A Visual Notation for Succinct Program Traces

Divya Bajaj
Oregon State University
bajajd@oregonstate.edu

Martin Erwig
Oregon State University
erwig@oregonstate.edu

Danila Fedorin
Oregon State University
fedorind@oregonstate.edu

Kai Gay
Oregon State University
gayk@oregonstate.edu

***Abstract*—Program traces are often used for explaining the dynamic behavior of programs. Unfortunately, program traces can grow quite big very quickly, even for small programs, which compromises their usefulness. In this paper we present a visual notation for program traces that supports their succinct representation, as well as their dynamic transformation through a structured query language.**

An evaluation on a set of standard examples shows that our representation can reduce the overall size of traces by more than 80%, which suggests that our notation is an effective improvement over the use of plain traces in the explanation of dynamic program behavior.

***Index Terms*—program trace, ellipsis, query language**

I. Introduction

To understand the dynamic behavior of programs, programmers regularly inspect program traces that keep track of the computation performed by a program on particular inputs. One important use of program traces is in the context of debugging, where they are used to understand unexpected program behavior. In addition, program traces are also used in educational settings to illustrate the working of programs to novice programmers. Either usage scenario is plagued by the fact that program traces tend to be large, even for rather small programs, which makes it often difficult and time-consuming to isolate those parts of a trace that are relevant for the task at hand: finding a bug or understanding a particular part of a program.

Our ultimate goal is to support a variety of users of traces (programmers, educators, learners) with a tool for the effective creation of targeted, succinct, and adaptable traces. To this end, we have investigated existing tracing approaches and, in particular, the trace representations they employ. Based on this analysis we have designed a new representation as a basis for such a tool that offers a number of innovations. In this paper we explain the design of our representation and discuss its features in relation to other approaches.

Our approach is based on the idea of applying a set of modular filters to automatically created complete traces. By using different sets of filters users can quickly customize traces to their needs. As we will show, the set of predefined filters supports a wide variety of customizations and is sufficient for many use cases. Still, new filters can be added as needed, since the filters are defined based on a trace query language. The definition of new filters using the query language is meant to be done by experts, whereas the use of filters does not require any understanding of the query language.

As the basis for our research we use traces for a small functional language (an extension of the untyped lambda calculus by numbers and algebraic data types). As an example, consider the following definition of a factorial function.

```
fact = \x -> case x of {0 -> 1; y -> x * fact (x-1)}
```

A major design decision for how to represent program traces is whether to employ linear sequences or trees of expressions. Before we present our approach, we briefly discuss a widely known form of linear traces in Section II to identify a number of aspects and issues that affect the design of any trace notation. This helps us motivate some of the design decisions for our tree-based trace representation, which we introduce in Section III. A key insight of our work on trace notation is that the construction of succinct traces requires a set of expressive filters that offer fine-grained control over the information presented in traces. We will discuss trace filters in Section IV. After we have presented our trace model, we provide a systematic comparison of the different trace models and illustrate their strengths and weaknesses in Section V. We then present an artifact-based evaluation of our approach in Section VI. Finally, we discuss related work in Section VII and present conclusions and our plans for future work in Section VIII.

II. Linear Traces

Linear traces are often used in introductory functional programming textbooks as explanations for how particular function definitions work [3], [12]. The simplicity of linear traces is very appealing, and they generally are quite effective in demonstrating computation, especially for small examples, as the sequential ordering of expressions makes them easy to follow. As an example, consider the linear trace for the computation of `fact 6`, which can take the following form.

```
fact 6
= case 6 of {0 -> 1; y -> 6 * fact (6-1)}
= 6 * fact 5
= 6 * (case 5 of {0 -> 1; y -> 5 * fact (5-1)})
= 6 * (5 * fact 4)
...
= 6 * (5 * (4 * (3 * (2 * (1 *
      (case 0 of {0 -> 1; y -> 0 * fact (0-1)}))))))
= 6 * (5 * (4 * (3 * (2 * (1 * 1)))))
= 720
```

We have omitted a number of intermediate steps from the complete trace, since they don't contribute anything new to an understanding of how the computation unfolds. Obviously,

978-1-6654-4592-4/21/$31.00 ©2021 IEEE

these include the sequence of steps replaced by the ellipsis, but the trace also elides details about the substitution of arguments for parameters, comparisons of values, and basic arithmetic computations. This observation suggests the need for filtering automatically produced traces, which raises questions regarding describing and applying such filters.

One approach is to offer an interactive GUI for selecting ranges and applying simple *hide* and *unhide* operations. Another approach is to programmatically specify filters and the target ranges on which they operate using some form of query language.

In a linear trace, an interactive approach might be as simple as selecting a range of lines. However, even this seemingly simple operation could turn out to be quite cumbersome when this range is large. Moreover, the need to select multiple disconnected ranges makes the approach even less attractive. Finally, when traces are generated repeatedly for different inputs, the need to repeat filtering operations by hand might be prohibitive. Therefore, a query approach to applying trace transformations seems to be more effective and preferable over a simple GUI interface, which reveals an important weakness of linear representations, namely, the difficulty in specifying the scope and effect of trace transformations.

