Paper

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

Abstract

Abstract.

1 Tracing

Tracing is a technique that is used a lot but often not particularly well-defined. The main idea and intuition behind tracing is that we take in a program and an input to said program, and output a list of operations the program used to produce the result it did. This list of operations can be seen as a program itself, sometimes referred to as a "single-line program"[cite me], to produce the result of the original program. The single-line referred to here, is a lack of branching paths in the program, thus eliminating all control-flow. This clearly highlights the first criteria for a tracing program: it must eliminate all branching.

Furthermore, as we aim to find only the main through line a program took for a set of inputs, we can also aim for eliminating any superflous structure in that program. This mainly means that we can eliminate functions, because we are just interested in what happens in the body of those functions. This then forms the basis for our second criterium: a tracing program must eliminate any superflous structure that we are not interested in. However, this criterium is still a little vague. For instance, we can imagine maybe tracing away things like tuples, arrays, or other collections of values. After all, we might only operate on the values, not on their collections. Where we draw the line on what structures we deem superflous will therefor impact the shape of our trace. However, we do know that destructuring collections into single values could result in an explosion to the size of our trace, depending on the size of the collection. Thus, it we might find it acceptable to leave these collections in. While we will not go into data collections in this section any further, it is important that the notion of "superflous" program structure, might change to meet our needs.

For now, our final criterium is about what the trace actually does. We have talked about eliminating control-flow, and eliminating functions, but what do we actually expect to keep intact? To actually achieve the single-line program we discussed, we must maintain any operations we actually perform on the inputs or other data that are relevant to our output. This comes down to preserving all mathematical and Boolean operations.
In summary we set the following criteria for a tracing program:

1. All branching/control-flow is eliminated.
2. All superflous structure (like functions) is eliminated.
3. We record all basic mathematical and Boolean operations.

To actually implement the trace we can view tracing as a source code transformation from a regular program (and input) written in a source language to a traced program in a target language. We define the source language in Listing 1. Here we have basic lambda calculus with added control flow in if-then-else statements, and room for variable references. We denote any term $s \in S$ with as $s : D$ if we require the term to evaluate to the domain D. It should also be noted that we have collected all our basic operations as $op(s_1, \cdots, s_n)$ for any operation of the form: $R_n \to R$ ($+$, $-$, etc.), $R_n \to \{\top, \bot\}$ ($=$, $>$, etc.), and $\{\top, \bot\}_n \to \{\top, \bot\}$ ($=$ or $\neq$ on Booleans).

Values:

$V \coloneqq \mathbb{R} \cup \{\top, \bot\}$

Terms:

$S \coloneqq \{$

- $s_1\ s_2$, *(Application)*
- $v \in V$, *(Bare real numbers or Boolean values)*
- $\lambda(x : V).s$,
- $\texttt{let}\ \ s_1 : V\ \ \texttt{in}\ \ s_2$,
- $\texttt{if}\ \ s_? : \{\top, \bot\}\ \ \texttt{then}\ \ s_\top\ \ \texttt{else}\ \ s_\bot$,
- $op(s_1 : V, \cdots, s_n : V)$,
- $x : \text{String}$ *(Variable reference)*

$\}$

Listing 1: Source Language

We continue by defining the target language in Listing 2. We see here that we've lost most of our terms in accordance to our three criteria: getting rid of functions means we can get rid of lambda's, and getting rid of branching means getting rid of if-then-else statements. Also, let-bindings will be easily resolved in the next steps. We should also note that the values of W are just the values of V, but now with an additional string that serves as an identifier to the value. This is so that when we have our values in our trace, we can keep track where they came from. We'll we assume that $V \Leftrightarrow W$, such that we can lift values in V to W and vice-versa. In reality this just means that a value that is created in the computation (and thus has no name), will get a generated name to keep track of it.

So, all that remains is to transform our source language into our target language through tracing. We can do this by looking at each of our source terms and transforming them into target terms. To do this, we define need to define an environment $\Gamma : \text{String} \to W^*$, here W^* is defined as $W^* \coloneqq W \cup \{W \to W\}$. W^* requires functions $W \to W$ to resolve lambda functions, as we will see. We will also define the function $\text{eval} : \Gamma \to S \to V$ as our evaluation function, and $\text{trace} : \Gamma \to S \to (W, [(\text{String}, T)])$ as our tracing function, where the tracing function returns both the value of the expression provided, and the trace of that expression as a list of tuples of named steps. This then finally allows us to create the program transformation in Listing 3.

In Listing 3, we see the program transformation from the regular program in the source

Values:

$W \coloneqq \{(\text{String}, \mathbb{R}), (\text{String}, \{\top, \bot\})\}$

Terms:

$T \coloneqq \{$
$\quad w \in W\,,$
$\quad op(t_1 : W, \cdots, t_n : W)\,,$
$\quad x : \text{String}$
$\}$

Listing 2: Target Language

language to the traced program in the tracing language. We represent the return value of the trace function as a tuple consisting of a value and a unsorted list of operations performed. It should be noted that this unsorted list has all its values as tuples consisting of an identifier and the actual trace step. This is why this list can stay unsorted, as all values that are referred to by certain steps are named as such in this list. This might also help us implementing parallelism on this trace later on.

$\text{trace}(\Gamma, s_1\ s_2) \Rightarrow (w_a, t_1 \cup t_2 \cup t_a)$
$\quad (w_1, t_1) \Leftarrow \text{trace}(\Gamma, s_1)$
$\quad (w_2, t_2) \Leftarrow \text{trace}(\Gamma, s_2)$
$\quad (w_a, t_a) \Leftarrow w_a(w_2)$

$\text{trace}(\Gamma, v \in V) \Rightarrow (v, \emptyset)$

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

$\text{trace}(\Gamma, \texttt{let}\ s_1\ \texttt{in}\ s_2) \Rightarrow (w_2, t_1 \cup t_2)$
$\quad (w_1, t_1) \Leftarrow \text{trace}(\Gamma, s_1)$
$\quad (w_2, t_2) \Leftarrow \text{trace}(\Gamma \cup w_1, s_2)$

$\text{trace}(\Gamma, \texttt{if}\ s_? \ \texttt{then}\ s_\top\ \texttt{else}\ s_\bot) \Rightarrow (w_a, t_? \cup t_a)$
$\quad (w_?, t_?) \Leftarrow \text{trace}(\Gamma, s_?)$
$\quad (w_a, t_a) \Leftarrow \texttt{if}\ w_?\ \texttt{then}\ \text{trace}(s_\top)\ \texttt{else}\ \text{trace}(s_\bot)$

$\text{trace}(\Gamma, op(s_1, \cdots, s_n)) \Rightarrow (w_{op}, \{op(w_1, \cdots, w_n)\} \cup (t_1 \cup \cdots \cup t_n))$
$\quad (w_1, t_1) \Leftarrow \text{trace}(\Gamma, s_1)$
$\qquad \vdots$
$\quad (w_n, t_n) \Leftarrow \text{trace}(\Gamma, s_n)$
$\quad w_{op} \Leftarrow \text{eval}(op(w_1, \cdots, w_n))$

$\text{trace}(\Gamma, x) \Rightarrow (x \in \Gamma, \emptyset)$

Listing 3: Tracing transformation