Paper

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

Abstract

Abstract.

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 isn't 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 aren't actually used as data we're 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 can't 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 1.3, 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 follows the two functions in eval and trace:

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

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 noted previously,

$$\begin{aligned} &\forall t \in \text{ExprExpr} \cap \text{SimpExpr} : \\ &\forall i \in \text{Input} : \\ &\texttt{snd}(\text{trace}(t, i)) = \text{run}(t, i) \land (\forall j \in I : \text{run}(\texttt{fst}(\text{trace}(t, j)), j) = \text{run}(t, j)) \end{aligned}$$

1.2 Tracing Steps

We now define some basic tracing steps for some arbitrary language. To do this we first define a language on which we will operate. We do this in Listing 1, where we define a basic lambda calculus.

Types:

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

Terms:

$s, t \coloneqq$

$r \in \mathbb{R}$ (literal real numbers)

$\mid \lambda(x : \tau).s$ (abstraction)

$\mid s\ t$ (application)

Listing 1: Basic language

Tracing through this lambda calculus is pretty straightforward. We will define our trace as some set T, such that our tracing function $\text{trace}(\Gamma, e) \to (v, T)$ gives us a value v and a trace T for some environment Γ and some expression e. To not go into too much implementational overhead, we will ignore naming the items in the trace for now. However, for clarity we will define an evaluation function $\text{eval}(e)$, which will resolve an expression e to its actual value. Listing 2 shows how tracing the lambda calculus in Listing 1 would look like, if we choose to trace away the function type (leaving us only with real numbers). We see that tracing a literal real number r, just adds r to the trace, after all real numbers aren't traced away. Tracing application is also fairly straightforward: we trace the argument t, and then the function s with t. Only our λ-abstraction is a little more complex. Since nothing actually happens our trace here is empty, however this does not mean that we can ignore the steps the abstracted expression takes, those we might still be interested in. So we can rewrite the lambda expression, to move the trace over, and trace only the body when it is actually applied. This also has as a side-effect on application, that we call the result of the trace of s as a function, as that will be the rewritten lambda expression we got from tracing abstraction.

$\text{trace}(\Gamma, r \in \mathbb{R}) \Rightarrow (r, \{r\})$

$\text{trace}(\Gamma, \lambda(x : \tau).s) \Rightarrow (\lambda(x : \tau).trace(\Gamma \cup x, s), \emptyset)$

$\text{trace}(\Gamma, s\ t) \Rightarrow (\text{eval}(s\ t), \text{trace}(\Gamma, s)(\text{trace}(\Gamma, t)))$

Listing 2: First tracing rules

It should be noted that tracing abstraction and application this way, effectively traces away function types. This is generally what we want from a trace, but it is somewhat meaningful to realize what would happen if we chose not to do that. In that case, we no longer deconstruct functions, so our trace on abstraction becomes similar to our trace on real numbers: we just return the function as we found it. As for application, we can no longer trace the body of our function, so all that remains in tracing its argument and perhaps denoting the application took place.

Now to illustrate how tracing would eliminate control-flow, we would first need to add some to our language. We can extend our language in Listing 1 with an if-then-else state-

ment, as shown in Listing 3. To make it easy on ourselves, we also add Booleans to the language as the domain $\mathbb{B} = \{\top, \bot\}$. However, we maintain, that we only want to trace to contain real numbers as a type, so this means we need to trace away these Booleans, and any operations performed on them. Which brings us to the following point: there are no operations in our language. Whilst we can validate our if-then-else statement with literal Booleans, it is probably more meaningful to actually add operations to the language. To keep things general, we will denote these operations as $Op_{\mathbb{X}}(s_1, \ldots, s_n) : \mathbb{Y}$, where $\mathbb{X}$ is the domain of the n inputs $s_1, \ldots, s_n$, and $\mathbb{Y}$ is the codomain of the operation. To keep things simple, we will assume for now that $Op : \mathbb{X}^n \to \mathbb{Y}$ for any domain $\mathbb{X}$ and codomain $\mathbb{Y}$. In Listing 3, we define three domain-codomain pairs for basic operators: $\mathbb{B} \to \mathbb{B}$ for comparing Booleans, $\mathbb{R} \to \mathbb{B}$ for comparing real numbers, and $\mathbb{R} \to \mathbb{R}$ for arithmetic operators. These three categories encapsulate the vast majority of what are considered "basic operations".

