Journal of Computer Languages 75 (2023) 101199

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.elsevier.com/locate/cola

A visual notation for succinct program traces☆

Divya Bajaj [a], Martin Erwig [b,*], Danila Fedorin [b]

[a] South Texas College, McAllen, TX, USA
[b] Oregon State University, Corvallis, OR, USA

ARTICLE INFO

Keywords:
Program trace
Proof tree
Ellipsis
Trace filter
Explanation

ABSTRACT

Program traces are a widely used representation for explaining the dynamic behavior of programs. They help to make sense of computations and also support the location and elimination of bugs. 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 concise representation. We explain the design decisions of the notation and compare it in detail with several alternatives. An important part of the trace representation is its flexibility and adaptability, which allows users to transform traces by applying filters that capture common abstractions for trace representations.

We also present an evaluation of the trace notation and filters on a set of standard examples. The results show that our representation can reduce the overall size of traces by at least 79%, which suggests that our notation is an effective improvement over the use of plain traces in the explanation of dynamic program behavior.

1. Introduction

Program understanding is intimately tied to the understanding of the dynamic behavior of programs, and program traces that keep track of the computation performed by a program on particular inputs play an important role in providing programmers with such an understanding. 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.

The use of program traces is complicated by the fact that they tend to become large quickly, even for rather small programs. Large traces make it 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 [1,2]. An important question is therefore how to provide effective access to those parts of program traces that are helpful to a user while at the same time ignoring those other parts that are unimportant. One approach is to support users of traces (programmers, educators, learners) with a tool for the creation and navigation of targeted, succinct program 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.

A preliminary version of this paper has appeared in the 2021 VL/HCC conference [3]. This revised and expanded version contains a number of changes and additions. Specifically, we have:

- developed and added a new representation of *quasi-linear traces* (Section 5),
- added an *analysis* of the extent of applicability of quasi-linear traces,
- added more details about the trace semantics (in particular, Sections 3.4 and 4.2),
- expanded the discussion of *related work*, and
- added a discussion of several areas of *future research* (Section 9).

Finally, we have supplied additional examples, explanations, and discussions throughout the paper.

A key component of our approach is 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 (which is defined in [4]). 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.

The basis for the investigation of traces is a small functional language, which is essentially an extension of the untyped lambda calculus

☆ This work is partially supported by the National Science Foundation, United States under the grants CCF-1717300, DRL-1923628, and CCF-2114642.
* Corresponding author.
E-mail addresses: dbajaj@southtexascollege.edu (D. Bajaj), erwig@oregonstate.edu (M. Erwig), fedorind@oregonstate.edu (D. Fedorin).

https://doi.org/10.1016/j.cola.2023.101199
Received 11 March 2022; Received in revised form 3 January 2023; Accepted 1 February 2023
Available online 9 February 2023

by numbers and algebraic data types. As an example, consider the following definition of the factorial function.

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

The concrete syntax of our language is actually a small subset of the functional programming language Haskell [5].

An important decision in the design of any trace notation is how to represent and keep track of branching and recursion/iteration. Specifically, one can try to retain the hierarchical structure of computation in trees or flatten trees into linear sequences. Before we present our approach, we discuss a widely known form of linear traces in Section 2, which will help us identify a number of crucial design questions that affect any trace notation. For example, to cope with the size of traces one needs some notion of partial trace and a way to hide (and unhide) (groups of) steps of a trace. Another problem that affects the size of individual steps is the handling of variable bindings. A simple substitution strategy can lead to too much redundancy that can impact the readability of traces significantly.

In Section 3 we present the basis for our trace representation. Some of the design decisions are a direct result of the issues we have identified for linear traces in Section 2. While the tree-based notation containing evaluation judgments may seem more verbose, we will show that improved opportunities for filtering and focusing ultimately result in a more flexible, scalable, and adaptable trace notation. A particularly useful feature of our notation is the ability to flexibly choose whether to show names or their values, depending on the context, which contributes significantly to the conciseness of traces. This possibility is the result of a so-called *call-by-named-value semantics*, a new variant of call-by-value semantics that we have defined.

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 various forms of trace filters and their (sometimes subtle) interaction with the trace structure in Section 4.

Out trace notation is inherently hierarchical and thus leads to a tree representation (which can be optimized in some cases to a DAG representation). However, in many cases we can create a linear or almost linear trace representation from our traces. We discuss the design of these so-called *quasi-linear traces* in Section 5.

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 6. We then present an artifact-based evaluation of our approach in Section 7. We discuss related work in Section 8 and present our plans for future work in Section 9. Finally, we present conclusions in Section 10.

2. Linear traces

Linear traces are often used in introductory functional programming textbooks as explanations for how particular function definitions work [6,7]. 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, 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 several questions regarding how to define and apply such filters. One approach is to always apply a specific set of simplifications implicitly. A problem with this approach is its inflexibility and that it may do too much in some situations and too little in others. Another approach is to offer an interactive GUI for selecting ranges and applying simple *hide* and *unhide* operations. Yet 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 to specify 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, lists or stacks of bindings) need to be maintained at all, which helps keep 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 the 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 and preferable 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`. An unfiltered linear trace would contain the trace for `fact 5` twice, which would not have any additional explanatory value and would only decrease readability. In filtering this trace, one might trust the reader and omit the steps associated with computing `fact 5` a second time; however, this does not make it entirely clear that prior lines contain the newly omitted information. We thus conclude that it would be preferable to share the subtrace for a common subexpression such as `fact 5` 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 is generally 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.

3. 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 [8]. This idea is not new and was already presented in [9], albeit in a different context and with a different goal.

3.1. Operational semantics via inference rules

In operational semantics the evaluation of expressions is defined through a set of inference rules that have the following form.

$$P_1, \ldots, P_n \implies C$$

The meaning of such an inference rule is that the statement C (which is called the *conclusion* of the rule) follows if all the statements $P_1, \ldots, P_n$ (called *premises*) are true. Inference rules are typically presented in a visual form with all premises on top of a horizontal line and the conclusion beneath it, like so:

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

In so-called *big-step operational semantics*, the premises and conclusions have the form $\rho : e \Downarrow v$, which says that expression e evaluates to the value v in the context of an environment of variable bindings ρ.

