Adaptable Traces for Program Explanations

Divya Bajaj[1], Martin Erwig[1], Danila Fedorin[1], and Kai Gay[1]

Oregon State University, Corvallis OR 97331, USA
[bajajd,erwig,fedorind,gayk]@oregonstate.edu

Abstract. Program traces are a sound basis for explaining the dynamic behavior of programs. Alas, program traces can grow big very quickly, even for small programs, which diminishes their value as explanations. In this paper we demonstrate how the systematic simplification of traces can yield succinct program explanations. Specifically, we introduce operations for transforming traces that facilitate the abstraction of details. The operations are the basis of a query language for the definition of trace filters that can adapt and simplify traces in a variety of ways. The generation of traces is governed by a variant of Call-By-Value semantics which specifically supports parsimony in trace representations. We show that our semantics is a conservative extension of Call-By-Value that can produce smaller traces and that the evaluation traces preserve the explanatory content of proof trees at a much smaller footprint.

Keywords: semantics · language design · domain-specific languages.

1 Introduction

Explaining program behavior has many uses, including program maintenance, debugging, and teaching. In particular, when the correctness of a program is in doubt, an explanation can help to regain the user's trust and confidence. Users often employ debuggers to understand program behavior [14], even though debugging is costly [15] and focuses more on identifying and removing bugs. Moreover, debuggers typically already assume an understanding of the program by the programmer [11]. Research on customizable debugging provides additional evidence for the limitations of generic debugging approaches [9, 7].

Perera, et al. [12] use partial program traces to explain program executions. Through *backward program slicing* only those parts of a trace are retained that contribute to a selected part of the output; all irrelevant parts of the trace are replaced by holes. However, the resulting traces can still be large, even for simple programs, because much of the information that is produced through slicing techniques, while technically relevant, might not contribute to the explanation.

Partial traces can be very effective, but they may omit the wrong information. In general, no one trace works equally well as an explanation for every user, since different users typically have different questions about the behavior of a program. The approach we present in this paper gives users the ability to manipulate program traces through a query language and thus gives them control over which

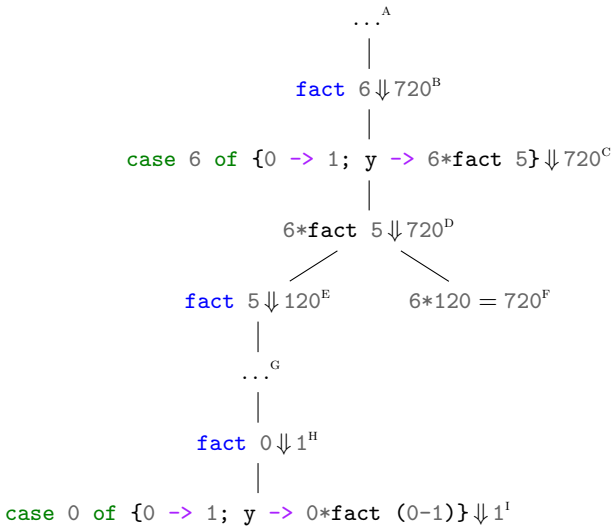

Fig. 1. Trace view for `fact 6`. (The LaTeX code for trace views was generated by our prototype implementation, with some manual adjustment of horizontal positioning.)

parts of a trace to hide and which parts to keep. To illustrate this aspect, we demonstrate how to create a trace for the factorial function that could be used, for example, as a teaching aid. Consider the following definition.

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

Suppose that we want to explain the computation of `fact 6`. A proof tree generated by a typical big-step Call-By-Value operational semantics consists of 80 nodes and 22 levels, which is a lot of information. However, to understand how this computation works, one doesn't need to see all instances of the recursive function call. 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 of the more clerical arithmetic computations (for example, for decrementing a counter) and the lookup of variable bindings. We call such a filtered trace a *trace view*. In Figure 1 we show a trace view with only 7 non-hidden judgments on 8 levels that meets these expectations. The trace view is obtained from a complete trace in six steps: (1) hiding top-level declarations, (2) hiding and propagating variable lookups, (3) hiding and propagating evaluations of subtractions, (4) hiding reflexive judgements, (5) hiding intermediate recursive calls, and (6) hiding pattern matching evaluations.

These steps are achieved by filter operations which hide nodes and subtrees, occasionally propagating information from hidden nodes to the rest of the trace. The nodes to which a particular filter is to be applied are determined by patterns that are matched against the judgments in the nodes of the trace.

First, `let` expressions that define the program to be explained are contained in nodes that carry judgements of the form `let` x`=`e' `in` $e \Downarrow v$. A pattern for such a judgment can use values or a wildcard symbol ◇. Thus, to hide the definition of `f` we use the pattern `let f =` ◇ `in` ◇ ⇓ ◇ Similarly, information about bindings is presented by so-called *binding nodes*, which carry judgments of the form $n\colon x{=}v$, saying that variable x has the value v and that the binding was introduced by node n. To hide all binding nodes, as we do in this example, we use the pattern ◇: ◇ = ◇. However, we do not simply hide binding nodes, but also propagate the bound values to where they are used. For reasons that will become clear in Section 4, we call this operation *factoring*. The effect of factoring can be seen, for example, in node C where the value `6` is used instead of the variable `x`.

While we have hidden the binding node for `fact` (which is a premise for node B), we *haven't* propagated its value (the function definition), as can be seen again in node C in the expression `6*fact 5`. Responsible for this behavior is our version of operational semantics, *Call-By-Named-Value*, introduced in Section 2, which stores names with values. As explained in Section 3, the presentation of traces exploits the names of function values to produce smaller and more readable traces. This effect is extremely useful for tracing the execution of higher-order functions where substituting function values for variables that are referenced (potentially multiple times) can render traces effectively unreadable.

A filter for hiding and propagating some of the arithmetic is also expressed through factoring with a pattern. In our example we use the pattern ◇`-1` ⇓ ◇ to hide only decrements by `1`, since we want to retain some of the multiplication expressions, which are important for explaining the functioning of `fact`. Also, some of the judgements, for example, $5 \Downarrow 5$, do not add explanatory value to the trace. A filter to hide all such judgements uses a pattern $\diamond_a \Downarrow \diamond_a$, that contains indexed wildcards. Indexed wildcards force equivalence on values in different places.

Hiding recursive `fact` calls is more complicated, since we don't want to hide all applications of `fact`. We can keep the first two calls and the last call as well as the first and last expansion of the function body by modifying the set of matched nodes through a function `limitRec`. The function `limitRec` is defined with combinators described in Section 5. Note that we shouldn't define `limitRec` to simply remove the first and last of the matched nodes (assuming we can rely on the matched nodes to appear in a particular order), because this wouldn't work well, for example, in the expression `fact 5+fact 6`. The definition provided in Section 5 is more robust and works well with cases like these.

We also hide all pattern matching judgements of the form $v|p \rightsquigarrow \rho$ (that match a value v against a pattern p and produce a binding ρ), again using a pattern with only wildcards: ◇| ◇ ⇝◇. As with the recursive function calls, we only hide nodes and don't propagate any information. Finally, we also hide the definition of `fact` to focus on the evaluation steps.

To summarize, the trace view in Figure 1 can be produced from the complete trace by applying the filters shown in Figure 2, which can be done step-by-step in the user interface of our prototype or by running a script.

```
hide (funDef fact)
factor binding
factor dec
hide reflexive
hide middleFact
hide patMatch
```

```
funDef f   ≡ let f = ◇ in ◇ ⇓◇
binding    ≡ ◇:◇ = ◇
dec        ≡ ◇-1⇓◇
reflexive  ≡ ◇a⇓◇a
middleFact ≡ limitRec fact
patMatch   ≡ ◇| ◇ ⇝◇
```

Fig. 2. Filters used to produce the trace view for `fact 6`.