Another feature of linear traces that is simultaneously a strength and a potential weakness is the way variable bindings are handled. Formally, when a function is applied to an argument, as in `fact 6`, a binding between the function parameter and the argument is created (in this case `x=6`). Then each reference to the parameter in the function body is replaced by the bound value. Linear traces typically don't show any bindings between function parameters and arguments and instead directly substitute arguments for all parameter references. The advantage of this approach is that no environments (that is, list or stacks of bindings) need to be maintained at all, which helps to keep the traces small and manageable. However, this approach turns into a disadvantage in situations where many parameter references have to be substituted by an argument that takes up a large amount of space. The following example illustrates this case. Consider a function `map1to6`, which maps an argument function to the list of numbers from 1 to 6.

```
map1to6 f = [f 1,f 2,f 3,f 4,f 5,f 6]
```

When we apply `map1to6` to a function with a large body, this function definition will be copied 6 times.

```
map1to6 (\x->some-big-expression)
= [(\x->some-big-expression) 1,
   (\x->some-big-expression) 2,
   (\x->some-big-expression) 3, ...]
```

This makes traces hard to read. In situations like these, keeping a name in the trace together with the binding information that shows the large expression only once is a more economic representation.

A related problem is the inability to factor out, and represent only once, subtraces for common subexpressions. Consider, for example, the trace for `fact 6 + fact 5`. A linear trace would contain the trace for `fact 5` twice, which would not have any additional explanatory value and would only decrease the readability of the trace. Representing the subtrace for a shared subexpression such as `fact 5` as a shared subtrace would be preferable in all cases.

The factorial trace also illustrates that the linear presentation generally has to produce many parentheses to express the order in which subexpressions have to be evaluated. This lexical overhead can be quite annoying and make the reading of traces more tedious. The need for bracketing is an intrinsic problem for any linear representation that a tree representation doesn't have, since grouping is expressed implicitly through the structure of the subtrees. Of course, a tree representation has its own disadvantages, including the need for good layout algorithms as well as generally requiring more space.

Finally, we point out a more subtle, but quite important, disadvantage of linear traces: The fact that each trace step consists of a complete expression makes it difficult to systematically isolate (and highlight or hide) the evaluation of subexpressions. Here a tree representation can be more modular, in particular when it contains not just expressions but evaluation judgments. This aspect will become clear after we have explained how the factorial trace looks in our visual representation, which we will do in the next section.

III. Non-Linear Traces from Proof Trees

We can obtain a structurally different alternative to a linear trace when we expose the hierarchical tree structure of expressions and show the evaluation of each subexpression. Such a tree of subexpressions and their results looks essentially like a proof tree obtained through the application of an operational semantics, which is an idea already presented in [6], albeit in a different context and with a different goal. Operational semantics define the evaluation of expressions through a set of inference rules that have the following form.

$$\frac{P_1 \quad \ldots \quad P_n}{C}$$

The statement C is a conclusion that follows if all the premises $P_1, \ldots, P_n$ are true. In so-called *big-step operational semantics*, the premises and conclusions are of the form $\rho : e \Downarrow v$ that say that expression e evaluates to the value v in the context of an environment of variable bindings ρ.

The semantics for our language are a fairly standard call-by-value operational semantics [10].[1] To give a few brief examples, the rule for evaluating constants c has the form $\rho : c \Downarrow c$; it has only a conclusion and no premises and says that each constant always evaluates to itself. Such rules without premises are also called *axioms*.

The rule for evaluating expressions e_1 *op* e_2 that involve a built-in binary operations *op* requires the evaluation of both

[1]This is actually not completely true: Our semantics rules maintain and sometimes use names for functions and otherwise eliminate variable environment and replace them by binding nodes. We will explain these details later in the paper.

argument expressions to values v_1 and v_2 and yields as a result the value v obtained from applying *op*.

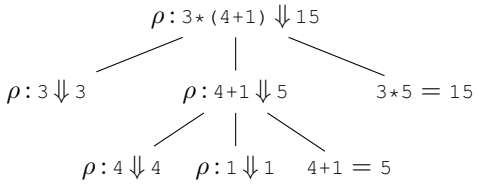
$$\frac{\rho : e_1 \Downarrow v_1 \qquad \rho : e_2 \Downarrow v_2 \qquad v_1 \; op \; v_2 = v}{\rho : e_1 \; op \; e_2 \Downarrow v}$$

Finally, the rule for evaluating the application of expression e_1 to another expression e_2 requires that e_1 evaluate to a function value (a lambda abstraction `\`x`->`e') and that e_2 evaluate to a value v'. The result of the application is then obtained by evaluating the defining expression of the function, e' in the environment ρ that is extended by the binding of the function parameter x to the result of the argument expression v'.

$$\frac{\rho : e_1 \Downarrow \backslash x \texttt{->} e' \qquad \rho : e_2 \Downarrow v' \qquad \rho, x{=}v' : e' \Downarrow v}{\rho : e_1 \; e_2 \Downarrow v}$$