The operational semantics for our language is a variation of call-by-value semantics [8]. The main difference to standard call-by-value is that 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 in Section 3.4.

Let's give a few brief examples for operational semantics rules. 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 (regardless of the environment). Such rules without premises are also called *axioms*.

The rule for evaluating expressions e_1 *op* e_2 that involve a built-in binary operation op requires the evaluation of both argument expressions to values v_1 and v_2 and yields as a result the value v obtained from applying op.

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

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'.

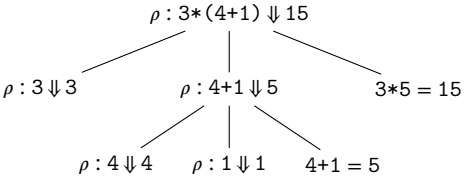
$$\frac{\rho : e_1 \Downarrow \backslash x\texttt{->}e' \qquad \rho : e_2 \Downarrow v' \qquad \rho, x = v' : e' \Downarrow v}{\rho : e_1 \; e_2 \Downarrow v}$$

3.2. Constructing proof trees with inference rules

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 $\rho : 6 \Downarrow v_2$. (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 4.

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 2. 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.

3.3. Simplifying proof trees to traces

At this point we encounter an important difference between our trace notation and 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. It looks as follows.

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

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 (like the linear notation does), we obtain an even simpler tree with a complexity similar to the linear trace.

```
3*(4+1) ⇓ 15
   4+1 ⇓ 5
```

This example is of course not very exciting; we have used it to illustrate the basic design that underlies our tree trace notation. In Section 4 we show how to construct concise tree traces through the judicious use of trace filters. But before we can do that, we describe a further simplification of traces that is achieved through a customization of the underlying operational call-by-value semantics.

3.4. Call-by-named-value semantics

The semantics we have sketched differs in a subtle but important way from the usual approach. Consider the following program in our language.

```
(let x = 1 in \n -> n + x) 5
```

The above snippet creates an unnamed function that is applied to the value 3 and increments it by 1. The body of the function refers to the variable `x`, but the variable itself is no longer in scope at the time the function is applied. Nevertheless, the program is completely valid. A standard semantics approach evaluates such a function to a so-called *closure*, which is a pair consisting of the function definition (here: `\n->n+x`) and the bindings of any non-local variables that are visible at the point of definition (here the single binding `x = 1`). This is to ensure static scoping: When the function is applied, the saved binding for the non-local variable is temporarily restored to prevent any capture and shadowing of the variable binding.

The problem with the closure representation of a function is that it goes against our attempts of reducing the amount of extraneous information presented within a trace: Just when we remove environments from our judgments, we put them right back through closures!

Our solution is to immediately substitute the referenced variables in a closure's body for their values. In the above example, the `let` expression simply evaluates to `\n->n+1`. By itself, this solution doesn't get us very far, since it re-introduces one of the issues we've identified with linear traces, namely the repeated substitution of large values, which quickly bloats the resulting expressions and worsens cognitive load.

To address this problem, we introduce *named values* into our semantics. When a value is retrieved from a variable, the name of that variable is attached to the value. This can happen multiple times, and a value can therefore accumulate multiple names. When creating a trace, named function values are replaced by their name (or the first of their names, if there are many). In this way, we are able to avoid repeated occurrences of function bodies in our traces. This simple strategy can be improved upon in various ways, and Section 9 considers some of them. The Call-by-Named-Value semantics are formalized in [4].

To see the effect of Call-by-Named-Value semantics on generated traces, consider first the following trace for the evaluation of the shown example that is produced by the standard Call-By-Value semantics . The need to repeatedly show the environments $\{x = 1\}$ and $\{x = 1, n = 5\}$ as well as the closure value ($\{x = 1\}$, `\n->n+x`) makes the presentation dense and complicates the reading (see Box I).

In contrast, the trace generated by our Call-by-Named-Value semantics doesn't have any environments and doesn't need to show any closures and thus looks much leaner. On the other hand, we do get two additional nodes showing the bindings (and their origins) of `x` and `n`. (Here we also show for illustration of named values the names attached to the values 1 and 5; in any actual trace either the name or the value is shown, see Box II.)

4. 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 2 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 show each part of a definition at least once, maybe twice for generic parts to illustrate how they operate in different circumstances and to highlight patterns, but generally not more than that. One might also want to filter out some of the more clerical arithmetic computations (for example, for decrementing a counter) and the lookup of variable bindings. We call such a tailored trace a *trace view*.

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

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.

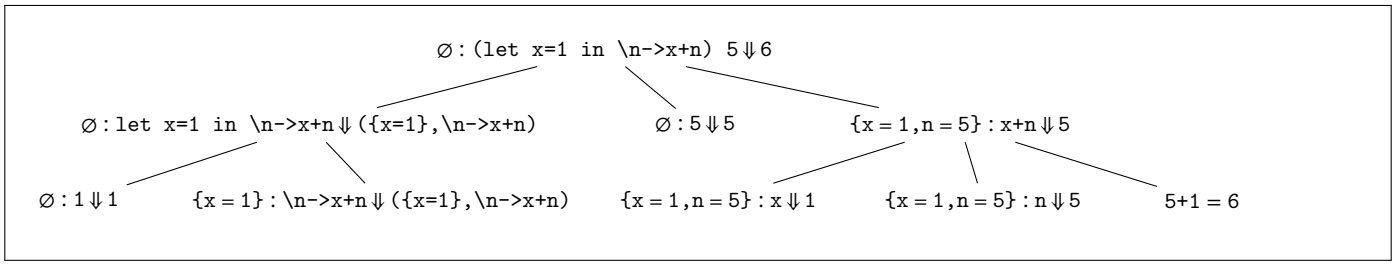
4.1. Binding nodes

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 Fig. 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.