Some of these filters are quite generic and can be reused in other examples. In fact, we reuse them all in the next example to illustrate more features of our approach. Consider the trace in Figure 3 that explains the following program.

```
let twice = \f -> \x -> f (f x) in
let fact = \x -> case x of {0 -> 1; y -> x * fact (x-1)}
 in twice fact 2
```

The trace view is generated by the following filter script, which uses two more patterns (`fun` and `fact2`) whose meaning should be obvious. The two patterns illustrate how general patterns can be combined into very specific ones. Here, the first factoring step excludes the binding for `f`, and the sequencing combinator `then` is used to apply the hiding operation to recursive nodes only after the application of `fact 2`. (The sequencing combinator s_1 `then` s_2 performs the node selection s_2 to each subtrace whose root matches the result of selection s_1 and then merges the results of the s_2 selections.)

```
hide (funDef ◇)
factor binding except fun
hide fact2 then middleFact
...
```

```
funDef f ≡ let f = ◇ in ◇ ⇓◇
fun      ≡ ◇:f = ◇
fact2    ≡ fact 2⇓◇
```

Note that `fact 2` needs to be computed twice in the above program. Thus, a trace that is based on a proof tree would have two occurrences of the subtrace for `fact 2`⇓2. However, explaining the same computation more than once does not provide any additional benefit. On the contrary, the extra space requirement is detrimental to an effective explanation. To address this problem, we represent traces as DAGs. Here node G is a shared premise of nodes D and F. To avoid potential clutter caused by DAG edges, we decided to represent multiple edges to the same premise by showing nodes as a reference.

We can also observe another benefit of our Call-By-Named-Value semantics in this trace. Standard Call-By-Value would have evaluated the expression in node C to `\x->f (f x)` where `f` is bound to the definition of `fact`. The judgement in node D would then be `f (f 2)`⇓2 which is semantically correct, but can be confusing, since the introduction of the alias `f` for `fact` causes an indirection that has to be tracked by the user. Also, when `f` is applied, the reference to `f` would be replaced by its value, the definition of `fact`, leading to a more complex trace.

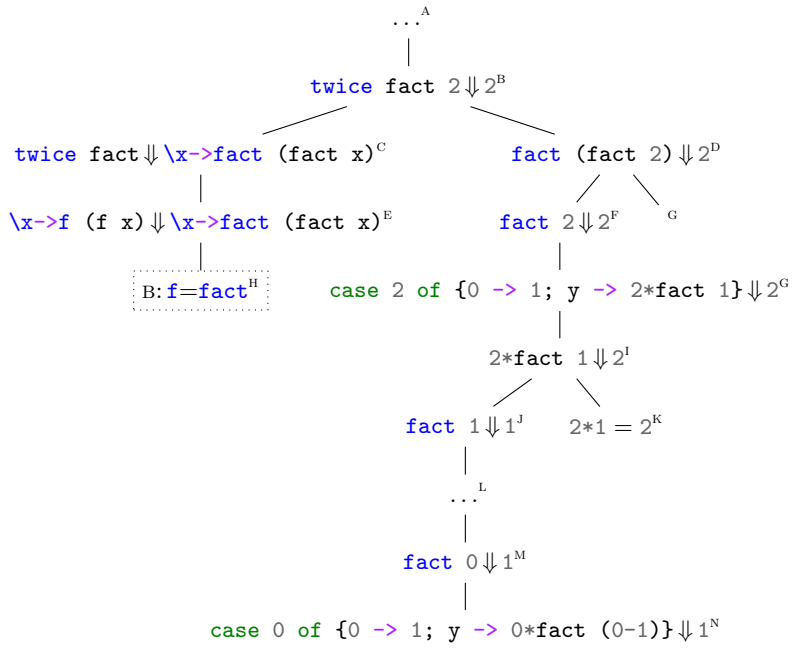

Fig. 3. Trace view for `twice fact 2`

Finally, we can observe how bindings are represented in traces using the aforementioned binding nodes instead of as part of environments. Node H shows that `f` is bound to the function `fact` (again showing the name instead of the definition) and that the binding was generated by node B. The concept of binding nodes allows us to omit environments in evaluation judgments, and our Call-By-Named-Value semantics save us from the need to use closures as function values. The main contributions of this paper are the following.

- A new *Call-By-Named-Value* semantics that facilitates the creation of parsimonious traces by employing names for values (Section 2). We show that Call-By-Named-Value is a conservative extension of Call-By-Value that can generate smaller traces.
- A DAG structure for *traces* that substitutes binding nodes for environments (Section 3) and uses operations for trace simplifications (Section 4). We show that the evaluation traces preserve the explanatory content of proof trees at a much smaller footprint.
- A notion of *trace view* that encapsulates the contraction of subtraces into single nodes, plus corresponding operations for producing trace views through the hiding and factoring of judgments, which preserve the explanatory content of traces. We show that the trace operations produce residual explanations that can be expected from the corresponding trace simplifications.

$$
\begin{array}{rrcl}
u, v \in & \mathit{Val} & ::= & c\ \nu\ \ldots\ \nu \mid \texttt{\textbackslash}x\texttt{->}e \mid \texttt{fix}(\texttt{\textbackslash}x\texttt{->}e, f) \\
p \in & \mathit{Pat} & ::= & c\ p \ldots p \mid x \\
\nu \in & \mathit{NVal} & ::= & v^{x\ \cdots\ x} \\
e \in & \mathit{Expr} & ::= & x \mid e\ e \mid \texttt{case}\ e\ \texttt{of}\ \{p\texttt{->}e\texttt{;}\ \ldots\ \texttt{;}\ p\texttt{->}e\} \mid \texttt{let}\ x\texttt{=}e\ \texttt{in}\ e \mid e\ \mathit{op}\ e \mid \nu
\end{array}
$$

Fig. 4. Expressions, Patterns, and (Named) Values

– A *trace query language* that supports the modular definition of reusable, expressive filters for trace simplifications (Section 5). The trace query language turns basic trace transformations into comprehensive strategies for simplifying traces.

After discussing related work in Section 6, we present conclusions in Section 7 where we also comment on future work and briefly report results from an evaluation of the space savings that can be achieved by trace views.

2 Call-By-Named-Value Semantics

Our object language is the untyped lambda calculus, extended by numbers and algebraic data types (see Figure 4). We use c to represent integers and constructor names and x, y, and f for variables. A *pattern* is a constructor applied to a (potentially empty) list of patterns or a variable. A *value* is a constructor applied to a (potentially empty) list of (named) values, a function or a fixpoint construction. A *named value* (ν) is a plain value which has a (possibly empty) sequence of names attached to it, written as $v^{x_1 \cdots x_k}$. The names have no semantic significance but will be used to make traces shorter and more readable. A *binding* is a pair of a name and a named value $x{=}\nu$, and an environment ρ is a sequence of bindings. Environments are extended on, and searched from, the right end.

The semantics of our language are defined in Figure 5 through rules for the judgment $\rho : e \Downarrow v^{\bar{x}}$. Notably, our definition uses named values in addition to plain values. Otherwise, the rules are a variation of Call-By-Value, and we call the semantics therefore *Call-By-Named-Value* (*CBNV*).