Types:

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

Terms:

$s, t \coloneqq$

$\quad \ldots$

$\mid$ `if` $s : \mathbb{B}$ `then` $t_\top : \tau$ `else` $t_\bot : \tau$

$\mid Op_{\mathbb{B}}(s_1, \ldots, s_n) : \mathbb{B}$

$\mid Op_{\mathbb{R}}(s_1, \ldots, s_n) : \mathbb{B}$

$\mid Op_{\mathbb{R}}(s_1, \ldots, s_n) : \mathbb{R}$

Listing 3: Booleans and operations in the language

With our language expanded, we can now trace away our control flow. The resulting cases are shown in Listing 4. It should be noted that even though we do not trace away basic operations in general, we can (and in fact have to) trace away operations either on Booleans, or producing Booleans, as we have no Boolean type in our trace. This effectively also means, that if we have some operations on real numbers, that only result in eventually producing some Boolean, these operations are omitted from the trace as well. If this seems odd, we must remind ourselves that the trace we produce can be interpreted as a "single-line program", and any part of the program that only exists to produce values that are irrelevant to the single-line program, should not be in the trace. It should also be noted that if we'd want to keep Booleans in our trace for some reason, this would still allow us to trace away control-flow, like the if-then-else statements, as they would still be irrelevant for the single-line program. However, what to do with the steps leading up to these would become a little more unclear. If the Booleans used by the if-then-else statements in our program are either part of the input or output of the program, they should probably also be present in the trace. However, if they are not, then they should probably still be discarded as being irrelevant to the single-line program.

Finally, we'd like to take a quick look at let-bindings. In general let-bindings are can be treated as lambda abstractions that are instantly applied. This allows us to streamline them a little more than we were able to with regular lambda abstractions in Listing 2. So, we can add let-bindings to our language in Listing 5, and their trace in Listing 6.

$$\text{trace}(\Gamma, \texttt{if}\ s\ \texttt{then}\ t_\top\ \texttt{else}\ t_\bot) \Rightarrow \texttt{if}\ \text{eval}(s)\ \texttt{then}\ \text{trace}(\Gamma, t_\top)\ \texttt{else}\ \text{trace}(\Gamma, t_\bot)$$

$$\text{trace}(\Gamma, Op_{\mathbb{B}}(e_1, \ldots, e_n) : \mathbb{B}) \Rightarrow (\text{eval}(Op_{\mathbb{B}}(e_1, \ldots, e_n) : \mathbb{B}), \emptyset)$$

$$\text{trace}(\Gamma, Op_{\mathbb{R}}(e_1, \ldots, e_n) : \mathbb{B}) \Rightarrow (\text{eval}(Op_{\mathbb{R}}(e_1, \ldots, e_n) : \mathbb{B}), \emptyset)$$

$$\text{trace}(\Gamma, Op_{\mathbb{R}}(e_1, \ldots, e_n) : \mathbb{R}) \Rightarrow (\text{eval}(Op_{\mathbb{R}}(e_1, \ldots, e_n) : \mathbb{R}), \{Op_{\mathbb{R}}(e_1, \ldots, e_n) : \mathbb{R}\} \cup \text{trace}(\Gamma, e_1) \cup \cdots \cup \text{trace}(\Gamma, e_n))$$

Listing 4: Control flow tracing

Types: ...

Terms:

$s, t :=$
...
| $\texttt{let}\ s : \sigma\ \texttt{in}\ t : \tau$

Listing 5: Adding let bindings

1.3 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, 3, and 5), as shown in Listing 7. 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 ($[\mathbb{R}] \to [\mathbb{R}]$).

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 isn't 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 8 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're 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.

$$\text{trace}(\Gamma, \texttt{let}\ s\ \texttt{in}\ t) \Rightarrow (\text{eval}(\texttt{let}\ s\ \texttt{in}\ t), \texttt{trace}(\Gamma, s) \cup \texttt{trace}(\Gamma \cup \{\text{eval}(s)\}, t))$$

Listing 6: Tracing let bindings

Types:

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

Terms:

$$\begin{aligned} s, t \coloneqq\ & \dots \\ \mid\ & [s_1 : \mathbb{R}, \dots, s_n : \mathbb{R}] \\ \mid\ & Op_{[\mathbb{R}]}(s) : \mathbb{R} \\ \mid\ & Op_{[\mathbb{R}]}(s) : [\mathbb{R}] \end{aligned}$$

Listing 7: Adding arrays

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

$$\begin{aligned} &\text{trace}(\Gamma, Op_{[\mathbb{R}]}([s_1, \dots, s_n]) : \mathbb{R}) \Rightarrow (\text{eval}(Op_{[\mathbb{R}]}([s_1, \dots, s_n]) : \mathbb{R}), \\ &\quad \text{trace}(\Gamma, Op^*_{\mathbb{R}}(s_1) : \mathbb{R}) \cup \dots \cup \text{trace}(\Gamma, Op^*_{\mathbb{R}}(s_n) : \mathbb{R})) \end{aligned}$$

$$\begin{aligned} &\text{trace}(\Gamma, Op_{[\mathbb{R}]}([s_1, \dots, s_n]) : [\mathbb{R}]) \Rightarrow (\text{eval}(Op_{[\mathbb{R}]}([s_1, \dots, s_n]) : [\mathbb{R}]), \\ &\quad \text{trace}(\Gamma, Op^*_{\mathbb{R}}(s_1) : \mathbb{R}) \cup \dots \cup \text{trace}(\Gamma, Op^*_{\mathbb{R}}(s_n) : \mathbb{R})) \end{aligned}$$

Listing 8: Tracing away arrays

We see in Listing 8 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 9. 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).

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

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

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

Listing 9: Keeping arrays in traces