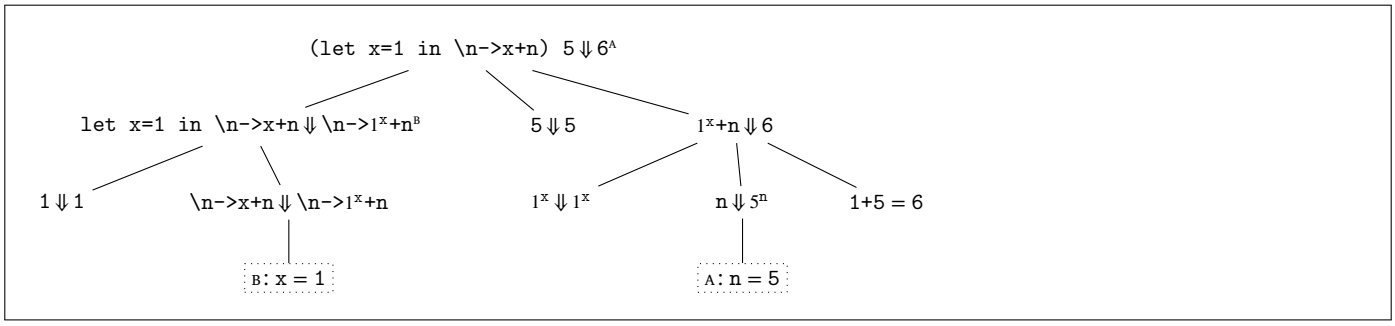
```
                fact 6 ⇓ 720^A
                      |
case x of 0 -> 1; y -> x*fact (x-1) ⇓ 720
                      |
   A: x = 6        6 ~ y      x*fact (x-1) ⇓ 720
```

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

[1] 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 page layout.



Box I.



Box II.

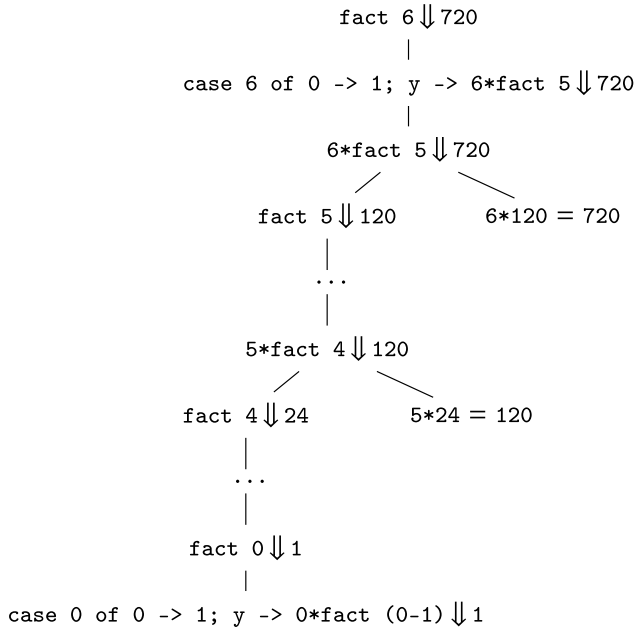


Fig. 1. Trace view for `fact 6` with many details hidden. Ellipses indicate elided paths with potentially off-branching subtrees.

child contains, as in Fig. 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.

4.2. Origin shift

Binding nodes interact in a subtle way 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.

Consider the trace in Fig. 2. The judgment `x+1 ⇓ 2` in node F that results from the evaluation of both `let` expressions is shared. One of its premises is a binding node, which has its origin in nodes B and C. We usually only show the origin of the binding for the context of the node, in this case B. However, when we transitively hide all the nodes in the subtrace starting at E, we still have to show node F as a premise for node C. But now the origin of the binding `x = 1` is node C, which should be indicated in the binding node. Our trace semantics and visualization does exactly this.

4.3. 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 ◇ = ◇, but we can also use more specific patterns such as `x` = ◇ to hide only bindings for the variable `x`.[2]

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 Fig. 1, and we have also hidden all pattern matching evaluations.

Another class of uninteresting steps that are often hidden from traces are reflexive judgments, 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 Fig. 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".

4.4. Selective filters

Note that the trace in Fig. 1 illustrates that we can also hide only a specific subset of nodes that match a pattern. One example is `case`

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

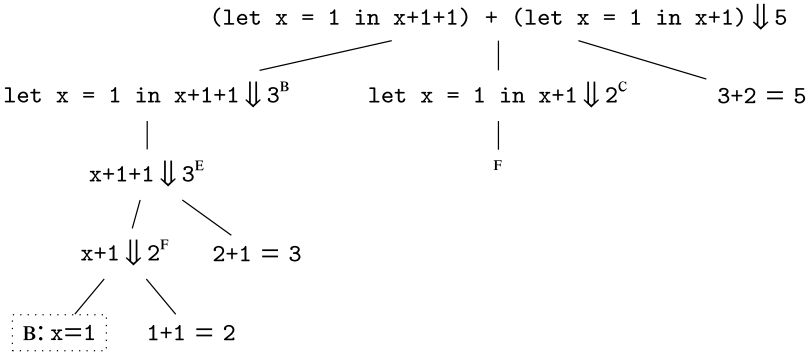


Fig. 2. Trace for `(let x = 1 in x+1+1) + (let x = 1 in x+1)`.

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 7.

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 "…" to indicate places where paths (with potential off-branching subtrees) have been omitted.

4.5. 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 DAGs. This idea is not new: In their paper on ProofTool, Dunchev et al. [10] 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, repeated copies of the same subtrace should be avoided, since they only add costs without any benefit. Specifically, a trace with a repeated copy of the same sub-trace makes the explanation bigger without adding any new information. A trace with repeated copies simply takes longer to read and process than one without the repetition;. Moreover, repeated subtraces cause an additional serious problem for the tree/DAG representation because the increase the width of traces, which makes browsing cumbersome and causes additional processing costs. 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 Fig. 3.

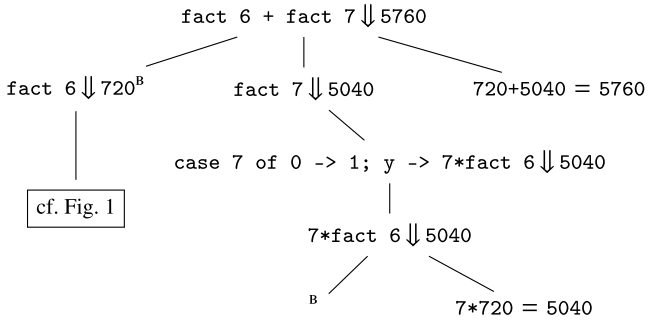


Fig. 3. Trace view for `fact 6 + fact 7`. DAGs, represented by using node references, are used to avoid the repetition of subtraces.