Names are attached to values when they are retrieved from the environment (in rule Var). By repeatedly binding a value to different variables, the value can accumulate a list of attached names (or "aliases"). Named values lose their attached names in basic computations as described in rule BinOp. Another departure from ordinary Call-By-Value is that we use plain lambda expressions instead of closures to represent function values. The purpose, again, is to achieve simpler traces. In rules Abs and Fix we substitute all free variables in abstraction bodies (except x and f) by their bound values in ρ. This is done using the environment as a function $\rho_{\bar{x}}(e)$, defined as follows.

$$
\begin{aligned}
\rho_{\bar{x}}(e) &= [\nu_1/x_1, \ldots, \nu_k/x_k]e \\
&\quad \textbf{where } \rho|_{dom(\rho)-\bar{x}} = \{\nu_1{=}x_1, \ldots, \nu_k{=}x_k\}
\end{aligned}
$$

$$\textsc{Con}\ \ \rho : c \Downarrow c \qquad \textsc{Var}\ \frac{x{=}v^{\bar{y}} \in \rho}{\rho : x \Downarrow v^{\bar{y}x}} \qquad \textsc{BinOp}\ \frac{\rho : e_1 \Downarrow v_1^{\bar{x}} \quad \rho : e_2 \Downarrow v_2^{\bar{y}} \quad v_1\ op\ v_2 = v}{\rho : e_1\ op\ e_2 \Downarrow v}$$

$$\textsc{AppF}\ \frac{\rho : e_1 \Downarrow (\backslash x\texttt{->}e')^{\bar{f}} \quad \rho : e_2 \Downarrow u^{\bar{y}} \quad \rho, x{=}u^{\bar{y}} : e' \Downarrow v^{\bar{x}}}{\rho : e_1\ e_2 \Downarrow v^{\bar{x}}} \qquad \textsc{AppC}\ \frac{\rho : e_1 \Downarrow c\ \bar{v}^{\bar{y}} \quad \rho : e_2 \Downarrow v^{\bar{x}}}{\rho : e_1\ e_2 \Downarrow c\ \bar{v}^{\bar{y}}\ v^{\bar{x}}}$$

$$\textsc{AppFix}\ \frac{\rho : e_1 \Downarrow \texttt{fix}(\backslash x\texttt{->}e, f)^{\bar{g}} \quad \rho : e_2 \Downarrow u^{\bar{y}} \quad \rho, x{=}u^{\bar{y}} : [\texttt{fix}(\backslash x\texttt{->}e, f)^{\bar{g}}/f]e \Downarrow v^{\bar{x}}}{\rho : e_1\ e_2 \Downarrow v^{\bar{x}}}$$

$$\textsc{Case}\ \frac{\rho : e \Downarrow u^{\bar{y}} \quad u^{\bar{y}} | p_i \rightsquigarrow \rho' \quad \rho, \rho' : e_i \Downarrow v_i^{\bar{x}} \quad \nexists j.1 \le j < i \wedge u^{\bar{y}} | p_j \rightsquigarrow \rho_j}{\rho : \texttt{case}\ e\ \texttt{of}\ \{p_1\texttt{->}e_1\,;\,\ldots\,;\,p_k\texttt{->}e_k\} \Downarrow v_i^{\bar{x}}}$$

$$\textsc{Abs}\ \ \rho : \backslash x\texttt{->}e \Downarrow \backslash x\texttt{->}\rho_{\bar{x}}(e) \qquad \textsc{Fix}\ \ \rho : \texttt{fix}(\backslash x\texttt{->}e, f) \Downarrow \texttt{fix}(\backslash x\texttt{->}\rho_{x,f}(e), f) \qquad \textsc{PVar}\ \ v^{\bar{y}} | x \rightsquigarrow x{=}v^{\bar{y}}$$

$$\textsc{PCon}\ \frac{v_1^{\bar{x}_1} | p_1 \rightsquigarrow \rho_1 \quad \ldots \quad v_n^{\bar{x}_n} | p_n \rightsquigarrow \rho_n}{c\ v_1^{\bar{x}_1} \ldots v_n^{\bar{x}_n} | c\ p_1 \ldots p_n \rightsquigarrow \rho_1, \ldots, \rho_n} \qquad \textsc{Let}\ \frac{\rho : e' \Downarrow u^{\bar{y}} \quad \rho, x{=}u^{\bar{y}} : e \Downarrow v^{\bar{x}}}{\rho : \texttt{let}\ x{=}e'\ \texttt{in}\ e \Downarrow v^{\bar{x}}}$$

Fig. 5. Big-Step Call-By-Named-Value Semantics

Since the only difference between CBNV and CBV semantics are the names attached to values, both evaluate expressions to the same results, except for possible attached names and the resolving of closures in CBNV. A closure (\x->e,ρ) can be viewed as being equivalent to its resolved form $\backslash x\texttt{->}\rho_{\bar{x}}(e)$. [1] Writing $v \approx v'$ for the extension of this relation to all values, we can express the relationship between CBV and CBNV as follows. (We ignore free variables in $\rho_{\bar{x}}(e)$, since such expressions are considered meaningless in both semantics.)

Theorem 1. $\rho : e \Downarrow^{CBNV} v^{\bar{x}} \wedge FV(v) = \varnothing \iff \rho : e \Downarrow^{CBV} v' \wedge v \approx v'$

3 From Proof Trees to Traces

We introduce a DAG model of traces for judgments $j = \rho : e \Downarrow \nu$ that eliminates duplicates, replaces variable lookups by *binding nodes*, replaces function values by their names in places they are not applied, and eliminates environments.

Let $P_j = (N, L, R, E)$ be the proof tree for j where N is a set of nodes, $L : N \to \mathcal{J}_\nu$ maps each node to the judgment it is labeled with, R maps each node to the name of the rule that was used to create it as a conclusion, and $(n, m) \in E$ iff m is a child of n in P_j. The root of P_j is labeled with j. The type of judgments $\mathcal{J}_\nu$ used in proof trees is defined in Figure 6. In addition to evaluation judgments, $\mathcal{J}_\nu$ includes pattern matching judgments, equations for binary operations, and variable lookups.

[1] Or $\texttt{fix}(\backslash x\texttt{->}\rho_{x,f}(e), f)$, if the closure is recursive.

$$
\begin{array}{lcl}
j \in \mathcal{J}_\nu & ::= & \rho : e \Downarrow \nu \;\mid\; \nu | p \rightsquigarrow b \;\mid\; v\, op\, v = v \;\mid\; x{=}\nu \in \rho \\
j \in \mathcal{J}_w & ::= & e \Downarrow w \;\mid\; w | p \rightsquigarrow b \;\mid\; v\, op\, v = v \;\mid\; \overline{n{\cdot}n} \triangleright x{=}w \;\mid\; \cdots \\
w \in W & ::= & x \;\mid\; v
\end{array}
$$

Fig. 6. Judgments stored in proof trees ($\mathcal{J}_\nu$) and traces ($\mathcal{J}_w$).

The type of judgments $\mathcal{J}_w$ used in traces is slightly different: First, evaluation judgments don't have an environment, and expressions don't evaluate to named values but to names *or* values (W). Second, variable lookups are replaced by binding nodes, and we have placeholder nodes ($\cdots$), explained in Section 4.1. Finally, all judgments only use plain values v (or names or values w) instead of named values ν. The names associated with values are exploited in the translation process to replace some of the values in $e \Downarrow w$ and $w | p \rightsquigarrow \rho$.