With such rules we can build a tree for the evaluation of an expression e as follows. First, we find a rule whose conclusion matches e, which produces bindings for the metavariables used in the rule. For example, to evaluate `fact 6` we have to use the rule for application, which binds e_1 to `fact` and e_2 to `6`. The environment ρ is bound to the current set of definitions, which must contain a definition for `fact`. With these bindings we then instantiate all premises of the rule. In the example, we obtain the premise instances $\rho : \texttt{fact} \Downarrow \backslash x \texttt{->} e'$ and $\rho : \texttt{6} \Downarrow v_2$. (No metavariable in the third premise is instantiated yet.) We then continue to find rules with matching conclusions for each premise. In the example, a rule for looking up variable bindings will locate the definition of `fact` in ρ and create corresponding bindings for x and e', and the application of the constant axiom binds v_2 to `6`. Both of these rules create leaves in the tree. With these new bindings we can now find a rule for evaluating the third premise to evaluate the function body of `fact` in the environment in which the parameter `x` is bound to `6`. This process continues recursively and ends with the application of axioms creating leaves. The complete evaluation tree for `fact 6` is too big to be of practical use and needs to be trimmed down further as discussed in Section IV.

To continue the discussion of the tree representation, we therefore consider for now a simpler example: the tree representing the evaluation of the expression `3*(4+1)`. This tree can be obtained by applying the rule for binary operations twice and the axiom for constants three times.



The root of the tree contains the judgment with the expression to be evaluated and its result; its children contain the judgments for the evaluation of the subexpressions. An important feature of the hierarchical tree structure is that it allows the viewer to decide which of the subexpressions (if any) to follow, and in which order. This also means that a user interface for exploring such trees can hide subtrees independently of one another, which is the modularity property mentioned in Section II. Certainly, this feature could also be considered a drawback, since it requires the user to decide which subtree to focus on next.

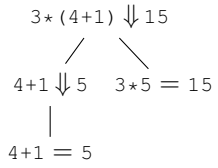
Compared to the following linear trace for evaluating the expression `3*(4+1)`, the tree requires more space and seems overly complex.

```
3*(4+1)
= 3*5
= 15
```

Maybe the tree representation isn't such a good idea after all? The tree representation is larger because it mentions details that are omitted in the linear representation, such as the evaluation of constants and the variable environment.

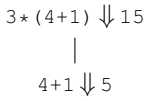
This is one area in which our trace notation deviates from generic proof trees. First, we eschew environments from judgments and replace variable lookups where necessary through so-called *binding nodes*; second, we provide a number of filters to eliminate, automatically or on request, judgments from the tree that are deemed unnecessary by the user.

After removing environments and filtering out all constant evaluation judgments, the tree trace for the arithmetic example becomes already much simpler.



Note that the tree trace notation suggests the reading of each node as a statement justified by the statements of its children. For example, the root node reads: "3*(4+1) evaluates to 15 because 4+1 evaluates to 5 and because 3*5 is 15."

If we also forgo the presentation of arithmetic facts, we obtain an even simpler tree with a complexity similar to the linear trace.



This example is of course not very exciting; we have used it to illustrate the basic design that underlies our tree trace notation. In the following section we show how to construct succinct tree traces through the judicious use of trace filters.

IV. Trace Filtering

The complete trace for the `fact 6` example consists of 80 nodes and 22 levels, which is a lot of information to slog through. However, as the example in Section II illustrates, the essence of the computation can be captured in a much smaller trace by omitting details and some repeated structures. Specifically, one might expect a trace to execute all parts of a definition once, but generally not more than that. One might also want to filter out some arithmetic computations (for example, for decrementing a counter) and the lookup of variable bindings. We call such a tailored trace a *trace view*.

In Figure 1 we show a trace view that meets these expectations.[2] The trace view is similar to the linear trace presented in Section II and is obtained from a complete trace in several steps through the application of filters.

[2]The LaTeX code for the trace views in this paper was generated by our prototype implementation, with occasional manual adjustment of the horizontal positioning to fit the paper layout.

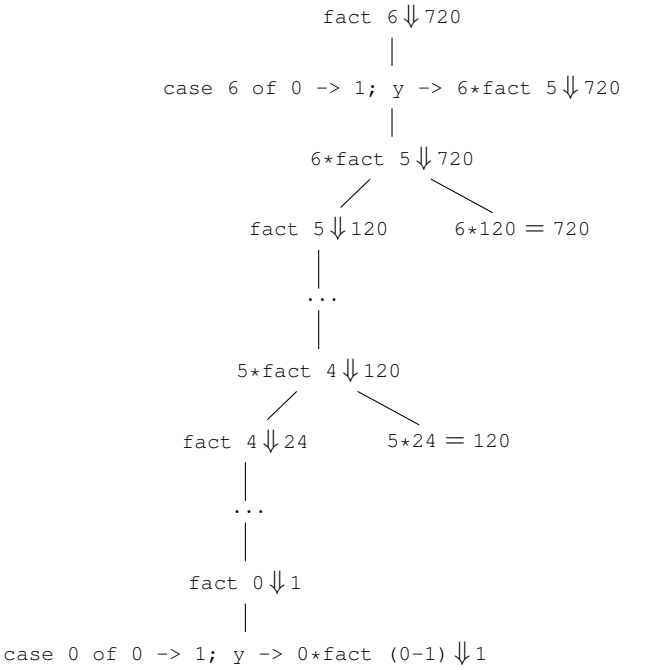
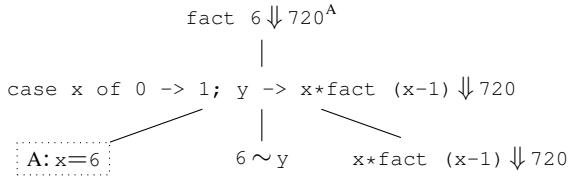


Fig. 1. Trace view for `fact 6` with many details hidden. Ellipses indicate elided paths with potentially off-branching subtrees.