4.6. Trace view prototypes

We have already mentioned that we have implemented a prototype for producing and filtering trace views. This software tool is not intended for end users. Rather, the main purpose of this tool is to allow us to investigate properties of traces and trace filters. It was also instrumental in developing the trace notation itself and the query language that is used to define all filters.

The trace exploration prototype is implemented in Haskell and provides a command-line interface as part of the Haskell GHCi interpreter. The tool offers commands to compile and run programs to produce original complete traces and apply (and unapply) filters to traces. Traces and filtered traces produce output in the DOT format [11], which is then rendered by a DOT viewer. More specifically, our tool allows one to (un)apply individual filters, but it 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).

Our prototype implements the tracing for a small functional language (for details, see [4]), and the generated traces are specific for this language because traces contain judgments of the operational semantics for that language. An adaptation of the approach to other functional languages is not difficult, but applying the approach to imperative or object-oriented languages is more involved, since it would require the definition of an operational semantics for such languages. Moreover, the query language that is used for defining the filters needs to be adapted also to recognize and act on the corresponding node types (for details of the query language, again see [4]).

In addition the command-line tool, a web-based GUI for working with program traces, called Tracr, has been developed recently [12]. Tracr is a single-page web application, which consists of a frontend (written in Elm [13], which compiles to JavaScript) and a backend server for running programs (to create traces) and applying filters (for manipulating traces). A screenshot of the main interface is given in Fig. 4, showing a slightly more filtered version of the example from Fig. 1.

The trace for the current program is shown in the main window, and the list of applied filters together with interaction elements for adding, removing, and configuring filters is presented in the form of a stack at the bottom right. The top of the window contains a menu with options for loading, saving, and deleting programs and traces. The tool also allows users to share a particular state of exploration with others through the generation of a URL. This allows others to see the same program and filters by navigating to that link.

This tracing tool can be accessed using any web browser [14].

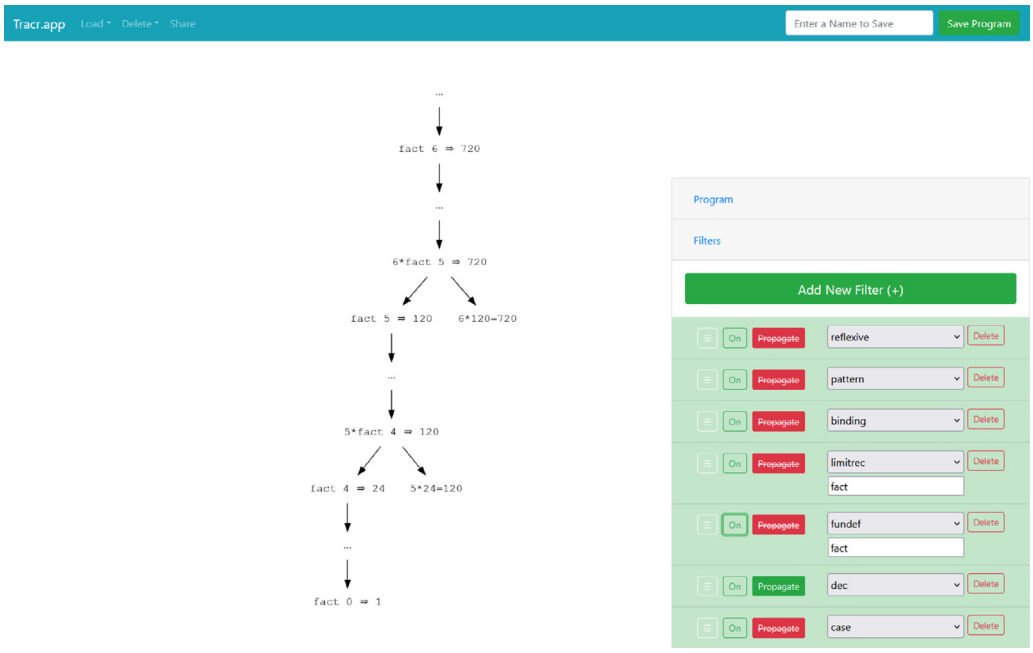


Fig. 4. Tracr user interface.

5. Quasi-linear traces

As discussed in Section 2, linear traces suffer from several disadvantages. Still, they can also be quite useful, in particular when they are not too long and repetitive. Thus the question arises whether we can create some form of linear traces from our tree representation, at least in some cases.

The trace notation presented in this paper generally diverges from the linear traces shown in the introduction rather quickly. This is due to the structure of the operational semantics whose rules often have several premises, which leads to a fan-out of proof trees. However, we can observe that a large number of tree traces take on an almost linear shape after filter operations have been applied. Again, the trace in Fig. 1 serves as an example.

Though its representation is clearly a tree, we can observe that at any branch, the second child (written on the right) doesn't itself have any children and can be viewed as a kind of "remark" or "footnote" justifying its parent. From such traces we can generate a *linear trace with annotations*, or *quasi-linear trace*, by gathering all statements of the major spine of the trace while annotating, when needed, some of the statements with a footnote symbol which points to a corresponding side note in an accompanying second column. This idea is illustrated in Fig. 5. Note how the second child for the statement in the third line is pointed to by the footnote symbol, and how the corresponding statement appears in the next line together with the "primary" child that was selected for the main linear sequence.

With the addition of this side notes column, we have recovered a linear form of traces which is similar to what we showed in Section 2. There are, however, several differences between the two traces. First, each line in our trace represents a sub-computation, rather than the state of the whole expression as it is being evaluated. Thus, we won't encounter ballooning sequences of applications or binary operation like those we observed in the "textbook" trace for `fact 6`. Second, our quasi-linear traces stay in one-to-one correspondence with the semantics because during the creation of the linear output we still maintained the structure of the original traces.

As another example consider the quasi-linear trace for the evaluation of the expression `filter even [1,2,3,4]`, shown in Fig. 6, that illustrates how the notation can handle even traces with many side notes quite well.