In the first step, we generate a DAG from the proof tree. To this end, we need an equivalence predicate on the judgments $\mathcal{J}_\nu$ used in labels. Two values v and v' are equivalent (written as $v \equiv v'$) if they are identical. The same is true for variables, patterns, and expressions. Two named values are equivalent if their plain values are, that is, $v^{\bar{y}} \equiv v^{\bar{x}}$ (ignoring names increases the opportunities for sharing). The following rules define the equivalence of environments and judgments.

$$
\frac{\nu_1 \equiv \nu_1' \quad \ldots \quad \nu_k \equiv \nu_k'}{\{x_1{=}\nu_1, \ldots, x_k{=}\nu_k\} \equiv \{x_1{=}\nu_1', \ldots, x_k{=}\nu_k'\}} \qquad \frac{\rho(e) \equiv \rho'(e') \quad \nu \equiv \nu'}{\rho : e \Downarrow \nu \equiv \rho' : e' \Downarrow \nu'}
$$

$$
\frac{\nu \equiv \nu' \quad \rho \equiv \rho'}{\nu | p \rightsquigarrow \rho \equiv \nu' | p \rightsquigarrow \rho'} \qquad v_1\, op\, v_2 = v \equiv v_1\, op\, v_2 = v \qquad \frac{\nu \equiv \nu'}{x{=}\nu \in \rho \equiv x{=}\nu' \in \rho'}
$$

The equivalence of labels induces a corresponding equivalence for nodes: $n \equiv m \Leftrightarrow L(n) \equiv L(m)$. To increase the potential for sharing, we could extend equivalence to account for α-equivalence. However, this would require to use "named variables" (similar to named values), since the transformation of traces may change the binding parent of shared computations. Such a "bound variable shift" is similar to the effect of "origin shift", explained later (cf. Figure 7). Since α-equivalence would complicate our model further, we leave it for future work.

To generate the DAG, we choose a minimal subset of N that doesn't lose any judgments, that is, we pick a smallest set $N_\equiv \subseteq N$ so that $\forall j \in rng(L). \exists n \in N_\equiv . L(n) \equiv j$. Next we redirect edges to/from nodes in $N_\equiv$.

$$
E_\equiv = \{(n, m) \mid (n', m') \in E \wedge \{n, m\} \subseteq N_\equiv \wedge n \equiv n' \wedge m \equiv m'\}
$$

Finally, we transform labels $\rho : e \Downarrow \nu \in rng(L)$ to $\rho|_{FV(e)} : e \Downarrow \nu$ to restrict ρ to the most recent bindings of free variables in e. We write $L^1_\equiv$ for the resulting labeling function. Then $G_j = (N_\equiv, L^1_\equiv, R, E_\equiv)$ is the proof DAG for j derived from P_j. We use superscripts to disambiguate the different versions of the labeling function that result from each step.

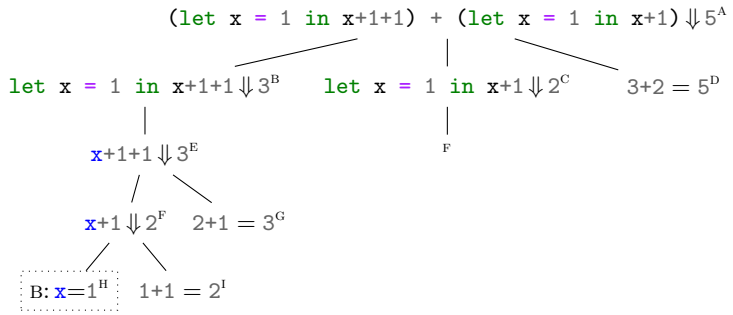

Fig. 7. Trace for `(let x = 1 in x+1+1) + (let x = 1 in x+1)`.

In the second step, we tag variable bindings in environments with the nodes of the judgments that created them, as well as with the nodes of the bindings' scopes. Since some nodes are shared, it may happen that an environment contains a binding that has more than one origin and scope. Consider the trace in Figure 7. The judgment `x+1 ⇓ 2` in node F that results from the evaluation of both `let` expressions is shared. One of its premises is the binding node H, 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, but when we transitively hide all the nodes in the subtrace with root 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 (see Section 4.1).

To tag variable bindings, we must determine which nodes produce bindings that are used by other nodes. To this end, we define a relation $O \subseteq N_{\equiv} \times N_{\equiv} \times \mathit{Var} \times N_{\equiv}$, where $(n, m, x, n_j) \in O$ means "the variable x is used in the label of node n_j, and has origin n and scope m." We first define an auxiliary relation $O' \subseteq N_{\equiv} \times N_{\equiv} \times \mathit{Var}$ that captures which nodes are considered origins and scopes. We consider two cases: (A) For a node n with $R(n) \in \{\textsc{AppF}, \textsc{Let}, \textsc{AppFix}\}$ that has a child m with label $\rho, x{=}u^{\bar{y}} : e' \Downarrow \nu^{\bar{x}}$, we have $(n, m, x) \in O'$. (B) For a node with $R(n) = \textsc{Case}$, that has a child m_1 with $R(m_1) = \textsc{PVar}$ or $R(m_1) = \textsc{PCon}$ (and thus having a label in the form $v|p \rightsquigarrow \rho'$) and a child m_2 with a label in the form $\rho, \rho' : e_i \Downarrow \nu_i^{\bar{x}}$, we have $(m_1, m_2, x) \in O'$ iff x is bound in b.

Intuitively, O' relates variables and their possible origins and scopes. However, O' is too general, since it does not include information regarding *which* occurrences of x have particular origins and scopes. We thus define a more precise O'' relation as follows.

$$O'' = \{(n, m, x, n_j) \mid (n, m, x) \in O' \wedge n_j \in \sigma_T^*(m)\}$$

In the above expression, $\sigma^*(m)$ is the set of nodes reachable from node m. The new relation adds nodes n_j that contain x as a free variable in their label. Unlike O', only occurrences of x in descendants of the scope m are included. Finally, to get O from O'', we have to account for variable shadowing.

$$O = \{(n, m, x, n_j) \in O'' \mid \forall n', m' \in N_{\equiv}.(n', m', x, n_j) \in O'' \Rightarrow m' \notin \sigma_T^*(m)\}$$

Intuitively, only the closest origins and scopes are included in O. We finally transform each label's environment to include the origin and scope information. To this end we define scope/origin pairs for a variable x and node n_j as follows.

$$\omega(x, n_j) = \{n{\cdot}m \mid (n, m, x, n_j) \in O\}$$

With this definition we then extend every binding $x{=}v$ in an environment ρ by the origin/scope information, yielding $\omega(x, n_j){\triangleright}x{=}v$. We thus evolve $L^1_{\equiv}$ as follows.

$$L^2_{\equiv}(n_j) = \{\omega(x, n_j){\triangleright}x{=}v \mid x{=}v \in \rho\} : e \Downarrow \nu$$

The scope node of a binding is the top-most node in which that binding is available. For some operations on traces, it's necessary to know the set of *all* nodes in which a binding is available, which is given by the following function.

$$S_x(m) = \sigma^*_T(m) - \bigcup\{\sigma^*_T(m') \mid (n', m', x, n'_j) \in O \wedge m' \in \sigma^*_T(m)\}$$

$S(m)$ includes all descendants of m, except those that are descendants of another scope node m' further down the tree, to account for shadowing.

