Abstract
We present techniques that enable higher-order functional computations to "explain" their work by answering questions about how parts of their output were calculated. As explanations, we consider the traditional notion of program slices, which we show can be inadequate, and propose a new notion: computation slices. We present techniques for specifying flexible and rich slicing criteria based on partial expressions part of which are replaced by holes. We formulate program slices in an algorithm independent fashion and show that least (minimal) slices exist. We then present an algorithm, called unevaluation, for computing least program slices from computations reified as traces. Observing limitations of program slices, we develop the notion of trace slices as another form of explanation and present an algorithm for computing them. We design traces (trace slices) to reflect closely the structure of programs (program slices) and such that the unevaluation algorithm can be applied to any subtrace of a trace (slice) to compute an program (slice) whose evaluation generates that subtrace. This close correspondence between programs, traces, and their slices enables the programmer to understand a computation interactively in terms of the programming language in which the computation is expressed. This enables employing the programming language as the medium for understanding computation. We present an implementation as a tool, discuss some important practical implementation concerns and techniques for addressing them.

1. Introduction
Many problem domains in computer science require understanding a computation and how a certain result was computed. For example, in debugging we aim to understand why some erroneous result was computed by a program.

This goal of understanding and explaining computations (and their results) often runs against our desire to treat computation as a black box that maps inputs to outputs. Indeed, we lack rich general-purpose techniques and tools for understanding and explaining computations and their relationship to their outputs. One such technique is program slicing, which was first explored in the imperative programming community.

Originally formulated by Weiser [22], a program slice is a reduced program, obtained by eliminating statements from the original program, that produces the behavior specified by a slicing criterion. A slice is called static if it makes no assumptions on the input—it works for any input—and dynamic if it works only for the specified input. Originally defined to yield backward slices, which identify parts of a program contributing to a specific criterion on the output, slicing techniques have later been extended to compute forward slices based on a program criterion. Since it pertains to a specific execution and since it explains the relationship between a computation and its result, dynamic and backward slicing is considered more relevant for the purposes of understanding a computation. In this paper, we consider dynamic, backward slicing only.

While primarily motivated by debugging and program comprehension, other applications of slicing have also been proposed, including parallelization [3], software maintenance [4], and testing [2]. For a comprehensive overview of the slicing literature, we refer the reader to existing surveys [21, 23].

While slicing has been studied extensively in the context of imperative languages, there is relatively little work in functional languages. Slicing techniques developed for imperative programs do not translate well to the functional setting for two reasons. First, in the functional setting, higher-order values are prevalent and slicing techniques must be adapted to handle them correctly. Second, functional programs typically manipulate values of complex recursive data types, whereas program slicing techniques in the imperative world have often considered only variables.

Reps and Turnidge [17] present slicing technique for functional programs. Their techniques compute static slices, however, and apply only to first-order functional programs; they also do not preserve the semantics of strict functional programs. Biswas [5] considers strict, higher-order functional programs and proposes slicing techniques with a single form of criterion: the entire output of the program. Ochoa, Silva, and Vidal [16] present techniques that allow more flexible criteria but which only apply to first-order, lazy functional languages. While the previous work on slicing of functional programs made significant progress, the problem of slicing of strict, higher-order functional programs has remained an open problem.

In this paper, we solve this problem for the specific case where program slices are defined as expressions with missing subexpressions. Other, richer notions of program slice appear feasible but our definition of slice is consistent with previous work.

While program slices are valuable in explaining computation, they can be quite imprecise, because each expression in a program slice must subsume the many different behaviors that the expression exhibits in an execution. For example, even though program slices were motivated by debugging, they are of limited assistance when the bug involves different execution paths along the same piece of code, which a program slice cannot distinguish. In the case of functional programs where higher-order features allow complex computations to be very concisely expressed, this limitation is all the more problematic.

