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. Certain automatic differentiation (AD) implement the forward-pass as effectively tracing, and then perform the reverse pass on the trace[cite me]. Tracing is also used to speed up program execution by making assumptions on the program's execution path from the trace[cite me]. Another example? Rule of threes!

However, despite its ambivalence, tracing is rarely properly defined, or defined only for a specific use case. So, in this section we will 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 generally defined as a set of operations the input program performed on the inputs. 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're generally more interested in what happens to the data in our program, we can "trace away" functions and data structures. More precisely, if 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.

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

By choosing a subset of the types in our program, we can indicate which data structures

should be traced away. A common option is to keep only "grounded types", where we defined a grounded type as a type that isn't constructed of other types. Looking at our example in Equation 1, a trace keeping only these grounded 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 ungrounded and discrete types. Do 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 ungrounded 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. Why exactly? 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 $\sigma' \to \tau'$ 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 grounded types are a natural set of types to keep, as they are never constructed from other types.

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].

Finally, we can quickly check if our trace is correct by executing the trace and checking whether the result is the same as in the input program. We can also check the correctness of our tracing program by running it with a trace as its input program. Tracing a trace should return the trace itself, because if it returned some more minimal trace, we know that the first trace wasn't properly finished.

1.1 Tracing steps

We will now define some of the basic tracing steps for some arbitrary language as logical transformation rules. To do this we first define a very basic language on which we will operate. We will start by only defining the most basic types for our variables:

$\tau, \sigma \coloneqq \mathbb{R} \mid \sigma \to \tau$. This allows us to define a simple lambda calculus on real numbers:

$$\frac{}{\Gamma \vdash c : \tau} \text{ (Constant)}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \text{ (Variable)}$$

$$\frac{\Gamma \cup \{x : \tau\} \vdash e : \sigma}{\Gamma \vdash (\lambda(x : \tau).e) : \tau \to \sigma} \text{ (Abstraction)}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \sigma \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \sigma} \text{ (Application)}$$

Tracing through this lambda calculus is then fairly straightforward. We will define τ_T and σ_T as type indicators for the types of the trace, such that $\tau_T, \sigma_T \subseteq \tau, \sigma$. In reality this might mean we do not add our ungrounded types like $\tau \to \sigma$ to our traced types τ_T and σ_T. However, as stated before, as long as the types in τ_T, σ_T do not contain types that are not in τ_T and σ_T, they may contain ungrounded types as well.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{trace}(e) : \tau_T} \text{ (Tracing Introduction)}$$

$$\frac{\Gamma \vdash \text{trace}(c : \tau_T)}{\Gamma \vdash c : \tau_T} \text{ (Constant tracing)}$$

$$\frac{(x : \tau_T) \in \Gamma \qquad \Gamma \vdash \text{trace}(x)}{\Gamma \vdash x : \tau_T} \text{ (Variable tracing)}$$

$$\frac{\Gamma \vdash \text{trace}(\lambda(x : \tau).e : \tau \to \sigma)}{\Gamma \vdash \lambda(x : \tau_T).\text{trace}(e) : \tau_T \to \sigma_T} \text{ (Abstraction tracing)}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \sigma \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash \text{trace}(e_1\ e_2)}{\Gamma \vdash ((\text{trace}(e_1) : \tau_T \to \sigma_T)\ (\text{trace}(e_2) : \tau_T)) : \sigma_T} \text{ (Application tracing)}$$

It should be noted that tracing abstraction and application means tracing away the function type $\tau \to \sigma$. If we want to keep the function type in our final trace, we cannot trace away abstraction, and the application process would end with the trace of e_2, rather than the trace of $e_1\ e_2$.

We can then extend this with the basic operations we wish to preserve through our trace. We will denote these operations as $op(e_1, \ldots, e_n)$ with n arguments, for all functions $op : \tau' \to \sigma'$ where $\tau' \in \{\mathbb{B}, \mathbb{R}\}$ and $\sigma' \in \{\mathbb{B}, \mathbb{R}\}$. Of course, to ever make use of operations that either take in or produce Boolean values, we would need to extend our variable types with the Boolean domain $\mathbb{B}$. This means our new variable types become $\tau, \sigma \coloneqq \mathbb{B} \mid \mathbb{R} \mid \tau \to \sigma$. As long as we want to preserve these operations op in our trace they are generally homomorphic:

$$\frac{\Gamma \vdash e_1 : \tau', \ldots, e_n : \tau' \qquad \tau' \in \{\mathbb{B}, \mathbb{R}\} \qquad \sigma' \in \{\mathbb{B}, \mathbb{R}\}}{\Gamma \vdash op(e_1, \ldots, e_n) : \sigma'} \text{ (Basic operations)}$$

$$\frac{\Gamma \vdash \text{trace}(op(e_1 : \tau_1, \ldots, e_n : \tau_n) : \sigma) : \sigma_T}{\Gamma \vdash op(\text{trace}(e_1) : \tau_T, \ldots, \text{trace}(e_n) : \tau_T) : \sigma_T} \text{ (Trace on basic operations)}$$