At this point, the environments contain information that will help us tailor our traces later on. In particular, for a judgement of the form $\rho : e \Downarrow \nu$, the scopes of the variables bound in ρ can be used to determine nodes where e can be safely replaced with ν. We capture this information in a function $\eta(n)$, which is defined to work on evaluation judgments in the current mapping $L^2_{\equiv}$.

$$\eta(n) = \begin{cases} \bigcap_{\bar{o}{\triangleright}x{=}\nu' \in \rho} \bigcup_{n'{\cdot}m \in \bar{o}} S_x(m) & \text{if } L^2_{\equiv}(n) = \rho : e \Downarrow \nu \\ \varnothing & \text{otherwise} \end{cases}$$

This definition ensures that an expression will be replaced by its value only in labels of nodes where all free variables are defined and have the same value. If no variables are bound in ρ, then the scope of a node is the entire trace, or $N_{\equiv}$.

In the third step, we replace applications of the VAR rule by binding nodes: For each node $k \in N_{\equiv}$ with $R(k) = \textsc{Var}$ and $L^2_{\equiv}(k) = \rho : x \Downarrow v^{\bar{y}x}$ where $\bar{o}{\triangleright}x{=}v^{\bar{y}} \in \rho$, we change $L^2_{\equiv}$ to $L^3_{\equiv}(k) = \bar{o}{\triangleright}x{=}v$ and remove the node m with $(k, m) \in E_{\equiv}$ as well as the edge (k, m).

In the final step, we replace named values by names or values. Specifically, we replace named functions by their names and remove names from other values, that is, we replace all $(\backslash x \texttt{->} e)^{z\bar{y}}$ by z (the first name assigned to the function) and $c^{\bar{x}}$ by c. As a special case, if the first name z is equal to the variable being bound, we instead use the second name (the first element of $\bar{y}$) if it is available. This is a very simple strategy, but CBNV offers opportunities to explore more refined replacement rules based on properties of the trace and guided by annotation from the user. We also eliminate the environments from all evaluation judgments, since the origins of variable bindings are captured by their node tags. This yields $L^4_{\equiv}$.

After the transformation steps, we can drop the rule labels R, since they are no longer needed. We call the resulting DAG $T_j = (N_{\equiv}, L^4_{\equiv}, E_{\equiv}, \eta)$ a *trace* for

the judgment j. In the following we simply use N instead of $N_{\equiv}$ (same for E and L) and consider the "$\equiv$" subscript as implicitly present.

In the rest of the paper we use the following notation for accessing specific parts of traces. In the definition of $T[n]$, the notation $L[n]$, $E[n]$, and $\eta[n]$ is used to denote restrictions of the sets to element to only include nodes in $\sigma^*_T(n)$.

$$\begin{aligned}
\hat{T} &\qquad && \textit{root node of trace } T\\
\sigma_T(n) &= \{m \mid (n,m) \in E\} && \textit{direct premises of node } n\\
\sigma^*_T(n) &= \{m \mid (n,m) \in E^*\} && \textit{direct \& indirect premises of node } n\\
T[n] &= (\sigma^*_T(n), L[n], E[n], \eta[n])\} && \textit{subtrace of } T \textit{ with root } n
\end{aligned}$$

Traces represent comprehensive explanations of program executions that are subject to systematic transformations using two specicific trace operations, to be introduced next.

4 Trace Views as Explanations

As described in [5], proof trees can be viewed as explanations. Specifically, the judgment in each node is explained by the judgments in its children. In the context of operational semantics, a rule defines what counts as a valid explanation of a judgment, that is, in a rule $P_1, \ldots, P_n \implies C$ the premises $P_1, \ldots, P_n$ explain the conclusion C in the sense that the correct answer to the question "Why is C true?" is: "Because P_1, P_2, etc." A proof tree is comprehensive as an explanation of the judgment at its root, since it contains explanations for all judgments in internal nodes that might themselves be in need of an explanation.

By strictly following the rules of the semantics in building a proof tree we also ensure that the proof tree provides a *correct explanation*. This seems to be obvious, but it is important to point out that an explanation could, in principle be incorrect, and since we will use transformations of explanations in the following, we need to guard against the construction of incorrect ones. There are several ways in which an explanation can be incorrect. First, an explanation could contain an incorrect judgment. For example, `3` $\Downarrow$ `4` cannot be derived by the rules and thus makes any explanation it is used in incorrect. Second, an explanation could contain a correct judgment that doesn't match the rule used for building the explanation. For example, the correct explanation for `3+4` $\Downarrow$ `7` contains the three premises `3` $\Downarrow$ `3`, `4` $\Downarrow$ `4`, and `3+4` $=$ `7`. If the first premise were replaced by `2` $\Downarrow$ `2` or the third premise were replaced by `2+5` $=$ `7`, the resulting explanation would contain only correct judgments, but it would still be an incorrect explanation. Finally, an explanation could contain extra judgments that, while true, don't contribute anything to the explanation. An example would be to add a fourth premise `7` $\Downarrow$ `7` to the explanation of `3+4` $\Downarrow$ `7`.

While the construction of traces described in Section 3 does change the structure of judgments and turns the tree into a DAG, it doesn't change the factual statements of the judgments, and it doesn't omit any facts either.

Proposition 1 *The trace T_j derived from a proof tree P_j for a judgment j is a correct and comprehensive explanation for j.*

In the following we write $T \therefore j$ when T is a correct explanation for j, that is, the correctness part of Proposition 1 can be simply rephrased as $T_j \therefore j$.

While we always want to have correct explanations, we do not necessarily need comprehensive explanations. Specifically, we don't need an explanation for a judgement that is well understood. A non-comprehensive explanation might be often even preferable to a comprehensive one, since it is smaller and can thus be understood more easily.

4.1 Hiding Judgments and Subtraces

We can consider two main cases for simplifying traces by omitting parts: removal of leaves or complete subtrees (or sub-DAGs), and removal of (one or more) internal nodes. The latter requires redirecting its incoming and outgoing edges, which may cause incorrect explanations. Consider, for example, the explanation for $\rho : \texttt{succ (succ 1)} \Downarrow \texttt{3}$, which according to the AppF rule has three premises, (A) $\rho : \texttt{succ} \Downarrow \texttt{\textbackslash x->x+1}$, (B) $\rho : \texttt{succ 1} \Downarrow \texttt{2}$, and (C) $\rho, \texttt{x} = \texttt{2} : \texttt{x+1} \Downarrow \texttt{3}$. If we remove (B) and replace it with its three premises (another (A), (D) $\rho : \texttt{1} \Downarrow \texttt{1}$, and (E) $\rho, \texttt{x} = \texttt{1} : \texttt{x+1} \Downarrow \texttt{2}$), the resulting explanation now has five premises, A, A, D, E, and C. Now the premises D and E do not match the premise B required by the AppF rule, and thus, the resulting trace is an incorrect explanation.

Therefore, nodes aren't removed from a trace, but rather only hidden. More precisely, they are marked as hidden, and (maximal) groups of connected hidden nodes are shown in the trace as an ellipsis ($\cdots$) when they have non-hidden premises. Given a total order $<$ on N, we can identify any connected hidden subgraph with its smallest node. With $\sim$ being the reflexive, transitive, and symmetric closure of edges from E that are between two nodes in H, we can define a function R that performs this identification as follows.

$$R(n) = \{\min([n]_\sim) \mid n \in N\}$$

Hidden sinks are subgraphs of hidden nodes with no outgoing edges, that is, $S_H = \{n \in R(N) \mid \nexists (l, m) \in E : R(l) = n \wedge R(m) \neq n\}$ The *trace view* of T induced by H is the graph $T_H = (N_H, L_H, E_H, \eta)$ where:

$$\begin{aligned} N_H &= R(N) - S_H \\ E_H &= \{(R(m), R(n)) \mid (m, n) \in E \wedge R(m), R(n) \in N_H \wedge R(m) \neq R(n)\} \\ L_H(n) &= \begin{cases} \cdots & \text{if } n \in H \\ L(n) & \text{otherwise} \end{cases} \end{aligned}$$

We use τ to range over trace views.