The modularity of these filters as well as the flexibility in defining them makes our trace representation highly adaptable to different situations and needs. In the following we use the example trace to illustrate the definition and use of trace filters and the principles that underlie our trace filtering approach.

A. Binding Nodes

We start by discussing a widely applicable trace simplification, also used in this example: the hiding and propagating of variable lookups. As can be seen in Figure 1, the trace does not contain environments even though bindings of variables are created and used repeatedly. In an expanded version of the trace, the top part would actually look as follows.



We observe that the node evaluating the `case` expression is shown in its unsubstituted form with `x` instead of `6` and `x-1` instead of `5` and that it now has three children. The first child is a binding node that explains that `x` was bound to `6` in the root node (which has been assigned the label A). The second node shows a simplified version of pattern matching judgments, which says that `6` matches `y` (that is, the second case applies) to justify the selection of the third premise. In general, pattern matching judgments produce bindings, but they need not be shown when they are not used (as is the case here). The third child contains, as in Figure 1, the recursive application of `fact`, but again using `x` and `x-1` instead of the substituted values.

Binding nodes are an important innovation of our trace model. Motivated by the need for small traces, they keep the sharing property of bindings offered by environments without having to repeat environments in every judgment. Binding nodes interact in interesting ways with filters, since they show information about the origin of the bindings. This origin information may change in some cases, which requires a careful definition of filter semantics.

B. Global Filters

To hide a set of nodes from a trace, we need a flexible way of expressing which nodes to hide. To this end, we have defined a concept of *selectors*, which are combinations of syntactic patterns that use wildcard symbols. The selector that matches all variable lookups, for any variable and value, is $\diamond = \diamond$,[3] but we can also use more specific patterns such as `x` $= \diamond$ to hide only bindings for the variable `x`.

We can use such patterns directly or via assigned names in operations for hiding and propagating. While *hiding* simply removes all the nodes that match the pattern from the trace, *propagating* also uses the pattern (when possible) as a rewrite rule. In the case of variable lookups this means to replace variables within their scope by their values. We have hidden and propagated all variable bindings in the trace in Figure 1, and we have also hidden all pattern matching evaluations.

Another class of uninteresting steps that are often hidden from traces are reflexive judgements, which include specifically axioms for evaluating constants, as well as simple arithmetic operations, especially increments and decrements by 1. These filters eliminate judgments such as `1` ⇓ `1` and `6-1` ⇓ `5` from the trace, as was done in Figure 1. Applied filters can always be selectively deactivated to temporarily show the hidden information. Repeated hiding and unhiding also can help users to understand the effect of filters "by example."

C. Selective Filters

Note that the trace illustrates that we can also hide only a specific subset of nodes that match a pattern. One example is `case` expressions of which we show only the first and last occurrence to illustrate the two different situations (base case and recursive case) covered. The other example is hiding most of the intermediate recursive calls of `fact`. Specifically, we have filters that allow the hiding of all but the first k recursive calls and/or the last recursive call.

We will discuss the set of available filters in Section VI.

While omitting leaves or whole subtrees does not affect the rest of a tree, the omission of intermediate parts from a tree requires some notation to connect the parts that are separated by the cut. We use ellipses "$\cdots$" to indicate places where paths (with potential off-branching subtrees) have been omitted.

D. From Trees to DAGs

To exploit the fact that any specific expression has to be evaluated only once, we have extended the tree notation to

[3]The node label is not part of the syntax for bindings and thus not part of the pattern.

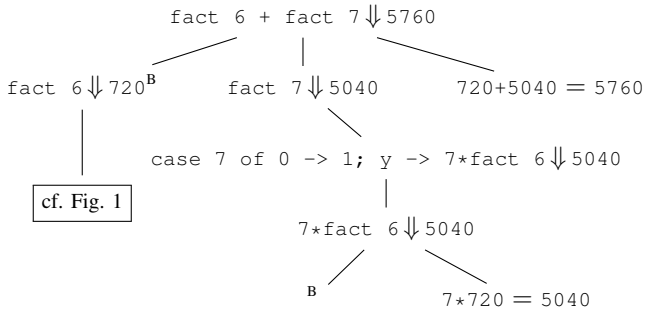


Fig. 2. Trace view for `fact 6 + fact 7`. DAGs, represented by using node references, are used to avoid the repetition of subtraces.