	Side Notes
`fact 6` ⇓ `720`	
⇐ `case 6 of {0->1;y->6*fact 5}` ⇓ `720`	
⇐ `6*fact 5` ⇓ `720`†	
⇐ `fact 5` ⇓ `120`	†`6*120 = 720`
...	
⇐ `fact 0` ⇓ `1`	
⇐ `case 0 of {0->1; y->0*fact(0-1)}` ⇓ `1`	

Fig. 5. Quasi-linear trace of `fact 6`. Each judgment is followed on the next line by a single judgment that explains or justifies it (indicated by the symbol ⇐). The *Side Notes* column show the second, auxiliary judgments for those nodes that have two children in a trace.

	Side Notes
`filter even [1,2,3,4]` ⇓ `[2,4]`	
...†	
⇐ `filter even [2,3,4]` ⇓ `[2,4]`	†`even 1` ⇓ `False`
...‡	
⇐ `2:filter even [3,4]` ⇓ `[2,4]`	‡`even 2` ⇓ `True`
⇐ `filter even [3,4]` ⇓ `[4]`	
...*	
⇐ `filter even [4]` ⇓ `[4]`	*`even 3` ⇓ `False`
...§	
⇐ `4:filter even []` ⇓ `[4]`	§`even 4` ⇓ `True`
⇐ `filter even []` ⇓ `[]`	

Fig. 6. Quasi-linear trace for `filter even [1,2,3,4]`. Side notes can also appear as a consequence of ellipses having multiple children.

Finally, we can expand the scope of quasi-linear traces by not only including leaves in side notes, but sequences of judgments, that is secondary children that are not necessarily leaves but have only a linear sequence of descendants in the tree. An example is given in Fig. 7 that shows a trace for a function computing subsets.

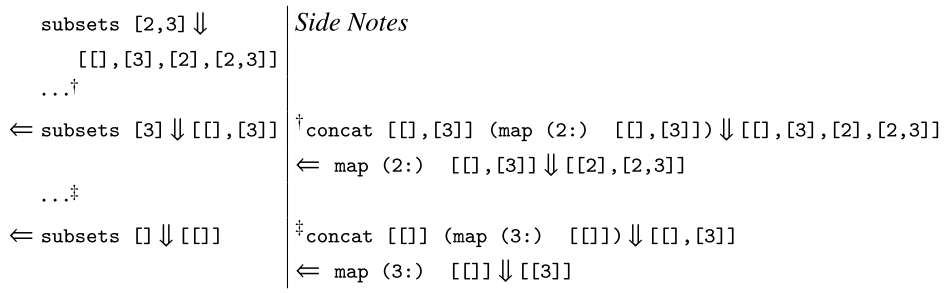


Fig. 7. Quasi-linear trace for `subsets [2,3]`. In this case, each of the side notes is itself a linear trace (each with one child).

The quasi-linear trace notation seems to be quite widely applicable. Out of the 21 example programs considered in the evaluation in Section 7, 10 have a quasi-linear trace in the strict sense (where side notes consist of only leaf auxiliary children) and 14 are quasi-linear trace in the extended sense (where side notes may consist of linear sequences of judgments).

6. Classification of trace models

Approaches for program tracing can differ in a variety of ways. In addition to aspects of the notation, the flexibility of manipulating 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, proofs, or other

We have already discussed the structure and binding information aspects in Sections 2 and 3, and the classification of the different approaches in these regards are shown in Table 1.

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 in [15] 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 discussed in [17,18], and [19] give users some indirect control through the selection of partial input/output. These user actions affect the trace globally.

The approach presented in [20] represents traces for proof systems as directed bipartite graphs (denoted as digraphs in Table 1). This system gives users full control over traces through the collapsing of related nodes. A GUI lets users select nodes and perform transformations on these nodes. Thus, users have 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 1. 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 constructed interactively by the user during a proof session.

7. 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 in several computer science classes 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 programming language analysis and interpretation (*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.

7.1. 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 [21]. Although subsequent research has shown that the situation is more complicated, the general trend remains: Our short-term memory is quite limited. Motivated by this insight, our approach is 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. Of course, the judgment of what parts actually do "significantly contribute" to program explanations is dependent on the user and the questions they have about a program execution. The ability for users to selectively apply filters in our approach accounts for this fact.

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 noticed 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.

Table 1
Classification of trace notations.

Approach	Form	Handling of bindings	User control	Scope	Domain
[6][7]	linear	substitution	full	by node	execution
[16]	linear	substitution	none	global	proofs
[15]	linear	environment	none	global	execution
[10]	tree	environment	limited	global	proofs
[17][18][19]	tree	environment	limited	global	execution
[20]	digraph	substitution	full	by node	proofs
This paper	DAG	binding nodes	full	global	execution

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.

7.2. 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 2 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 ° 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 judgments.
- PartialApp hides all partial function applications and all of their descendants. Partial function application is identified by patterns of the form `f x ⇓ \y -> e`; however, some additional care is required to avoid hiding the subtrees corresponding to the evaluation of the function's arguments.
- FunDef$_f$ 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.
- OuterCase 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.

In addition to these universal filters, some filters are used only in the creation of 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 Fig. 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 with 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 given in Fig. 8.

7.3. Results

Table 2 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 that are always applied) for generating trace views.

We can observe that traces are reduced by at least 79% (up to 98% in the case of *collision*), and for at least 85% of the programs the traces have been reduced by 85% or more. For the height of the traces, the min/max/median reductions are 13%, 77%, and 39%, respectively. Wider traces cause a significant amount of horizontal scrolling. We therefore also measured the width of traces (in number of nodes). The width of the traces was significantly reduced. For 10 out of 21 programs, the width was reduced by at least 92%, where the minimum reduction in width was 67%.

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

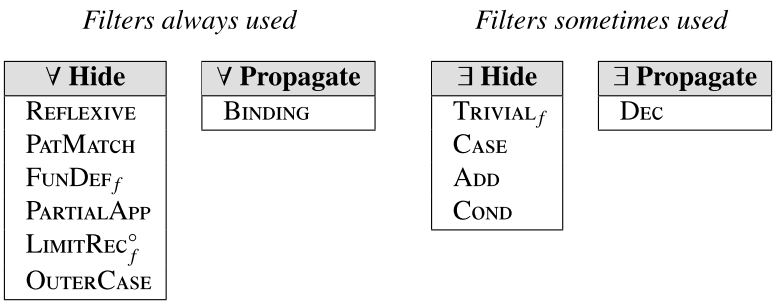


Fig. 8. Complete list of filters together with their applicability (always vs. sometimes) and their effect (hiding only vs. also propagating values).

