traces regardless if we need them or not. Then we can just take the first trace returned by the naïve map tracing, and rename all references to the first item of both the new and old arrays, to references of the whole old and new arrays respectively. For this end we define a function `deepRename` at the end of Listing 13. Like with the renaming function in Listing 8, the implementation of this function is not all that interesting: since all a `Trace` object is, is a list of tuples with a name that may need renaming and a `Traced` constructor referencing zero to two strings that may need renaming. All `deepRename` would do is go over these items and rename any occurrences it finds.

```
data Traced
    = ...
    -- We leave the naive TMap untouched
    | TMap  [Trace] String
    -- And add a new one for vectorized traces
    | TMapV Trace   String

trace :: TEnvironment -> Expression -> (TValue, Trace)
trace n (ELambda s1 e1) =
    -- We add branch checking when we handle abstraction
    let b = branchCheck e1
        f = TFunc b (\x -> trace (insert s x) e1)
    in  (f, [])

trace n (EOp2 op e1 e2) =
    let (v1, t1) = trace n e1
        (v2, t2) = trace n e2
        s = getName
    in  case (op, v1, v2) of
        ...
        (Map, TFunc b f, TArray sa va) ->
            -- We first get the result array (and all the traces) using the naive method
            let (vs, ts) = traceMapNaive f sa s va 0
            in  if   b
                -- If the function contains branching, use the naive method
                then (vs, (s, TMap ts sa) : t1 ++ t2)
                -- Otherwise use the new method
                else let t = vectorizeTrace sa s (head ts)
                     in  (vs, (s, TMapV t sa) : t1 ++ t2)

vectorizeTrace :: String -> String -> Trace -> Trace
-- Rename the references to individual items to the whole array
vectorizeTrace so sn t = deepRename iso so (deepRename isn sn t)
    -- The names for the individual items in this trace
    where iso = so ++ ``!0''
          isn = sn ++ ``!0''

deepRename :: String -> String -> Trace -> Trace
```

Listing 13: Array mapping with trace vectorization

What is important to take away from the shenanigans with the map operators is that, whilst our definitions and correctness assertions from Sections 2 and 2.1 gave us some

guidance, there is ultimately no single way to trace everything. The most important factor here is to keep reminding ourselves of the information we wish to keep in the trace. Not only the value types, but we also need the information needed to actually run the trace as a program. Keeping this in mind, it becomes much more obvious how to trace the map operation.

References

[1] Blelloch, G. E. *NESL: a nested data parallel language.* Carnegie Mellon Univ., 1992.

[2] Chakravarty, M. M., and Keller, G. More types for nested data parallel programming. *ACM SIGPLAN Notices 35*, 9 (2000), 94–105.

[3] Chakravarty, M. M., and Keller, G. Functional array fusion. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming* (2001), pp. 205–216.

[4] Chakravarty, M. M., Keller, G., Lechtchinsky, R., and Pfannenstiel, W. Nepal—nested data parallelism in haskell. In *Euro-Par 2001 Parallel Processing: 7th International Euro-Par Conference Manchester, UK, August 28–31, 2001 Proceedings 7* (2001), Springer, pp. 524–534.

[5] Chakravarty, M. M., Keller, G., Lee, S., McDonell, T. L., and Grover, V. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming* (2011), pp. 3–14.

[6] Chakravarty, M. M., Leshchinskiy, R., Peyton Jones, S., Keller, G., and Marlow, S. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming* (2007), pp. 10–18.

[7] Ellmenreich, N., Lengauer, C., and Griebl, M. Application of the polytope model to functional programs. In *Languages and Compilers for Parallel Computing: 12th International Workshop, LCPC'99 La Jolla, CA, USA, August 4–6, 1999 Proceedings 12* (2000), Springer.

[8] Feitelson, D. G. Packing schemes for gang scheduling. In *Job Scheduling Strategies for Parallel Processing: IPPS'96 Workshop Honolulu, Hawaii, April 16, 1996 Proceedings 2* (1996), Springer, pp. 89–110.

[9] Grelck, C., and Scholz, S.-B. Generic parallel array programming in sac. In *21. Workshop der GI-Fachgruppe 2.1.4 Programmiersprachen und Rechenkonzepte, Bad Honnef, Germany* (2005), pp. 43–53.

[10] Henriksen, T., Serup, N. G., Elsman, M., Henglein, F., and Oancea, C. E. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2017), pp. 556–571.

[11] Herrmann, C. A., and Lengauer, C. Parallelization of divide-and-conquer by translation to nested loops. *Journal of functional programming 9*, 3 (1999), 279–310.

[12] Hill, J. M. D. *Data parallel lazy function programming.* PhD thesis, Queen Mary, University of London, 1995.

[13] Iverson, K. E. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference* (1962), pp. 345–351.

