Paper

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

Abstract

Abstract.

`This is an` amazing test
`This is an amazing test`

1 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 1.4, 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).

1.1 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'd 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'll 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.

1.2 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 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
```

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.

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

```haskell
data TValue = TBool Bool
            | TReal String Float
            | TFunc (TValue -> (TValue, Trace))

data Traced = TLift TValue | TOp2 Op2 String String

type TEnvironment = Map String TValue

type Trace = [(String, Traced)]

getName :: String
```

Listing 2: Basic trace building blocks

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 1.1 we already discussed what would happen if our program were to return a Boolean value. And in Section 1.4, 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 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.

```
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

1.3 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.

```
1  trace :: TEnvironment -> Expression -> (Value, Trace)
2  trace n (EApply e1 e2) =
3      -- First trace e1 and e2
4      let (v1, t1) = trace n e1
5          (v2, t2) = trace n e2
6      -- Check if v1 actually returns a function
7      in  case v1 of
8          -- Do the application, return the result and the combined trace
9          TFunc f -> let (vf, tf) = f v2
10                    in  (vf, tf ++ t2 ++ t1)
11         _       -> error "Type mismatch in trace/EApply"
12
13 trace n (ELambda s e1) =
14     -- Define the function, insert value x as variable s into the environment that is currently
15     -- present, and trace the body
16     let f = TFunc (\x -> trace (insert s x) e1)
17     -- Return the function as abstracted function as a value, and no trace
18     in  (f, [])
```

Listing 4: Tracing away functions

1.4 Array Tracing

An important part of programming, including the programs where tracing is likely to be used, is arrays and other data structures. Tracing data structures like arrays might seem a little more complicated, however the main point is about whether or not we want to trace away arrays. For our upcoming example, we first add Arrays to our language

from Section 1.2 (Listings 1, **??**, and **??**), as shown in Listing 5. For now, we will limit our discussion to arrays of real numbers only. We will also add operations on arrays, currently just as the operations producing a single value from the array, like indexing or sum ($[\mathbb{R}] \to \mathbb{R}$), and those that produce a new array like mapping a function ($[\mathbb{R}] \to [\mathbb{R}]$).

Types:

$\sigma, \tau \coloneqq \mathbb{B} \mid \mathbb{R} \mid \sigma \to \tau \mid [\mathbb{R}]$

Terms:

$s, t \coloneqq$

$\quad \ldots$

$\mid \; [s_1 : \mathbb{R}, \ldots, s_n : \mathbb{R}]$

$\mid \; Op_{[\mathbb{R}]}(s) : \mathbb{R}$

$\mid \; Op_{[\mathbb{R}]}(s) : [\mathbb{R}]$

Listing 5: Adding arrays

If we were to extend our example from Section 1.2, we might be inclined to not allow arrays in our trace (and keep only real numbers), meaning we would have to trace the arrays away. This is not too complicated. When tracing away arrays as terms, we might break them down into their component parts. For instance, an array of real numbers, can be traced as just the instantiation of every real number in the array. The same goes for operations done to the array: we just trace these operations as they are applied to the individual items in the array, effectively ignoring the fact that these items were in an array in the first place. Listing 6 shows what this would look like in the form of our earlier examples. Here we have $Op^*_{\mathbb{R}} : \mathbb{R}$ as the operator we are applying on a single item in an array. To clarify, for operations like sum, $Op^*_{\mathbb{R}} : \mathbb{R}$ would be $s_1 + \cdots + s_n$, where all these binary additions would end up in the trace as single operations. For operations like mapping a function, $Op^*_{\mathbb{R}} : \mathbb{R}$, would represent the function application on a single item in the array, where we would trace the body of the function like we had traced generic application earlier. This is probably too vague, the notation also falls a bit short here.

$$\text{trace}([\Gamma, s_1, \ldots, s_n]) \Rightarrow ([s_1, \ldots, s_n], \text{trace}(\Gamma, s_1) \cup \cdots \cup \text{trace}(\Gamma, s_n))$$

$$\text{trace}(\Gamma, Op_{[\mathbb{R}]}([s_1, \ldots, s_n]) : \mathbb{R}) \Rightarrow (\text{eval}(Op_{[\mathbb{R}]}([s_1, \ldots, s_n]) : \mathbb{R}),$$
$$\quad \text{trace}(\Gamma, Op^*_{\mathbb{R}}(s_1) : \mathbb{R}) \cup \cdots \cup \text{trace}(\Gamma, Op^*_{\mathbb{R}}(s_n) : \mathbb{R}))$$

$$\text{trace}(\Gamma, Op_{[\mathbb{R}]}([s_1, \ldots, s_n]) : [\mathbb{R}]) \Rightarrow (\text{eval}(Op_{[\mathbb{R}]}([s_1, \ldots, s_n]) : [\mathbb{R}]),$$
$$\quad \text{trace}(\Gamma, Op^*_{\mathbb{R}}(s_1) : \mathbb{R}) \cup \cdots \cup \text{trace}(\Gamma, Op^*_{\mathbb{R}}(s_n) : \mathbb{R}))$$

Listing 6: Tracing away arrays

We see in Listing 6 a template for tracing almost any operation performed on the array. This comes down to knowing that array operations are interested in the individual values of the array, rather than the array as an closed object. However, tracing arrays away also comes with some caveats. First off, if we have large arrays, our traces will become very large, even if the operations performed on these large arrays are relatively simple. Furthermore, it might be unclear in the trace that we were even using arrays in the original program, which depending on your application might be a problem.

So what if we do not trace away these arrays? We would first need to extend the types

in our trace with arrays ($[\mathbb{R}]$), however if we wish to keep arrays in our trace this should not be a problem. What is more interesting is how we would actually trace these arrays. Like the operations on regular real numbers, if we keep arrays in our trace, we keep operations on arrays in our trace. This makes tracing arrays really simple as well, as we just repeat arrays and operations on arrays in our trace, this is shown in Listing 7. This does mean that we have no room for individual array items in our trace, except if they actually come out of the array (like with indexing).

$$\text{trace}(\Gamma, [s_1, \ldots, s_n]) \Rightarrow ([s_1, \ldots, s_n], \{[s_1, \ldots, s_n]\})$$

$$\text{trace}(\Gamma, Op_{[\mathbb{R}]}(s) : \mathbb{R}) \Rightarrow (\text{eval}(Op_{[\mathbb{R}]}(s) : \mathbb{R}), \{Op_{[\mathbb{R}]}(s) : \mathbb{R}\} \cup \text{trace}(\Gamma, s))$$

$$\text{trace}(\Gamma, Op_{[\mathbb{R}]}(s) : [\mathbb{R}]) \Rightarrow (\text{eval}(Op_{[\mathbb{R}]}(s) : [\mathbb{R}]), \{Op_{[\mathbb{R}]}(s) : [\mathbb{R}]\} \cup \text{trace}(\Gamma, s))$$

Listing 7: Keeping arrays in traces

A ADT Evaluation

In Section 1.2, 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 8, 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 8: ADT Evaluator