Table 2
Size of traces (T) and trace views (T^*) (#: number of nodes, ↕: depth, ↔: width) and space savings (Δ%) achieved. Column #F shows the number of additional filters needed, and the last column lists those additional filters.

Program	#T	#T^*	Δ%	↕T	↕T^*	Δ%	$\overline{T}$	$\overline{T^*}$	Δ%	#F	Additional filters
Factorial	80	12	85	22	10	55	6	2	67	1	Dec
Twice-fact	70	15	79	13	10	23	9	3	67	1	Dec
Collision	532	13	98	25	8	68	66	3	96	3	Case, Cond, Trivial$_f$
Reverse	203	18	91	21	18	14	19	1	95	1	Trivial$_f$
Reverse-accum	89	8	91	13	8	39	13	1	92	1	Case
Replicate	97	8	92	20	8	60	8	1	88	1	Dec
Sum	148	9	94	34	8	77	8	2	75	0	–
Filter	127	13	90	20	11	45	12	2	83	1	Trivial$_f$
Merge-lists	140	12	91	25	12	52	13	1	92	1	Cond
Cart-product	242	13	95	20	9	55	27	2	93	1	Trivial$_f$
Subsets	359	17	95	23	12	39	41	2	95	2	Trivial$_f$, Case
Quicksort	900	34	96	38	13	66	77	4	95	1	Trivial$_f$
Search-BST	71	8	89	10	8	20	16	1	94	2	Case, Cond
Insert-BST	115	14	88	18	14	22	15	1	93	2	Case, Cond
Delete-BST	167	22	87	24	18	25	16	2	88	1	Cond
inorder-BST	70	9	87	11	9	18	15	1	93	0	–
Constants	239	27	89	18	12	33	29	5	83	2	Case, Trivial$_f$
Eval-expr	158	18	89	17	10	41	23	5	78	0	–
Eval-fun	226	20	91	20	11	45	24	4	83	1	Trivial$_f$
typecheck	127	25	80	16	14	13	15	5	67	0	–
Fold-const	133	25	81	13	11	15	20	5	75	2	Add, Trivial$_f$

A direct comparison with the size reduction potential of other approaches is either not possible (approaches in the proof domains cannot express traces for program executions), or not very useful (the approaches [17–19] provide only indirect, limited control over trace size).

8. Related work

In this section we discuss related work on tracing that has appeared in several different areas. We group the discussion by specific methods employed or aspects focused on: *program slicing*, *proof trees*, *trace structures & operations*, *user interfaces*, *specialized semantics*, and *traces for explanations*.

Program slicing. To address the problem of trace size in the context of debugging, Perera et al. [17] and Ricciotti et al. [18] 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* [22] 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. In any case, we don't see our tracing approach in competition to Perera et al. but rather as a potential orthogonal extension.

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. [19] 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. [17] 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.

Our approach also differs from program and trace slicing in that transformations applied to traces can be re-used for more than one input or program. Selectors created for one trace can be immediately applied to another, without any insight into its input or output. The default filters in Fig. 8 were determined experimentally, and produce satisfactory initial results when applied to traces of new programs. This is different from trace slicing, where the simplified trace is produced from a concrete, if partial, output example.

Proof trees. Considering proof trees as a basis for explanations has been suggested by Ferrand et al. [9]. They observe that a proof tree is a *declarative* view of the trace of a computation that can be used to explain programs in the domain of Constraint Logic Programming. Specifically, a node in the proof tree (or explanation) for a constraint

logic program is an answer (which can be positive or negative). Thus, an answer is explained as a consequence of other answers. This declarative view of explanations is then used to explain the solvers for constraint satisfaction problems.

Our trace notation is also based on proof trees, but is not tied to a particular programming paradigm. Instead, we adopt the standard vie from operational semantics that uses a proof tree as a compositional, formal representation of statements about program evaluation steps.

Trace structures & operations. 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. [20] 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. [23] 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.

Jalbert et al. [15] 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 simplification 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.

In [4] we have defined a query language for the systematic transformation of (tree-based) program traces. The filters used in this paper are all defined with that query language. The query language consists of operations for hiding and propagating information plus a rich set of so-called *selectors*, which specify the set of nodes operations should be applied to.

User interfaces. Dunchev et al. [10] 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 judgments; 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. While Dunchev et al. decided against DAG-based views of explanations (due to a lack of good layout algorithms), we opted to use references to nodes, thus showing identical parts of a trace only once.

Bertot et al. [16] 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.

Specialized semantics. The Call-By-Named-Value semantics we have developed is somewhat similar in purpose to the *Clairvoyant Call-By-Value* semantics, which was presented by Hackett et al. [24] as an alternative to the *Call-By-Need* semantics [25] to make reasoning about programs easier. The paper explains how the state is not required to be threaded around through every computation in the new semantics, enabling a cleaner denotational cost semantics for lazy evaluation.

Call-By-Named-Value semantics was first introduced in [4] and is used to support the readability of traces by replacing a value by the name bound to it. This presentation supports the understanding of applications of functions without being overwhelmed by having to read and absorb the complete function definition every time the function is used in the trace. Thus, traces generated by Call-By-Named-Value semantics are often shorter and simpler than those by Call-By-Value, which reduces cognitive load and therefore supports comprehension [26].

The work of Acar et al. [19] defines programming language semantics augmented with tracing to support provenance tracking. Much like our own, traces constructed in Acar et al.'s work are not linear, and closely match the structure of the proof tree generated by the semantics. The approach differs, however, in that traces are explicitly specified in the inference rules: in addition to the environment, expression, and resulting value, the evaluation relation in [19] explicitly states the corresponding trace. In contrast, our traces are constructed implicitly from the proof trees generated via the semantics. This is motivated by the goal of explaining how a particular result was computed, given an evaluation model specified by the semantics.