In this paper, we present techniques for slicing higher-order strict programs with flexible slicing criteria and present a more powerful computation-slicing technique. We express flexible slicing criteria as partial values and differential (partial) values. A partial value is a value where some sub-values may have been replaced by a hole, written $\square$. Intuitively, a partial value specifies the parts of a value that may be of no interest to the user. Complementary to
partial values, we introduce differential (partial) values to isolate a part of a value of particular interest. For example, the partial value \(\mathbf{□}, 2\) specifies a pair where the first part is of no interest, and the partial value \(\text{cons}(\mathbf{□}, \text{cons}(2, \mathbf{□}))\) specifies a list where the first element and the second tail is of no interest. The differential value \(\text{cons}(\mathbf{□}, \text{cons}(2, \mathbf{□}))\) specifies a list which identifies the second element to be of particular interest.

As forms of explanation for (differential) partial values, we propose two techniques: program slices, and computation slices. A partial program is a program where some subexpressions may have been replaced by holes. We give an operational semantics for partial expressions, and we call a partial program a program slice for a partial output value if executing the partial program yields that partial value. We present an unevaluation algorithm for computing the least slice for a partial value. Intuitively, the unevaluation algorithm pushes the partial value backward through the computation, taking care to construct least partial values for sub-computations along the way. To make unevaluation possible, we present a tracing semantics that reifies computations as traces. We prove that our unevaluation algorithm yields least partial programs.

While our program slices can give valuable explanations about a computation, as with any program slicing technique, they can lose their effectiveness in complex computations. As an example, we show how program slices become ineffective when trying to understand a buggy mergesort implementation. As a more precise mechanism, we introduce the notion of a partial trace as a trace whose parts may have been replaced by holes. We design our traces to resemble expressions closely: syntactically they are essentially unrolled expressions.

We say that a partial trace explains a partial value if the trace contains enough information to unevaluate the partial value. We then present a trace-slicing algorithm for computing the least trace which explains a partial value. The trace-slicing algorithm uses unevaluation to calculate least slices of the bodies of higher-order values.

Our techniques offer an interactive and precise approach to understanding computations. The user can execute a program with some input and then compute a program slice for some slicing criterion expressed as a partial value. If interested in understanding the computation in more depth, the user can compute a trace slice, which will present a precise description of how the computation has unfolded, eliminating subtraces that do not contribute to the slicing criterion and highlighting those that contribute. Since the trace slices reflect closely the program code itself, the user can read the trace slice as an unrolled program. Moreover, explanations (in the form of slices) are couched in the programming language that the computation itself is expressed in.

We present a tool, Slicer, that enables interactive tracing and computation exploration for source code written in an ML like language that we call TML (Transparent ML). Slicer itself is implemented in Haskell and takes advantage of the lazy evaluation strategy of Haskell to avoid constructing traces except when needed. Inspired by Haskell’s laziness, we present a technique for trading space for time by a form of controlled lazy evaluation that can also be used in the context of strict languages. Slicer enables visualizing partial programs and traces, making it possible to visualize and understand complex computations in the language that they are expressed in.

Our contributions include the following.

- Techniques for specifying flexible slicing criteria based on differential and partial values.
- An algorithm-independent formulation of least program slices.
- An algorithm, called unevaluation, for computing least slices of strict, higher-order functional programs.
- The concept of computation or trace slices and an algorithm for computing them.
- Proofs of relevant correctness and minimality properties.
- The Slicer tool for computing and visualizing program and computation slices.

2. Overview

We present an overview of our techniques considering examples with increasing sophistication. We produce all our examples with an implementation of our proposed techniques as a tool, which we call Slicer. Slicer accepts programs written in TML and allows them to be executed and queried by using partial values to generate partial programs and traces. Slicer enables visualizing partial programs and traces in an interactive way, allowing the programmer to “browse” them. By default, Slicer prints out the outermost nesting level of programs and traces and hides the deeper levels under ellipses written as “...”. The user can click on ellipses to see the hidden contents. Slicer prints holes, written as \(\mathbf{□}\) throughout the formalism, as \(\mathbf{□}\), because the user can ask for what is hidden behind the hole to be displayed.