DAGs. This idea is not new: in their paper on PROOFTOOL, Dunchev et al. [4] mention the need to avoid the overlapping edges that are associated with their DAG-based proofs. The workaround they choose is to copy shared subgraphs to every place where they are referenced. However, it is our view that repeated copies of the same subtrace are merely noise in a trace, and should therefore be avoided, since they incur the additional cost for users of having to recognize and deliberately ignore the copy. Explicitly performing sharing between such subtraces and displaying them only once can automatically simplify traces in cases when the same expression is evaluated multiple times.

As an example, consider the evaluation of the expression `fact 6 + fact 7`. The expression `fact 6` has to be evaluated a second time in the first recursive step of evaluating `fact 7`. Instead of recreating the whole subtrace, our tool will produce a reference to the root node (labeled with B) of the already existing subtrace, see Figure 2.

E. Trace View Prototype

We have already mentioned that we have implemented a prototype for producing and filtering trace views. The main purpose of this tool is to allow us to investigate properties of traces and trace filters. It has also helped to develop the query language that is used to define all filters. The current implementation provides a command-line interface as part of the Haskell GHCi interpreter that interacts via the DOT format with a viewer. The tool allows a user to (un)apply individual filters and can also load scripts containing groups of filter definitions. In addition, the tool also allows the step-by-step customization of traces by targeting individual nodes (and ellipses) with filters. The tool can export traces in LaTeX, and all the trace figures in this paper have been generated by the tool (some have been adjusted manually afterwards).

V. CLASSIFICATION OF TRACE MODELS

Tracing approaches can differ in several ways. In addition to the notation aspects, the flexibility of manipulation traces plays an important role in their effectiveness. To get a better understanding of the existing approaches, we can organize the design space along several orthogonal dimensions and classify the approaches accordingly. The decisions each trace representation has to make are the following.

- **Structure**: linear, trees, or DAGs
- **Handling of Binding Information**: using environments and variable lookups or substitutions
- **Control over Trace Simplification**: full, limited, or none
- **Scope of Filters**: global or individual nodes
- **Domain**: dynamic program behavior or proofs

We have already discussed the structure and binding information aspects in Sections II and III, and the classification of the different approaches in these regards are shown in Table I.

Approach	Form	Handling of Bindings	User Control	Scope	Domain
[3] [12]	linear	substitution	full	by node	execution
[2]	linear	substitution	none	global	proofs
[7]	linear	environment	none	global	execution
[4]	tree	environment	limited	global	proofs
[9] [11] [1]	tree	environment	limited	global	execution
[5]	digraph	substitution	full	by node	proofs
This paper	DAG	binding nodes	full	global	execution

TABLE I
CLASSIFICATION OF TRACE NOTATIONS

Another important question of any trace model is how to control the size of traces. We can distinguish two aspects that are relevant in this regard. On the one hand, does the trace creator have any control over the information to be included in the trace and thus can control the size of the trace? On the other hand, if a user can exert control to omit trace information, do decisions apply to single nodes only or more globally to a range of nodes?

The linear traces found in textbooks are carefully crafted and, by definition, under full control over what is represented, but decisions to present or omit steps are made line-by-line. This approach is very flexible, but doesn't scale well, and it is not automated. The linear traces generated by [7] for concurrent program execution, are simplified using a heuristic algorithm that applies 3 different simplification operations repeatedly on the trace in a particular order. Thus, it does not provide users with any control over these transformations.

The slicing approaches of [9], [11], and [1] give users some indirect control through the selection of partial input/output. These user actions affect the trace globally.

The approach presented in [5] represents traces for proof systems as directed bipartite graphs (denoted as digraphs in Table I). This system give users full control over traces through the collapsing of related nodes. A GUI lets users select nodes and perform transformations on these nodes. Thus, the user has complete control over the transformations, but transformations are applied on only individually selected nodes.

Finally, we also show the domain for which each of the trace approach is intended in Table I. This information puts some of the design decisions into perspective. In particular, traces of proofs and traces of programs can benefit from some of the same operations, but the method in which they are generated

differs: proof trees are often "built" interactively by the user during a proof session.

VI. Artifact Evaluation

To evaluate the effectiveness of our approach we have created trace views for 21 mostly well-known example programs that we regularly employ as teaching material. The programs are grouped into different sections: functions on numbers (*fact*, *twice-fact*, and *collision*), lists (*reverse**, *replicate*, *sum*, *filter*, *merge-lists*, *cart-product*, *subsets*, and *quicksort*), (binary search) trees (**BST* and *constants*), and language processing (*eval**, *typecheck*, and *fold-constants*). We have measured the size reductions that can be achieved and the filters that needed to be employed to create the trace views. In the following we describe the details of this experiment.

A. Systematic Creation of Trace Views

Among the most well-known results in psychology is Miller's demonstration that humans can only hold a small number of "chunks" in their short-term memory [8]. Although subsequent research has shown that the situation is more complicated, the general trend remains: our short-term memory is quite limited. In accordance with this, our approach was to reduce (fairly aggressively) the number of nodes in the trace that do not significantly contribute to the explanation of the program. Rather than cluttering the reader's mind with the details of additions and environment lookups, or the trivial reflexive evaluations necessitated by the semantics, we focus on the control flow.