The two basic operations for hiding and unhiding a single node are:

$$T_H - n := T_{H \cup \{n\}} \qquad\qquad T_H + n := T_{H - \{n\}}$$

Due to their type, both operations are left associative.

A trace view is a correct explanation if all hidden nodes can be substituted by (subgraphs of) judgments so that the resulting trace is a comprehensive and correct explanation. Of course, this can be easily achieved by unhiding all the hidden nodes, that is, trace views are by construction correct explanations.

Lemma 1. $\tau \therefore j \implies \tau - n \therefore j \qquad \textit{and} \qquad \tau \therefore j \implies \tau + n \therefore j$

To make the interaction work with trace views intuitive, it is important that node hiding and unhiding is commutative.

Lemma 2. $\tau - n - m = \tau - m - n \qquad \textit{and} \qquad \tau + n + m = \tau + m + n$

Commutativity of node (un)hiding supports the incremental construction of explanations, since hiding operations can be applied and undone in arbitrary order. Hiding and unhiding are idempotent, but they are not inverses of each other, because even though unhiding a hidden node will make the node visible, hiding a node after unhiding it will still hide it in the resulting trace view.

$$\tau - n - n = \tau - n \qquad \tau - n + n = \tau$$
$$\tau + n + n = \tau + n \qquad \tau + n - n = \tau - n$$

If we could only hide individual nodes one by one, the construction of explanations would be too arduous. Since it's only natural to want to transitively hide all premises of an understood judgment, we define a corresponding operation. However, we cannot simply hide all nodes $m \in \sigma_T^*(n)$, since we shouldn't hide nodes that are still used as premises in other (non-hidden) parts of the trace view. We should hide only those descendants of n that are only reachable through n. We can gather this set of *weak descendants* through the following definition.

$$\sigma_T^\circ(n) = \{n\} \cup \{m \in \sigma_T^*(n) \mid \mathit{deg}^-_{T[n]}(m) = \mathit{deg}^-_T(m)\}$$

With the help of that function we can define the following operation for hiding a node and all of its weak descendants. Similarly, we can also define transitive functions for hiding and unhiding nodes.

$$T_H \ominus n := T_{H \cup \sigma_T^\circ(n)} \qquad T_H \oplus n := T_{H \cup \sigma_T^\circ(n)}$$

The transitive (un)hiding operations enjoy the same algebraic properties as the single-node versions of the operations.

$$\tau \ominus n \ominus n = \tau \ominus n \qquad \tau \ominus n \oplus n = \tau$$
$$\tau \oplus n \oplus n = \tau \oplus n \qquad \tau \oplus n \ominus n = \tau \ominus n$$

4.2 Applying Judgments and Factoring Traces

In some cases traces can be simplified beyond hiding. For example, understanding the judgment `length [] ⇓ 0`, we may in addition to hiding it actively employ it to replace subexpressions `length []` elsewhere by `0`. We call the use of a judgment $L(n) = e \Downarrow v$ as a rewrite rule *applying a judgment*; it is used within n's scope, that is, for the set of nodes in the trace where e is certain to be evaluated to the same result v. The function η, included in each trace, contains this scope for every node. We use η with one adjustment: We consider the scope of a

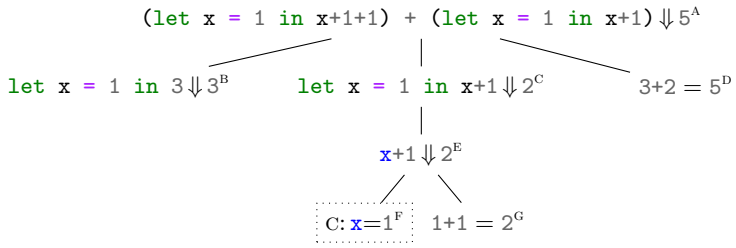

Fig. 8. Factoring judgment x+1+1 ⇓ 3 in node E (cf. Fig. 7).

lambda abstraction to be only the node in which it is evaluated, and thus avoid substituting lambda abstractions.

In the following definition we write $!m$ for the condition $m \in \eta(n) \wedge L(m) = e' \Downarrow w'$, which identifies nodes that are subject to the simplification substitution.

$$(N, L, E, \eta)_H \bullet n := (N, L', E, \eta)_H \text{ where}$$
$$L'(m) = \begin{cases} [v/e]e' \Downarrow w' & \text{if } !m \wedge L(n) = e \Downarrow w \\ [w/x]e' \Downarrow w' & \text{if } !m \wedge L(n) = \bar{o} \triangleright x{=}w \\ L(m) & \text{otherwise} \end{cases}$$

For example, to apply the judgment x+1 ⇓ 2 in node F in the trace from Figure 7, we have to substitute 2 for x+1 in the scope for x+1, which is given by the scope for variable x. Since node F is shared, the binding node for x (that is, H) has two scope nodes associated with it, namely B and C.

Applying a judgment leads to a redundant judgment of the form $w \Downarrow w$. We generally want to hide such judgments, since they don't contribute to the explanation. Therefore, we define an additional operation *factor* that applies a judgment and transitively hides it at the same time.

$$\tau \div n := (\tau \bullet n) \ominus n$$

We call the application of a factor operation *trace factorization* and also refer to the result as *factored trace*. As an example, consider the factoring of the judgment x+1+1 ⇓ 3 in node E in Figure 7: Node E and its premise G are removed from the trace. Because F, H, and I are shared as a premise of the judgment in C, they will not be removed. The factored trace is shown in Figure 8. Nodes F, H, and I from Figure 7 now appear as E, F, and G. Note that these nodes are no longer children of B. The binding in F (which was previously H) is created in C, which means that the binding node's origin has to be changed from B to C.

Unlike the hiding of nodes, which changes merely the presentation of a trace, the applying and factoring of judgments can change traces substantially through the simplification of expressions.

In particular, when the root of an explanation is affected by $e' \Downarrow w'$, such an altered trace does not explain the original judgment in the root anymore, that is, $j = e \Downarrow w$ turns into $j' = [w'/e']e \Downarrow w$, and we have $\tau \div n \therefore j'$ but *not* $\tau \div n \therefore j$.

$$
\begin{array}{lcl}
q & ::= & \texttt{hide}\ s \mid \texttt{hideAll}\ s \mid \texttt{apply}\ s \mid \texttt{factor}\ s \mid q\ \texttt{\&}\ q \\
s & ::= & \iota \mid s\ \texttt{and}\ s \mid s\ \texttt{or}\ s \mid s\ \texttt{except}\ s \mid s\ \texttt{then}\ s \mid \texttt{fix}\ s \mid \texttt{try}\ s \mid \texttt{root} \\
\iota & ::= & e^{\diamond} \Downarrow v^{\diamond} \mid v^{\diamond} | e^{\diamond} \rightsquigarrow b^{\diamond} \mid v^{\diamond}\ op\ v^{\diamond} = v^{\diamond} \mid x^{\diamond} \triangleright x^{\diamond} = v^{\diamond} \mid \diamond_a
\end{array}
$$

Fig. 9. Queries, Selectors, and Patterns

But that is, we argue, exactly what one should expect of an explanation: A residual explanation for $e \Downarrow w$ that omits everything related to explaining $e' \Downarrow w'$ is an explanation for $[w'/e']e \Downarrow w$, and that is the trace that one gets.