All our examples have the form

\[
\text{let fun } f \; \text{xs} = e \; \text{in } f \; i \; \text{end.}
\]

Here the expression \(e\) is the body of the function \(f\) and the function \(f\) is called with the input \(i\). Using Slicer we evaluate this piece of code and then generate a partial trace by specifying some (differential) partial value \(v\). Because of the way we structure the code and because of the way our trace-slicing algorithm works, the trace slice has the form

\[
\text{let fun } f \; \text{xs} = e' \; \text{in } t \; \text{end.}
\]

Here the \(e'\) is a partial expression of \(e\) and \(t\) is a trace slice of the evaluation \(f \; i\). In other words, the trace slice consists of the least slice of the function \(f\) that is consistent with \(v\) followed by the least trace slice of the computation running \(f\) with the specified input. The reader might expect a trace slice to contain much detailed information. Indeed, this can be so if special care is not taken.\(^1\) We have overcome this problem by establishing a close connection between the syntax and the semantics of expression (expression slices) and traces (trace slices). This allows us to represent trace slices in a style that can be viewed as an “unrolling” of an expression slice. Slicer can also display the partial value associated with a slice, which is a feature that we often utilize; such partial values are printed in shaded (gray) boxes.

2.1 Example: List Length

Consider using the standard \text{length} function to compute the length of a list with three elements:

\[
\text{let fun length } xs =
\begin{align*}
\text{case } xs \text{ of } \\
\text{N1} & \to 0 \\
\text{Cons}(x, xs) & \to 1 + \text{length } xs' \\
\end{align*}
\text{in length } (\text{Cons}(1, \text{Cons}(2, \text{Cons}(3, \text{N1})))) \text{ end.}
\]

To understand the computation, we ask Slicer to compute the partial slice for the result \(3\), which is shown in Figure 1. The interesting point about the program slice (Figure 1(a)) is that the elements of input are all replaced by a hole, because they do not contribute to the output. This is consistent with our expectation of \text{length} that its output does not depend on the contents of the input.

\(^1\)Our earlier formulations, for example, suffered from this problem, making traces extremely difficult to interpret.
The partial trace for the partial value \(\text{Cons}(\text{Nil},\text{Cons}(8,\text{Nil}))\) enhances our understanding of how the second element was computed. To understand how exactly the second element, 8, is computed from the input, we ask Slicer to isolate it by using as a slicing criterion the differential partial value \(\text{Cons}(\text{Nil},\text{Cons}(8,\text{Nil}))\). Slicer returns a trace slice where parts contributing directly to the second element are highlighted; in Figure 2(a) the sub-traces of the slice in focus are highlighted (with green). The highlighted pieces show that the second element is computed by applying the increment function and the by application of the increment function to the second element.

We generate differential trace slices by taking advantage of a monotonicity property of partial traces. Consider two partial values \(u, v\) where \(v\) is greater than \(u\) in the same sense than the trace slice for \(u\). This property allows us to compute the delta between two partial traces which have a common structure in this way by performing a trace traversal. To compute the differential trace slice for \(\text{Cons}(\text{Nil},\text{Cons}(8,\text{Nil}))\), we compute the edit between the traces \(\text{Cons}(\text{Nil},\text{Cons}(8,\text{Nil}))\) and \(\text{Cons}(\text{Nil},\text{Cons}(\text{Nil},\text{Nil}))\). It is this difference that is highlighted in Figure 2(a). For the purposes of comparison, we show the trace slice for \(\text{Cons}(\text{Nil},\text{Cons}(\text{Nil},\text{Nil}))\) in Figure 2(b). Note that the (green) highlighted parts are exactly those parts that are holes in the trace (b) that are not holes in the trace (a).