We can also introduce control flow in the form of if-then-else statements. These will be eliminated by the tracing process, because when we are tracing, we've been provided with the inputs and can thus resolve the conditional and only follow the relevant branch.

$$\frac{\Gamma \vdash t_c : \mathbb{B} \qquad \Gamma \vdash t_\top : \tau \qquad \Gamma \vdash t_\bot : \sigma}{\Gamma \vdash \texttt{if } t_c : \mathbb{B} \texttt{ then } t_\top : \tau \texttt{ else } t_\bot : \sigma} \text{ (if-then-else introduction)}$$

$$\frac{\Gamma \vdash \text{trace}(\texttt{if } t_c : \mathbb{B} \texttt{ then } t_\top : \tau \texttt{ else } t_\bot : \sigma) \qquad t_c = \top}{\Gamma \vdash \text{trace}(t_\top : \tau) : \tau_T} \text{ (if-then-else tracing, true)}$$

$$\frac{\Gamma \vdash \text{trace}(\texttt{if } t_c : \mathbb{B} \texttt{ then } t_\top : \tau \texttt{ else } t_\bot : \sigma) \qquad t_c = \bot}{\Gamma \vdash \text{trace}(t_\bot : \sigma) : \sigma_T} \text{ (if-then-else tracing, false)}$$

Finally, we will also introduce let-bindings as a program structure. While let-bindings can be rewritten als lambda expressions, and then traced that way, it is actually a little simpler to trace let-bindings directly, as they are executed immediately anyways. This results in us technically tracing away the let bindings, but as the output of the tracing function can be interpreted as a list of let-bindings this remains only a technicality. This is probably too vague.

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \cup \{e_1 : \tau\} \vdash e_2 : \sigma}{\Gamma \vdash (\texttt{let } e_1 : \tau \texttt{ in } e_2 : \sigma) : \sigma} \text{ (let-binding introduction)}$$

$$\frac{\Gamma \vdash (\texttt{let } e_1 : \tau \texttt{ in } e_2 : \sigma) : \sigma}{\Gamma \cup \{e_1 : \tau\} \vdash e_2 : \sigma} \text{ (let-binding elimination)}$$

$$\frac{\Gamma \vdash \text{trace}((\texttt{let } e_1 : \tau \texttt{ in } e_2 : \sigma) : \sigma) : \sigma_T}{\Gamma \cup \{\text{trace}(e_1 : \tau) : \tau_T\} \vdash \text{trace}(e_2 : \sigma) : \sigma_T} \text{ (tracing let-binding)}$$

1.2 Array Tracing

Of course, the glaring omission from the lambda calculus described in Section 1.1 is data structures as present in modern programming languages. As an example, in this section, we will discuss tracing data in an array. We will also limit our array discussion to arrays of real numbers for now.

If we want to trace an array, there is an obvious naïve approach. Rather than tracing the array as a single data object, we instead trace every entry in our array as a loose variable. We can easily extend the example in Section 1.1 for lambda calculus:

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

$$\frac{v_1 : \mathbb{R}, \ldots, v_n : \mathbb{R}}{\Gamma \vdash [v_1, \ldots, v_n] : [\mathbb{R}]} \text{ (Array Introduction)}$$

$$\frac{\Gamma \vdash \text{trace}([v_1, \ldots, v_n] : [\mathbb{R}]) : [\tau_T]}{\Gamma \vdash [\text{trace}(v_1), \ldots, \text{trace}(v_n)] : [\tau_T]} \text{ (Array Tracing)}$$

The major disadvantage here is that we effectively deconstruct the array to trace it. Oftentimes, operations on arrays are applied to all or predefined subset of the items in the array. For instance if we want to square all the numbers in an array, we wouldn't need to see every item in the array being squared in the array. This is just extra information that probably makes our trace less useable. Instead, we might envision an entry in our trace where this we see the whole array being "squared", as this would convey almost the same information as having an entry for every item in the array.

Unfortunately, this isn't always possible. Imagine an array with the numbers one through ten, and a function that only squares an item in the array if it is a three. We immediately find that there is no way to have the trace of this function mapped on our array as a single entry in the trace. After all, the function does something different for threes than for other numbers. So in this case we might need to use our naïve approach from before.

This also highlights something about tracing functions mapped over arrays. If that function contains some branching (like an if-then-else statement checking for threes), we have to trace the application of the function to every item in the array individually. However, if the mapped function is the same for all items in the array (it does not branch), we would actually be able to trace it as if applying the function to the array rather than its items.

There's probably more to say about array tracing.