Traces for explanations. While program traces are traditionally viewed as a vehicle for finding and eliminating program bugs, some trace notations were specifically designed as explanation artifacts. For example, the notation employed in Chapter 6 of [27] is specifically used for explaining sorting algorithms. In particular, this notation displays side-by-side the intermediate lists that are created as a sorting algorithm executes. Unlike the "textbook" notation in the introduction, this notation does not display the code used to describe the sorting algorithm. This makes it more approachable to readers who are not familiar with computer programming. Taken by itself, the notation in [27] is manually constructed and specific to each sorting algorithm. However, we believe it's possible to apply the approach presented in this paper to create trace views that closely correspond to the list visualizations in the book.

The work on *Probula* [28–30] presents a visual language for explanations in a domain different from proofs and execution. Specifically, the presented visual language is used to explain probabilistic reasoning. Rather than considering a particular execution of a program, Probula presents explanations in the form of connected sequences of probability distributions, which are generated by a small number of operations. Whereas the visual language presented in this paper uses branching to

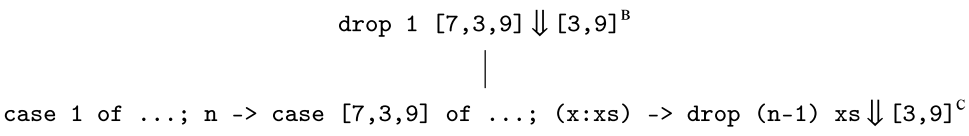


Fig. 9. Explanation of the drop function, with irrelevant branches removed.

describe subcomputations, branches in Probula correspond to decisions made by an agent (for example, whether or not to switch doors in the Monty Hall Problem.

Furthermore, Probula features a "group" operation, which can be used to arbitrarily combine related states. This is in contrast to the grouping in our visual language, where nodes are shared automatically if their computation is equivalent. A similarity between the two approaches is that both allow for multiple explanations for the same underlying computation. In Probula's case this is done through various transformation laws which may be used to reduce the number of intermediate probability distributions. In our case, this can be done by adjusting the set of filters applied to a trace.

The approach pursued in [31–33] is based on explicit representations of so-called *value decompositions* to facilitate the generation of explanations for specific program executions. Value decompositions are granular representations of values that are aggregated during a computation. Such representations work only in specific domains that exhibit a mathematical structure to support value decompositions and the necessary operations, and they take a different form, depending on different application areas (for example, dynamic programming algorithms [32,33] or hierarchical decision making [31]). The traces created by the approach do not record detailed effects of low-level program activities but rather capture key aspects in the form of value components that can be aggregated into a final value. The aggregation relationship among the value components allows the creation of explanations why computed results are better than ostensible alternatives. The fact that traces are not directly derived from low-level computations but have to be explicitly created is similar to the approach in [19].

9. Future work

We have identified a number of promising avenues for future work. First, there are more opportunities to make explanation traces more concise. One such opportunity is dead code elimination, both within expressions and traces. Suppose, for example, that the following judgment is part of an explanation trace.

`case False of ...; False -> 0 ⇓ 0`

Notably, the `True` branch of the case expression has been replaced by an ellipsis, which is justified because it doesn't contribute to the explanation of the current computation. This simplification saves space and keeps cognitive load low. The strategy of partial code hiding is actually applicable in many more cases. Even if part of the code is not dead, it might be "dormant" and thus explanatorily irrelevant in the current part of the trace. For example, the part of the trace shown in Fig. 9 displays only those parts of the `case` expression that are relevant in the current situation.

Finally, consider the following judgment.

`let x = twice fact 2 in sqr 5 ⇓ 25.`

While the expression `twice fact 2` is technically not dead (it will be evaluated in Call-By-Named-Value semantics), the result of the subtrace is not at all used in the computation of `sqr 5` and thus doesn't contribute to its explanation. While a user could hide the unused traces manually in the current model, this needs to occur on a case-by-case basis, and it would be useful to offer automatic filters for these kind of situations.

The Call-By-Named-Value semantics offers another opportunity for improving explanation traces: We can implement more sophisticated strategies for deciding whether to show a value or its name. In this paper we have chosen to omit the contents of named function values in favor of their names to reduce the size of the judgments and improve the explanation's clarity. However, doing so is not always the optimal approach. For example, a user may define the identity function as follows.

```
let theIdentity = \x -> x in ...
```

In this case, using the name may be overly verbose and indeed less clear than simply showing the function value itself. Conversely, some non-function values can benefit from naming. A user may (excessively) define the mathematical constant π as:

```
let pi = 3.1415926535898 in ...
```

Rather than showing the exact value over and over, the name `pi` is sufficient. There are many possible heuristics that can be explored to improve the display of named values. One such heuristic is hinted at by the prior examples, and involves comparing the textual length of a named value with that of its names. This heuristic, though, may overly favor terse and inexpressive names.

Other, more intelligent methods of displaying named values may include asking the user of the system to tag important names, and giving these names precedence even for non-function named values. Similar heuristics may be applied to *which* names are chosen to be displayed. Named values in CBNV semantics can have an arbitrary number of names; at present, we always display the first of these to the user. However, a value may acquire a more meaningful name later on in the evaluation of a program. For example, an element of a list may acquire the name `pivot` during the execution of `quicksort`, which may be more significant than its previous names. It could then be beneficial for the quality of the explanation to show the more expressive name, or even both.

Another avenue for research is the qualities of difficult-to-read explanations. While studying and generating explanations used as examples, we noted several qualities shared by what we considered good, tailored explanations. We believe that some of these qualities can be measured and therefore allow us to have a convenient metric for explanation quality beyond the count of the nodes or the dimensions of the DAG. There are many qualities we have identified in these explanations: traces that are too long or too wide, nodes that contain too much information, and graph structures that are difficult to comprehend at a glance. Existing research into cognitive load could be leveraged to measure the difficulty of (a section of) an explanation such as the *Cognitive Complexity of Computer Programs* framework given in Duran et al. in [34]. A metric for explanation complexity can be the basis for a cost/benefit of explanation traces.

Finally, it is not clear how well our approach actually works in practice. For example, it could turn out that users find it difficult to identify the combination of filters that helps them focus traces on the information that helps them answer their specific question about a computation. We need to conduct user studies to find out what does and what does not work for different kinds of users in different circumstances.

10. Conclusions

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 our 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.