### 2.3 Example: Mergesort

Our final example is merge sort, an algorithm that significantly restructures its input to produce an output in a way that can be difficult to understand. We consider a buggy implementation and describe how Slicer can help us locate the bug. Our implementation
is entirely standard: it splits non-singleton lists into two, sorts them recursively, and merges the results to produce the sorted output list. The bug is in the merge function: "else" branch should be recursively, and merges the results to produce the sorted output list.

Figure 3: Outermost merge phase of mergesort.

To identify the bug, we slice the trace with respect to the differential value Cons(1, Cons(2, Cons(3, Nil))) isolating the second element. Unfortunately, the program slice computed by Slicer provides no information. This is an example where program slices are not, and cannot be, precise enough even when they are least slices as guaranteed by our techniques. We therefore inspect the actual trace slice.

3. A Characterisation of Program Slicing

Before we discuss traces and their role in calculating program slices, we formalize our notions of slicing criteria and program slices in the setting of a typed, call-by-value reference language with familiar functional programming constructs such as recursive types and higher-order functions. We formulate the problem of computing least (minimal) dynamic slices under flexible criteria in an algorithm-independent fashion and show that least dynamic slices exist.

3.1 The reference language

The syntax of the reference language is given in Figure 4. Types include the usual unit, sum, product and function types, plus iso-recursive types \( \rho, \tau \), type variables \( \alpha \), and primitive types \( b \). Variable contexts are defined inductively in the usual way. Expressions include the unit value \( \bot \), standard introduction and elimination forms for projection products, sums and recursive functions, roll and unroll forms for recursive types, primitive constants \( c \), and applications \( e_1 \oplus e_2 \) of primitive operations. The typing judgments \( \Gamma \vdash e : \tau \) for expressions and \( \Gamma \vdash \rho \) are given in Figure 6; the latter means that \( \rho \) is a well-formed environment for \( \Gamma \). The signature \( \Sigma \) assigns to every primitive constant \( e \) the primitive type \( c : b \in \Sigma \), and to every primitive operation \( \oplus \) the argument types and return type \( \oplus : b_1 \times b_2 \to \tau \in \Sigma \).

Evaluation for the reference language is given by a conventional call-by-value big-step semantics, shown in Figure 5. The judgment \( \rho, e \in ref \vdash v \) states that expression \( e \) evaluates in closing environment \( \rho \) to value \( v \). Values include the usual forms, plus closures \( \langle \rho, f(x) \rangle \). The choice of an environment-based semantics is deliberate: environments will be helpful later when we want to record an execution as an unrolling of the program syntax. As usual \( \hat{\oplus} \) means \( \oplus \) suitably interpreted in the meta-language.

Evaluation is deterministic and type-preserving. We omit the proofs, which are straightforward inductions.

Figure 5: Reference language: call-by-value evaluation

<table>
<thead>
<tr>
<th>( \rho, e \in ref \vdash v )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rho, e \in ref \vdash x )</td>
</tr>
<tr>
<td>( \rho, e_1 \in ref \vdash c_1 )</td>
</tr>
<tr>
<td>( \rho, e_1 \in ref \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)).e )</td>
</tr>
<tr>
<td>( \rho, e_1 \in ref \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_2 )</td>
</tr>
<tr>
<td>( \rho, e_1 \in ref \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e_1 \in ref \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e_1 \in ref \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_2 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_2 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_2 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_2 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_2 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_2 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_2 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_2 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_2 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_2 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_2 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_1 )</td>
</tr>
<tr>
<td>( \rho, e \in ref \vdash \rho(f(x)) \vdash v_2 )</td>
</tr>
</tbody>
</table>
The partial order \( \sqsubseteq \) is simply the inclusion order on these sets. For example, that \( \inl 1 \) and \( \inl 2 \) are related by \( \sqsubseteq \) comes about because the sets of paths that comprise the two expressions are similarly related:

\[
\begin{align*}
&\inl 1 \subseteq \inl 2, \\
&\inl 2 \subseteq \inl 3, \\
&\inl 2 \not\subseteq \inl 4.
\end{align*}
\]

(The child indices are shown in bold to avoid ambiguity.)