[14] Jay, C. B., and Cockett, J. R. B. Shapely types and shape polymorphism. In *Programming Languages and Systems—ESOP'94: 5th European Symposium on Programming Edinburg, UK, April 11–13, 1994 Proceedings 5* (1994), Springer, pp. 302–316.

[15] Keller, G., and Chakravarty, M. M. Flattening trees. In *Euro-Par'98 Parallel Processing: 4th International Euro-Par Conference Southampton, UK, September 1–4, 1998 Proceedings 4* (1998), Springer, pp. 709–719.

[16] Keller, G., and Chakravarty, M. M. On the distributed implementation of aggregate data structures by program transformation. In *Parallel and Distributed Processing: 11th IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing San Juan, Puerto Rico, USA, April 12–16, 1999 Proceedings 13* (1999), Springer, pp. 108–122.

[17] Keller, G., Chakravarty, M. M., Leshchinskiy, R., Peyton Jones, S., and Lippmeier, B. Regular, shape-polymorphic, parallel arrays in haskell. *ACM Sigplan Notices 45*, 9 (2010), 261–272.

[18] Keller, G. C. *Transformation-based implementation of nested data parallelism for distributed memory machines*. PhD thesis, Technical University of Berlin, Germany, 1999.

[19] Maclaurin, D., Radul, A., Johnson, M. J., and Vytiniotis, D. Dex: array programming with typed indices. In *Program Transformations for ML Workshop at NeurIPS 2019* (2019).

[20] Maessen, S. A. A. J.-W., Augustsson, L., and Nikhil, R. S. Semantics of ph: A parallel dialect of haskell. In *In Proceedings from the Haskell Workshop (at FPCA 95)* (1995), pp. 35–49.

[21] Paszke, A., Johnson, D., Duvenaud, D., Vytiniotis, D., Radul, A., Johnson, M., Ragan-Kelley, J., and Maclaurin, D. Getting to the point. index sets and parallelism-preserving autodiff for pointful array programming. *arXiv preprint arXiv:2104.05372* (2021).

[22] Peyton Jones, S., Leshchinskiy, R., Keller, G., and Chakravarty, M. M. Harnessing the multicores: Nested data parallelism in haskell. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science* (2008), Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[23] Scholz, S.-B. Single assignment c-functional programming using imperative style. In *Proceedings of IFL* (1994), vol. 94.

[24] Scholz, S.-B. Single assignment c: efficient support for high-level array operations in a functional setting. *Journal of functional programming 13*, 6 (2003), 1005–1059.

[25] Slepak, J., Shivers, O., and Manolios, P. An array-oriented language with static rank polymorphism. In *ESOP* (2014), vol. 8410, Springer, pp. 27–46.

A ADT Evaluation

In Section 2.2, we introduced an extended lambda calculus. In this section we will quickly go over how an evaluator function for this ADT would look like in Haskell. We define our evaluator function in Listing 14, using the definitions of `Expression`, `Value`, and `Environment` from Listing 1.

```haskell
eval :: Environment -> Expression -> Value

eval n (EApply e1 e2) =
        -- Evaluate e1 and e2 first
    let v1 = eval n e1
        v2 = eval n e2
    in  case v1 of
            -- Only apply v1 to v2 if v1 is a function as expected
            VFunc f -> f v2
            _       -> error "Type mismatch in eval/EApply"

eval n (EIf e1 e2 e3) =
    -- Evaluate e1 as the condition of the if-then-else statement
    case eval n e1 of
        -- If e1 evaluates to true, evaluate e2
        VBool True  -> eval n e2
        -- Otherwise, evaluate e3
        VBool False -> eval n e3
        _           -> error "Type mismatch in eval/EIf"

-- For abstractions, we return the function by moving the evaluation into the body.
-- Where we insert the anonymous value x into the environment as it was when the
-- function was defined.
eval n (ELambda s1 e1) = VFunc $ \x -> eval (insert s1 x n) e1

eval n (ELift v1) = v1

eval n (EOp2 op e1 e2) =
        -- Evaluate e1 and e2 first
    let v1 = eval n e1
        v2 = eval n e2
        -- This case syntax allows us to select for the right op with the right
        -- value types at the same time.
    in case (op, v1, v2) of
        (Add, VFloat a, VFloat b) -> VFloat $ a +  b
        (Equ, VBool  a, VBool  b) -> VBool  $ a == b
        (Equ, VFloat a, VFloat b) -> VBool  $ a == b
        (Mul, VFloat a, VFloat b) -> VFloat $ a *  b
        (Neq, VBool  a, VBool  b) -> VBool  $ a /= b
        (Neq, VFloat a, VFloat b) -> VBool  $ a /= b
        _                         -> error "Type mismatch in eval/EOp2"

-- Resolving references means getting the value from the environment by name.
eval n (ERef s1) = n ! s1
```

Listing 14: ADT Evaluator