CRediT authorship contribution statement

Divya Bajaj: Software, Writing, Formal analysis. **Martin Erwig:** Conceptualization, Funding acquisition, Project administration, Supervision, Writing, Formal analysis. **Danila Fedorin:** Software, Writing, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] A. Hamou-Lhadj, T. Lethbridge, Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system, in: IEEE Int. Conf. on Program Comprehension, 2006, pp. 181–190.
[2] Yang Feng, Kaj Dreef, James Jones, Arie van Deursen, Hierarchical abstraction of execution traces for program comprehension, in: IEEE/ACM Int. Conf. on Program Comprehension, 2018, pp. 86–96.
[3] D. Bajaj, M. Erwig, D. Fedorin, K. Gay, A visual notation for succinct program traces, in: IEEE Int. Symp. on Visual Languages and Human-Centric Computing, 2021, pp. 1–9.
[4] D. Bajaj, M. Erwig, D. Fedorin, K. Gay, Adaptable traces for program explanations, in: Asian Symp. on Programming Languages and Systems, in: LNCS, vol. 13008, 2021, pp. 202–221.
[5] S.L. Peyton Jones, Haskell 98 Language and Libraries: The Revised Report, Cambridge University Press, Cambridge, UK, 2003, p. 270.
[6] R.S. Bird, Introduction to Functional Programming using Haskell, Prentice-Hall International, London, UK, 1998.
[7] S. Thompson, Haskell – the Craft of Functional Programming, second ed., Addison-Wesley, Harlow, England, 1999.
[8] B.C. Pierce, Types and Programming Languages, MIT Press, Cambridge, MA, 2002.
[9] Gérard Ferrand, Willy Lesaint, Alexandre Tessier, Explanations and proof trees, Comput. Inform. 25 (2006) 105–125.
[10] Cvetan Dunchev, Alexander Leitsch, Tomer Libal, Martin Riener, Mikheil Rukhaia, Daniel Weller, Bruno Woltzenlogel-Paleo, PROOFTOOL: a GUI for the GAPT framework, Electron. Proc. Theor. Comput. Sci. 118 (2013) 1–14.
[11] Graohviz, DOT language, 2022, https://graphviz.org/doc/info/lang.html.
[12] J. Barnes, Tracr - Visual Interface for Generating Explanations, M.S. Final Project Report, School of EECS, Oregon State University, 2022, https://ir.library.oregonstate.edu/concern/graduate_projects/9306t607n.
[13] E. Czaplicki, S. Chong, Asynchronous functional reactive programming for GUIs, in: ACM Conf. on Programming Languages Design and Implementation, 2013, pp. 411–422.
[14] Tracr, Program trace exploration tool, 2022, http://tracr.engr.oregonstate.edu/.
[15] Nicholas Jalbert, 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, 2010, pp. 57–66.
[16] Y. Bertot, L. Théry, A generic approach to building user interfaces for theorem provers, J. Symbolic Comput. 25 (2) (1998) 161–194.
[17] R. Perera, U.A. Acar, J. Cheney, P.B. Levy, Functional Programs That Explain Their Work, in: ACM Int. Conf. on Functional Programming, 2012, pp. 365–376.
[18] Wilmer Ricciotti, Jan Stolarek, Roly Perera, James Cheney, Imperative functional programs that explain their work, Proc. ACM Program. Lang. 1 (ICFP) (2017).
[19] Umut A. Acar, Amal Ahmed, James Cheney, Roly Perera, A core calculus for provenance, in: Int. Conf. on Principles of Security and Trust, 2012, pp. 410–429.
[20] William M. Farmer, Orlin G. Grigorov, Panoptes: An exploration tool for formal proofs, Electron. Notes Theor. Comput. Sci. 226 (2009) 39–48, Proceedings of the 8th International Workshop on User Interfaces for Theorem Provers (UITP 2008).
[21] George A. Miller, The magical number seven, plus or minus two: Some limits on our capacity for processing information, Psychol. Rev. 63 (2) (1956) 81–97.
[22] Mark Weiser, Program slicing, IEEE Trans. Softw. Eng. 10 (4) (1984) 352–357.
[23] Steven Trac, Yury Puzis, Geoff Sutcliffe, An interactive derivation viewer, Electron. Notes Theor. Comput. Sci. 174 (2) (2007) 109–123.
[24] Jennifer Hackett, Graham Hutton, Call-by-need is clairvoyant call-by-value, Proc. ACM Program. Lang. 3 (ICFP) (2019) 1–23.
[25] John Launchbury, A natural semantics for lazy evaluation, in: ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, 1993, pp. 144–154.
[26] Weidong Huang, Peter Eades, Seok-Hee Hong, Measuring effectiveness of graph visualizations: A cognitive load perspective, Inf. Vis. 8 (3) (2009) 139–152.
[27] M. Erwig, Once Upon an Algorithm: How Stories Explain Computing, MIT Press, Cambridge, MA, 2017.
[28] M. Erwig, E. Walkingshaw, A visual language for explaining probabilistic reasoning, J. Vis. Lang. Comput. 24 (2) (2013) 88–109.
[29] M. Erwig, E. Walkingshaw, Visual explanations of probabilistic reasoning, in: IEEE Int. Symp. on Visual Languages and Human-Centric Computing, 2009, pp. 23–27.
[30] M. Erwig, E. Walkingshaw, A DSL for explaining probabilistic reasoning, in: IFIP Working Conference on Domain-Specific Languages, in: LNCS, vol. 5658, 2009, pp. 335–359.
[31] M. Erwig, P. Kumar, MADMAX: A DSL for explanatory decision making, in: ACM SIGPLAN Conf. on Generative Programming: Concepts & Experiences, 2021, pp. 144–155.
[32] M. Erwig, P. Kumar, Explainable dynamic programming, J. Funct. Program. 31 (e10) (2021).
[33] M. Erwig, P. Kumar, A. Fern, Explanations for dynamic programming, in: Int. Symp. on Practical Aspects of Declarative Languages, in: LNCS, vol. 12007, 2020, pp. 179–195.
[34] Rodrigo Duran, Juha Sorva, Sofia Leite, Towards an analysis of program complexity from a cognitive perspective, in: ACM Conf. on International Computing Education Research, 2018, pp. 21–30.