The creation of the traces was guided by a set of rules that we have established based on observations and our experience with working with traces. Specifically, over the course of many months spent on analyzing trace notations and developing our prototype we have noted a number of patterns in our interaction with traces (with respect to hiding and propagating information) from which we have derived a set of rules that have proved repeatedly useful in focusing traces on the most relevant information. The creation of the trace views has been guided by these rules.

We group these rules into two groups. The first group consists of quasi-universal rules that correspond to standard filters and will always be applied by default to any created trace. The second group consists of rules that apply only in some situations. Since we have formalized these rules through the definition of filters, we can be sure that they are followed strictly and systematically.

To decide which filters to apply in which situation, the trace views were initially created by one of the researchers and then reviewed and critiqued by 2-3 other members of the research group, which sometimes led to a revision of the applied filters and resulting traces. A threat to the validity of this experiment is of course the potential bias of the group assessing the filters. Different groups may come to different results, but we believe that these differences would be minor and not change the overall picture significantly. Furthermore, the general principle of our approach remains: users have the flexibility to customize traces differently, yet succinct traces remain always an option.

B. Filter Collections

Next we describe the filters we have used in our experiment. All filters can be divided into two categories: those that are applied to all of our example trace views, and those that are only applicable in some cases. Table II shows for which trace views these filters were used. We also classify filters by scope, that is, while in general each filter is applied globally, some filters are applied in an intelligent way hiding or propagating only in some places. One example is the recursion filter, which hides only intermediate recursive calls. Such filters are identified by a trailing $^\circ$ symbol. These filters cannot be defined by a simple syntactic pattern, but require a more sophisticated query. Finally, some filters are parameterized by name or value. We report this information as well and indicate it by a subscript $_f$.

In the following we list all filters that we used. Many are simply a straightforward hiding of a specific syntactic category. For some more interesting filters we add a brief explanation. We start with filters that are used in every example.

- Reflexive hides all of the constant evaluations.
- PatMatch hides all pattern matching judgements.
- PartialApp hides all partial function applications and all of their descendants. Partial function application is identified by patterns of the form `f x`$\Downarrow$`\y -> e`; however, some additional care is required to avoid hiding the subtrees corresponding to the evaluation of the function's arguments.
- FunDef$_f$ filter hides top-level function declarations. Since the filter is parameterized, we can instantiate it to hide several different declarations. The minimal functional language on which our prototype implementation is based requires function definitions to be given as `let` expressions. In an implementation that stores function definitions in separate program files, this filter would not be needed. Note that we could achieve the same effect with a non-parameterized filter, but parameterization gives us the flexibility to hide only some declarations while keeping others (for instance, definitions of constants).
- LimitRec$^\circ_f$ is a parameterized filter that hides all of the intermediate applications of the function supplied as an argument to this filter. We use this filter to show the first two and one last function application of recursive functions.
- The OuterCase filter hides all `case` expressions that have another `case` expression as one of their immediate children. For example, the filter will hide a node containing `case e of p -> case e' of ds; cs`, since it has a child containing `case e' of ds`. OuterCase will not hide the latter node if `e'` and `ds` don't contain any `case` expressions.
- Binding hides all binding nodes but is also used to propagate the values to where they are used in the trace.

Program	$\#T$	$\#T^*$	$\Delta\%$	$\updownarrow T$	$\updownarrow T^*$	$\Delta\%$	$\overleftrightarrow{T}$	$\overleftrightarrow{T^*}$	$\Delta\%$	$\#F$	Additional Filters
factorial	80	7	91	22	8	64	3	2	33	1	DEC
twice-fact	70	13	81	13	10	23	3	2	33	1	DEC
collision	532	8	98	25	8	68	3	2	33	3	CASE, COND, TRIVIAL$_f$
reverse	203	9	96	21	13	38	3	1	67	1	TRIVIAL$_f$
reverse-accum	89	4	96	13	8	39	3	1	67	1	CASE
replicate	97	6	94	20	8	60	3	1	67	1	DEC
sum	148	7	95	34	8	77	3	2	33	0	-
filter	127	9	93	20	11	45	3	2	33	1	TRIVIAL$_f$
merge-lists	140	8	94	25	12	52	3	1	67	1	COND
cart-product	242	10	96	20	9	55	4	2	50	1	TRIVIAL$_f$
subsets	359	13	96	23	12	48	3	2	33	2	TRIVIAL$_f$, CASE
quicksort	500	22	96	28	13	54	4	2	50	1	TRIVIAL$_f$
search-BST	71	5	93	10	8	20	4	1	75	2	CASE, COND
insert-BST	115	8	93	18	12	33	4	1	75	2	CASE, COND
delete-BST	167	15	91	24	16	33	4	2	50	1	COND
inorder-BST	70	7	90	11	9	18	4	1	75	0	-
constants	239	17	93	18	12	33	4	3	25	2	CASE, TRIVIAL$_f$
eval-expr	158	13	92	17	10	41	4	3	25	0	-
eval-fun	226	13	94	20	11	45	3	3	0	1	TRIVIAL$_f$
typecheck	127	20	84	16	14	13	4	3	25	0	-
fold-const	133	9	93	13	10	23	4	2	50	2	ADD, TRIVIAL$_f$