Clearly, an expression smaller than a given expression \( e \) is a variant of \( e \) where some paths have been truncated in a way which preserves prefix-closure. It is \( e \) with some sub-expressions “missing”, with the absence of those sub-expressions indicated in the conventional syntax by the presence of a \( \square \). It is natural to talk about such a truncated expression as a prefix of \( e \), and so we denote the set of such expressions by \( \text{Prefix}(e) \).

**Definition 1 (Prefix of \( e \)).** \( \text{Prefix}(e) = \{ e' \mid e' \subseteq e \} \)

What is more, the set \( \text{Prefix}(e) \) forms a finite, distributive lattice with meet and join denoted by \( \sqcap \) and \( \sqcup \). To see why, consider two prefixes \( e_1 \) and \( e_2 \) of \( e \). We can take their meet \( e_1 \sqcap e_2 \) by taking the intersection of the sets of paths that comprise \( e_1 \) and \( e_2 \); intersection preserves prefix-closure and deterministic extension, and so yields another prefix of \( e \) which is the greatest lower bound of \( e_1 \) and \( e_2 \).

Dually, we can take the join \( e_1 \sqcup e_2 \) by taking the union of the sets of paths comprising \( e_1 \) and \( e_2 \). Set union will not in general preserve deterministic extension (consider taking the union of \( \inl 1 \) and \( \inl 2 \), for example) but it does so whenever the two expressions have compatible structure. Here, \( e_1 \) and \( e_2 \) do have compatible structure because both are prefixes of \( e \). And so union also yields a prefix of \( e \), in this case the least upper bound of \( e_1 \) and \( e_2 \), as illustrated by the following example:

\[
\begin{align*}
&\inl 1 \sqcup \inl 2 = \inl 3, \\
&\inl 2 \sqcup \inl 2 = \inl 4.
\end{align*}
\]

Finally, we note that the greatest element of \( \text{Prefix}(e) \) is \( e \) itself, and the least element is \( \square \).

### 3.3 Slicing with respect to partial output

We now have a way of representing programs with missing parts: we simply replace the parts we want to delete with appropriately typed holes. What we need next is a way of saying whether the missing bits “matter” or not to some part of the output. We will do this by enriching the base language with rules that allow programs with holes in to be executed.

The intuition we want is that if we encounter a hole during evaluation, then we had better be computing a part of the output that was also unneeded. In other words, \( \psi_{\text{out}} \) is free to consume a hole, but only in order to produce a hole. We capture this informal notion by extending the \( \psi_{\text{out}} \) rules with the additional rules for propagating holes given in Figure 7. Hole itself evaluates to \( \square \), and moreover for every type constructor, there are variants of the elimination rules which produce a hole whenever an immediate sub-computation produces a hole. From now on, by \( \psi_{\text{out}} \) we shall mean the extended version of the rules.

Since evaluation can produce partial values, clearly environments may now map variables to partial values. This gives rise
Expressions $e ::= \ldots | \square$

Values $v ::= \ldots | \square$

$$\Gamma \vdash e : \tau$$

$$\vdash v : \tau$$

$$\rho, v \in \mathbb{V}_{\ref v}$$

$$\begin{align*}
\rho, e_1 \in \mathbb{V}_{\ref e_1} & \quad \rho, e_2 \in \mathbb{V}_{\ref e_2} \\
\rho, e_1 \in \mathbb{V}_{\ref e_1} & \quad \rho, e_2 \in \mathbb{V}_{\ref e_2} \\
\rho, e_1 \in \mathbb{V}_{\ref e_1} & \quad \rho, e_2 \in \mathbb{V}_{\ref e_2} \\
\rho, \text{case } e \in \mathbb{V}_{\ref e} & \quad \rho, \text{ detachment } e \in \mathbb{V}_{\ref e} \\
\rho, \text{ case } e \in \mathbb{V}_{\ref e} & \quad \rho, \text{ unroll } e \in \mathbb{V}_{\ref e}
\end{align*}$$