To formulate the formal relationship for factored traces, we write $\lceil j \rceil$ to denote the trace for a judgment j and $\frac{j}{j'}$ for $\lceil j \rceil \div n$ with $L(n) = j'$. In general, we have the following relationship for factored traces.

Theorem 2.

$$FV(e') = \varnothing \implies \frac{e \Downarrow w}{e' \Downarrow w'} = \frac{[w'/e']e \Downarrow w}{w' \Downarrow w'}$$

We can explain the idea also in terms of factoring and hiding.

Lemma 3. $L(n) = e' \Downarrow w' \wedge FV(e') = \varnothing \implies \lceil e \Downarrow w \rceil \div n = \lceil [w'/e']e \Downarrow w \rceil \ominus n$

5 Trace Query Language

The query language consists of two parts: operations and selectors. The operations are as described in the previous sections. Selectors are used to find nodes where operations should be applied. The grammars for these two components of the language are given in Figure 9; it also contains a grammar for patterns, which is similar to $\mathcal{J}_w$ from Figure 6, without the ellipsis and extended by a wildcard symbol (we use $e^{\diamond}$ to stand for e or $\diamond$, $v^{\diamond}$ to stand for v or $\diamond$, etc.). Different occurrences of an non-indexed wildcard $\diamond$ can be bound independently of one another. To force the occurrence of the same bound value in different places, the wildcard can be indexed, as for example in $\diamond_a \Downarrow \diamond_a$, which matches expressions that evaluate to themselves.

When a selector s is applied to a trace, it yields a set of nodes matching the selector, ordered according to a breadth-first traversal of the trace.

If the selector is a pattern ι, it yields the set of all nodes with matching labels (written as $l \prec \iota$). The matching relation is fairly straightforward: Each pattern ι matches the corresponding judgment, while the wildcard $\diamond$ matches anything.

A selector may also be a combination of other selectors. For instance, s_1 `or` s_2 finds nodes that are selected by either s_1 or s_2. Similarly, s_1 `and` s_2 finds nodes selected by both s_1 and s_2, and s_1 `except` s_2 will yield all nodes matched by s_1 that are not matched by s_2.

More sophisticated queries can be built with the selectors `root`, s_1 `then` s_2, `try` s, and `fix` s: `root` returns the root node of the trace, sequencing s_1 `then` s_2

applies s_2 to subtraces $T[n]$ of T for every node n selected by s_1 and merges the final results, and `fix` computes fixed points.

We can use `then` to find evaluations of the factorial function that occur as children of other evaluations of the function (indicating recursion).

$$(\texttt{fact} \diamond \Downarrow \diamond) \texttt{ then } ((\texttt{fact} \diamond \Downarrow \diamond) \texttt{ except root})$$

Here `except root` ensures that only the children of the function application are selected, and not the parent application itself. Here are a few more frequently used general-purpose selectors (where `all` $= \diamond$ is just a convenient alias):

$$\begin{array}{ll} \texttt{none} = \texttt{root except root} & \texttt{descendants} = \texttt{all except root} \\ \texttt{first } s = s \texttt{ except } (s \texttt{ then } (s \texttt{ except root})) & \texttt{children} = \texttt{first descendants} \end{array}$$

The selector `first` s will find all nodes selected by s that are not children of other nodes also selected by s. The selectors `descendants` and `children` find the transitive and immediate children of the trace's root node, respectively.

The combinator `then` in itself cannot be used to define more complicated traversals of a trace. Instead, the `fix` selector can be used to perform an arbitrary number of sequencing operations. It works by repeatedly sequencing s with itself until the result of the sequencing stops changing. To help avoid fixed points where no nodes are selected, the `try` s combinator can be used, which returns the root of the graph if s does not select any nodes. This way, we are able to terminate the sequencing right before hitting an empty result, rather than after. We can use `fix` to find all nodes that are used outside of a call to the factorial function.

$$\texttt{fix } ((\texttt{children or root}) \texttt{ except } (\texttt{fact} \diamond \Downarrow \diamond))$$

The new combinator enables us to define a few more general purpose selectors.

$$\begin{array}{rl} \texttt{last } s = & \texttt{fix } (\texttt{try } (s \texttt{ except root})) \\ \texttt{uniqueChildren } s = & (s \texttt{ then all}) \texttt{ except } (\texttt{fix } ((\texttt{children or root}) \texttt{ except } s)) \end{array}$$

The `last` s selector will find all nodes selected by s that do not have other nodes selected by s as descendants, while `uniqueChildren` s will find descendants of nodes selected by s that are not referenced anywhere else in the trace.

The semantics for the selector language are given in Figure 10. We can now use the arsenal of selectors to construct specialized queries to help with creating explanations. For instance, we may want to hide all evaluations of a recursive function except for the first and last one. This can be achieved with the `limitRec` f selector, defined as follows, where f is the name of the recursive function.

$$\begin{array}{l} \texttt{nonFirst } s = (s \texttt{ except first } s) \texttt{ then descendants} \\ \texttt{afterLast } s = \texttt{last } s \texttt{ then all} \\ \texttt{limitRec } f = \texttt{notFirst } (f \diamond \Downarrow \diamond) \texttt{ except afterLast } (f \diamond \Downarrow \diamond) \end{array}$$

We can now define the semantics of queries as a transformation of trace views through the operations introduced in Section 4. (Note that for any ordered set

$$
\begin{array}{ll}
[\![\iota]\!](L,N,E) & = \{n \mid L(n) \prec \iota\} \\
[\![s_1 \texttt{ and } s_2]\!]T & = [\![s_1]\!]T \cap [\![s_2]\!]T \\
[\![s_1 \texttt{ or } s_2]\!]T & = [\![s_1]\!]T \cup [\![s_2]\!]T \\
[\![s_1 \texttt{ except } s_2]\!]T & = [\![s_1]\!]T - [\![s_2]\!]T \\
[\![\texttt{root}]\!]T & = \{\hat{T}\}
\end{array}
\qquad
\begin{array}{ll}
[\![s \texttt{ then } s']\!]T & = \bigcup_{n \in [\![s]\!]T} [\![s']\!](T[n]) \\
[\![\texttt{fix } s]\!]T & = \begin{cases} [\![s]\!]T & \text{if } [\![s \texttt{ then } s]\!]T = [\![s]\!]T \\ [\![s \texttt{ then } (\texttt{fix } s)]\!]T & \text{otherwise} \end{cases} \\
[\![\texttt{try } s]\!]T & = \begin{cases} [\![\texttt{root}]\!]T & \text{if } [\![s]\!]T = \varnothing \\ [\![s]\!]T & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 10. Selector Semantics

of nodes $M = \{n_1, \ldots, n_k\}$, we use for all $\odot \in \{-, \ominus, +, \oplus, \bullet, \div\}$ the notation $T \odot M$ as an abbreviation for $T \odot n_1 \odot \ldots \odot n_k$).

$$
\begin{array}{rlrl}
[\![\texttt{hide } s]\!]T & = T - [\![s]\!]T & [\![\texttt{apply } s]\!]T & = T \bullet [\![s]\!]T \\
[\![\texttt{hideAll } s]\!]T & = T \ominus [\![s]\!]T & [\![\texttt{factor } s]\!]T & = T \div [\![s]\!]T \\
[\![q_1 \texttt{ \& } q_2]\!]T & = [\![q_2]\!]([\![q_1]\!]T)
\end{array}
$$

Note that the `uniqueChildren` combinator was not created arbitrarily; in fact, its behavior closely aligns with that of the $\sigma^{\circ}_T(n)$ function (defined in Section 4.1). We capture this in the following lemmas (where Lemma 5 is a direct consequence of Lemma 4).