∀ **Hide**
REFLEXIVE
PATMATCH
FUNDEF$_f$
PARTIALAPP
LIMITREC$^\circ_f$
OUTERCASE
∀ **Propagate**
BINDING

∃ **Hide**
TRIVIAL$_f$
CASE
ADD
COND
∃ **Propagate**
DEC

TABLE II
SIZE OF TRACES (T) AND TRACE VIEWS (T^*) (#: NUMBER OF NODES, $\updownarrow$: DEPTH, $\leftrightarrow$: WIDTH) AND SPACE SAVINGS ($\Delta\%$) ACHIEVED. COLUMN $\#F$ SHOWS THE NUMBER OF ADDITIONAL FILTERS NEEDED, AND THE LAST COLUMN LISTS THOSE ADDITIONAL FILTERS.

In addition to these universal filters, some filters are used only for creating some of the trace views.

- CASE is used to simply hide all `case` expressions. It can be used to keep all control flow decisions out of a trace, which is sometimes useful. For example, we could filter the remaining `case` expressions from the trace view shown in Figure 1 and still get a useful illustration of the computation.
- TRIVIAL$_f$ hides the evaluation of function applications (such as `10 > 0`) whose behavior is well understood. Even though one could argue for placing all TRIVIAL$_f$ filters into the category of always-applied filters (at least for a single user), there are functions that we may want to explain separately, but then assume them to be understood when used elsewhere. One example is the list function *filter*, which has its own trace view, but is considered trivial when used as part of the trace for *quicksort*.
- Finally, we have a number of very specific filters that help the customization of trace views. For the set of example programs these are filters for hiding decrementing a variable (DEC), additions (ADD), and simple comparisons (COND). With DEC, we also propagate the values.

A summary of the filters is shown on the right of Table II.

C. Results

Table II summarizes the size information for the examples. The first three columns show the sizes of the original traces T and trace views (T^*) as well as the percentage size reduction achieved by the trace views. The next six columns show the same information for the depths and widths of the traces. The last column ($\#F$) shows the number of filters used (in addition to the standard ones) for generating trace views.

We can observe that traces are reduced by more than 80% (up to 98% in the case of *collision*), and for at least 90% of the programs the traces have been reduced by 90% or more. For the height of the traces, the min/max/median reductions are 13%, 77%, and 41%, respectively. Wider traces cause a significant amount of horizontal scrolling. We therefore also measured the width of traces (in number of nodes). In only one of the cases the width could not be reduced. Nine programs had the maximum trace width of 4. For three of these programs the resulting trace view was linear. Four trace views had a maximum width of 3, and a linear trace was obtained for seven of the programs.

The median of 1 for column $\#F$ reflects the fact that trace views can be generally generated quite quickly.

A direct comparison with the size reduction potential of other approaches is either not possible (approaches in proof domains cannot express traces for program executions), or not very useful (the approaches [1], [9], [11] provide only indirect, limited control over trace size). In any case, we don't see our tracing approach in competition to Perera et al., but rather as a potential orthogonal extension.

VII. RELATED WORK

To address the problem of trace size in the context of debugging, Perera, et al. [9] and Ricciotti et al. [11] have employed program slicing to generate smaller program traces. Their technique takes a section of the computation result that is selected by the user and uses *backward slicing* [14] to generate a path to the input focusing on the computations that were responsible for that section. It replaces all the irrelevant computations by holes, thereby focusing on the steps that are important for understanding the section of the result

that was surprising to the user in the first place. While this technique generates a technically correct subtrace, the result can still be large, even for simple programs. The problem is that much of the information that is produced through slicing techniques, while technically relevant, might not contribute to the explanation sought by the user.

The program slicing approach works well in the context of debugging when the focus on part of a trace can be guided by questions about specific parts of a computation's output. However, when no information is available to inform the program slicing analysis, traces cannot be simplified. Moreover, parts of the trace that are not eliminated by slicing cannot be simplified either.

The goal of Acar et al. [1] is to provide information about how a particular output was generated from the execution of a program. Traces can then be fed into a *disclosure slicing* algorithm, which, given a partial output of a program, generates information about how this output was produced. This approach is very similar to backward slicing presented by Perera et al. [9] and consequently suffers similar limitations. Furthermore, while the disclosure slicing method works very well for its specific task (the tracking of provenance), it is too rigid for tracing in general.

The relation between proofs and programs means that there is much related work on representations of proofs, especially those generated by computers. Proofs that are generated with the aid of a proof assistant or an automated theorem prover face the same challenges regarding size and irrelevant information that we've noted in this paper.

For example, Farmer et al. [5] present an interface for exploring *deductive graphs* (implemented as directed bipartite graphs) of a simple proof system called IMPS. These deductive graphs share many similarities with our tree traces. Their approach allows users to collapse related nodes to reduce the size of the deductive graph. They also maintain a history of operations applied to the graph, although they don't provide means for defining and reusing filters.