Lemma 4. $[\![\texttt{uniqueChildren } s]\!]T = \bigcup_{n \in [\![s]\!]T} \sigma^{\circ}_T(n)$

Lemma 5. $[\![\texttt{hideAll } s]\!]T = T - [\![\texttt{uniqueChildren } s]\!]T$

6 Related Work

Our *Call-By-Named-Value* semantics is similar to the work of Acar et al. [1], in which the semantics of a language are extended to support provenance. They use a fixed algorithm for *disclosure slicing* to reduce the size of traces, whereas our approach allows tailoring of traces through a query language.

Problems with the visualization of large proof trees has been addressed Dunchev et al. [3] through hiding structural rules (similar to our Var rule) and unused contexts (similar to our hidden environments). Their *Prooftool* allows users to focus on sub-proofs, similar to our hiding and factoring operations. They also discuss the use of proof DAGs but decided against them because of the difficulty of finding graph layouts that avoid crossing edges.

Proof trees are universal structures to trace arbitrary programs. A different kind of structure called *value decomposition* was introduced in [4] for explaining the execution of dynamic programming algorithms. This approach is based on a semiring model of dynamic programming, and while it can produce succinct explanations, it is limited to only a small set of programs.

The work on explaining (imperative) functional programs [12, 13] employs program slicing as a technique to generate dynamic explanations for the part of

output selected by the user. Program slicing filters out parts from traces that do not lead to the selected partial output. The approach assumes that a user would like to understand certain part of the output, which isn't always the case. Also, the generated traces can still be quite large. Our approach is somewhat orthogonal and could in principle be combined with program slicing techniques.

The idea of *algorithmic debugging* is to incrementally tailor a proof tree for a computation by repeatedly asking users about the expected results of subexpression evaluations [10]. Like most other debugging approaches, the goal is not to provide any explanation of why the output was generated in the first place.

The *Java Whyline* [8] is a debugger for Java program that allows programmers to ask questions about the output, which the debugger tries to answer by computing a backward trace of the computations that caused the output. Haskell's debugger Hood [6] generates a trace of intermediate values of computation. A programmer needs to annotate the interesting parts in the source code. When the code is recompiled and rerun, the debugger results in a trace of intermediate values along with the actual output. This is similar to our approach in that the user has some control over the form and size of the produced trace.

The selector component of our query language was inspired by the *rewrite strategies* approach presented by Visser et al. [16], which are used to define algorithms that apply optimizations to programs. Much like our query language, these strategies provide a toolbox of combinators that allow the user of the system to construct more complicated traversal and transformation algorithms. The selectors defined in this paper are more linmited: they lack the ability to transform the trace, and they are not capable of maintaining a context of encountered expressions, or making decisions based on that context.

7 Conclusions and Future Work

We have presented a new approach for explaining program behavior that is based on a new Call-By-Named-Value semantics, a DAG-based representation for traces, and a query language for expressing trace manipulations. A major innovation of our traces is the economical presentation of information and an effective method for hiding large parts of uninteresting regions from a trace.

Our initial experiments with this new approach are encouraging: For a benchmark set of 21 functional programs used in an introductory CS course, we could achieve reductions between 79% and 98% (in 90% of the cases the traces have been reduced by 85% or more). For this we needed 12 standard filters (7 of these were always applied and 5 only in specific instances). (Details about this evaluation, can be found in Bajaj et al. [2].)

In future work, we can make explanation traces even more succinct through dead-code elimination, especially within expressions. For example, the `True` branch in a judgment `case False of {...; False -> 0}` $\Downarrow$ `0` can be omitted. This strategy is applicable even if part of the code is not dead but "dormant" and thus explanatorily irrelevant in the current part of the trace. Moreover, we can further exploit the Call-By-Named-Value semantics by having users tag impor-

tant names, and we can also exploit the fact that named values can have an arbitrary number of names. Instead of always displaying one specific name, the decision can be made dynamically, for example, when a value acquires a more meaningful name in the evaluation of a program (such as when list elements acquire the name `pivot` during the execution of quicksort).

Acknowledgements

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

References

1. Acar, U.A., Ahmed, A., Cheney, J., Perera, R.: A Core Calculus for Provenance. In: Int. Conf. on Principles of Security and Trust. pp. 410–429 (2012)
2. Bajaj, D., Erwig, M., Fedorin, D., Gay, K.: A Visual Notation for Succinct Program Traces. In: IEEE Int. Symp. on Visual Languages and Human-Centric Computing (2021), to appear
3. Dunchev, C., Leitsch, A., Libal, T., Riener, M., Rukhaia, M., Weller, D., Woltzenlogel-Paleo, B.: Prooftool: a gui for the gapt framework. Electronic Proceedings in Theoretical Computer Science **118**, 1–14 (Jul 2013)
4. Erwig, M., Kumar, P.: Explainable Dynamic Programming. Journal of Functional Programming **31**(e10) (2021)
5. Ferrand, G., Lesaint, W., Tessier, A.: Explanations and proof trees. Computing and Informatics **25**, 105–125 (2006)
6. Gill, A.: Debugging Haskell by observing intermediate data structures. Electronic Notes in Theoretical Computer Science **41**(1), 1 (2001)
7. Khoo, Y.P., Foster, J.S., Hicks, M.: Expositor: Scriptable Time-travel Debugging with First-class Traces. In: Int. Conf. on Software Engineering. pp. 352–361 (2013)
8. Ko, A.J., Myers, B.A.: Finding causes of program output with the java whyline. In: SIGCHI Conf. on Human Factors in Computing Systems. pp. 1569–1578 (2009)
9. Marceau, G., Cooper, G.H., Spiro, J.P., Krishnamurthi, S., Reiss, S.P.: The Design and Implementation of a Dataflow Language for Scriptable Debugging. Automated Software Engineering **14**(1), 59–86 (2007)
10. Nilsson, H., Fritzson, P.: Algorithmic Debugging for Lazy Functional Languages. Journal of Functional Programming **4**(3), 337–369 (1994)
11. Parnin, C., Orso, A.: Are Automated Debugging Techniques Actually Helping Programmers? In: Int. Symp. on Software Testing and Analysis. pp. 199–209 (2011)
12. Perera, R., Acar, U.A., Cheney, J., Levy, P.B.: Functional Programs That Explain Their Work. In: ACM Int. Conf. on Functional Programming. pp. 365–376 (2012)
13. Ricciotti, W., Stolarek, J., Perera, R., Cheney, J.: Imperative Functional Programs that Explain their Work. Proc. ACM Prog. Lang. **1**(ICFP) (2017)
14. Roehm, T., Tiarks, R., Koschke, R., Maalej, W.: How do professional developers comprehend software? In: Int. Conf. on Software Engineering. pp. 255–265 (2012)
15. Vessey, I.: Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols. IEEE Trans. on Systems, Man, and Cybernetics **16**(5), 621–637 (1986)
16. Visser, E., Benaissa, Z., Tolmach, A.: Building Program Optimizers with Rewriting Strategies. In: ACM Int. Conf. on Functional Programming. pp. 13–26 (1998)