Trac et al. [13] present a DAG-based view of the proofs generated by automated theorem provers. Using heuristics one can create a "synopsis" of proofs via hiding nodes. These heuristics are computed to produce an "interestingness" value, and then all nodes below a user-provided threshold are hidden. Some of the heuristics for removing nodes are similar to some of our simpler filters. For instance, removing tautologies corresponds to applying REFLEXIVE.

Dunchev et al. [4] present PROOFTOOL, a tool for viewing proofs. PROOFTOOL presents proofs in the sequent calculus using binary trees. The sequents are very similar to our evaluation judgements; because of this, the tool must deal with very similar issues to those faced by our own representation. The growing sizes of the assumption lists (analogous to our environments) led the authors to hide all such assumptions that are not in use. In addition, the tool gives users the ability to hide irrelevant portions of a proof, or focus on its specific subset. While the hiding of structural rules applies globally, hiding irrelevant proof parts has to be done manually, by interacting with the visualization of the proof tree displayed in PROOFTOOL's user interface. This shares the limitations of tailoring linear traces: for large enough programs, user-guided manipulation becomes impractical, if not impossible.

Bertot et al. [2] developed an approach for displaying explanations of proofs within theorem provers. In their system, the proof objects constructed within the logical system were converted into a textual representation of the proof. As mentioned earlier, the trees in our approach are proof objects of the proposition that the expression e evaluates to a value v in the environment ρ. Under this interpretation, some similarities between Bertot et al.'s work and our own arise. For instance, much like we tag bindings nodes with the application nodes in which the variable was first introduced, the textual notation used in Bertot et al.'s system marks uses of assumptions with the locations where they were first introduced as hypotheses. Unlike our own work, however, the presented approach more aggressively manipulates the displayed structure of the proof objects in order to improve readability: while we attempt to preserve the overall structure of the proof trees, inserting ellipses where nodes and paths are omitted, Bertot et al. rearrange and restructure their proofs to reduce the amount of nesting and to provide context as early as possible. Because this tool focuses on the method of displaying proofs to the user, it does not provide users with the tools to further adjust what they are seeing.

Jalbert et al. [7] introduce a heuristic algorithm for simplifying traces of concurrent programs. The goal is to support the identification of concurrency bugs. Traces in this approach are linear and show the results of concurrent execution as interleaved operations from the constituent programs. The trace simpification attempts to minimize the amount of interleavings between the concurrently executing programs, thereby attempting to de-obfuscate the location of concurrency bugs. The approach does not offer other trace simplification features and thus does not provide users with any control over the presentation of the trace.

VIII. CONCLUSIONS AND FUTURE WORK

We have presented a new approach for tracing program executions that is based on the systematic transformation of traces through the application of filters. A key component of approach is the visual trace representation that is based on DAGs, trades environments for binding nodes, and systematically employs ellipses. Our evaluation indicates that we can achieve sophisticated trace manipulations without exposing users to an underlying query language (on which the filters are based) and that our approach is quite effective in reducing the size of traces. In future work we will build a user interface that provides users convenient access to the application of filters.

ACKNOWLEDGMENT

This work is partially supported by the National Science Foundation under the grants CCF-1717300, DRL-1923628, and CCF-2114642.

REFERENCES

[1] Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. A Core Calculus for Provenance. In *Int. Conf. on Principles of Security and Trust*, pages 410–429, 2012.

[2] Y. Bertot and L. Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25(2):161–194, 1998.

[3] R. S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall International, London, UK, 1998.

[4] Cvetan Dunchev, Alexander Leitsch, Tomer Libal, Martin Riener, Mikheil Rukhaia, Daniel Weller, and Bruno Woltzenlogel-Paleo. Prooftool: a gui for the gapt framework. *Electronic Proceedings in Theoretical Computer Science*, 118:1–14, Jul 2013.

[5] William M. Farmer and Orlin G. Grigorov. Panoptes: An exploration tool for formal proofs. *Electronic Notes in Theoretical Computer Science*, 226:39–48, 2009. Proceedings of the 8th International Workshop on User Interfaces for Theorem Provers (UITP 2008).

[6] Gérard Ferrand, Willy Lesaint, and Alexandre Tessier. Explanations and proof trees. *Computing and Informatics*, 25:105–125, 2006.

[7] Nicholas Jalbert and Koushik Sen. A trace simplification technique for effective debugging of concurrent programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 57–66, 2010.

[8] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63(2):81–97, March 1956.

[9] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional Programs That Explain Their Work. In *ACM Int. Conf. on Functional Programming*, pages 365–376, 2012.

[10] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.

[11] Wilmer Ricciotti, Jan Stolarek, Roly Perera, and James Cheney. Imperative Functional Programs that Explain their Work. *Proceedings of the ACM on Programming Languages*, 1(ICFP), 2017.

[12] S. Thompson. *Haskell – The Craft of Functional Programming (2nd ed.)*. Addison-Wesley, Harlow, England, 1999.

[13] Steven Trac, Yury Puzis, and Geoff Sutcliffe. An interactive derivation viewer. *Electronic Notes in Theoretical Computer Science*, 174(2):109–123, 2007.

[14] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, (4):352–357, 1984.