Figure 7: Additional rules for partial expressions

to a partial order on environments. Specifically, we overload $\sqsubseteq$ so that $\rho \sqsubseteq \rho'$ iff there exists $\text{dom}(\rho) = \text{dom}(\rho')$ and $v \in \text{dom}(\rho), \rho(x) = \rho'(x)$. For any $\Gamma$, we will write $\Box \Gamma$ for the least partial environment for $\Gamma$, viz. the partial such that $\rho(x) = \Box \Gamma$ for every $x \in \text{dom}(\Gamma)$. Again, the set $\text{Prefix}(\rho)$ forms a finite distributive lattice where the join $\rho \sqcup \rho'$ is the partial environment $\{x \mapsto \rho(x) \sqcup \rho'(x) \mid x \in \text{dom}(\rho)\}$, and analogously for meet.

It follows from the inductive definition of environments that environment extension with respect to a variable $x$ is a lattice isomorphism. Suppose $\Gamma \vdash \rho$ and $\vdash v : \tau$. Then for any $x$, the bijection $-x \mapsto -x$ from $\text{Prefix}(\rho(x)) \times \text{Prefix}(v)$ to $\text{Prefix}(\rho(x \mapsto v))$ satisfies:

$$(\rho' \sqcap \rho')[x \mapsto u \sqcup u'] = \rho'[x \mapsto u'] \sqcap \rho'[x \mapsto u']$$

and similarly for joins.

Now suppose $\rho, e \in \mathbb{V}_{\ref e}$ and some partial output $u \subseteq v$. We are now able say what it is for a partial program $(\rho', e') \in (\rho, e)$ to be a “correct” slice of $(\rho, e)$ for $u$. The idea is that if running $(\rho, e)$ produces a value $v'$ at least as big as $u$, then $(\rho', e')$ is “correct” in the sense that it is at least capable of computing the part of the output we are interested in. We say that $(\rho', e')$ is a slice of $(\rho, e)$ for $u$.

Definition 2 (Slice of $\rho, e$ for $u$). Suppose $\rho, e \in \mathbb{V}_{\ref e}$ and $u \subseteq v$. Then any $(\rho', e') \subseteq (\rho, e)$ is a slice of $(\rho, e)$ for $u$ if $\rho', e' \upharpoonright u'$ with $u' \supseteq u$.

This makes precise our intuition above: a slice for partial output $v$ is free to evaluate holes as long as the resulting holes in the output are subsumed by those already specified by $u$, the slicing criterion.

3.4 Existence of least slices

We have defined what it is to be a slice for some partial output $u$. Now let us turn to the question of whether there is a unique minimal slice for $u$. We shall see that introducing holes into the syntax, and then extending evaluation with hole-propagation rules, induces a family of Galois connections between partial programs and partial values and that this guarantees the existence of least slices.

Suppose a terminating computation $\rho, e \in \mathbb{V}_{\ref e}$. Evaluation has several important properties if we restrict the domain of evaluation to just the prefixes of $(\rho, e)$, which we will write as $\text{eval}_{\rho, e}$. The first is that $\text{eval}_{\rho, e}$ is total: the presence of a hole cannot cause a computation to get stuck, but only to produce an output with a hole in it. Second, $\text{eval}_{\rho, e}$ is monotonic. Third, $\text{eval}_{\rho, e}$ preserves meets and joins. The preservation of either meets or joins implies monotonicity; for our purposes we only need the preservation of meets, so we just state the following.

Theorem 1 ($\text{eval}_{\rho, e}$ is a meet-preserving function from $\text{Prefix}(\rho, e)$ to $\text{Prefix}(v)$). Suppose $\rho, e \in \mathbb{V}_{\ref v}$. Then:

1. If $(\rho', e') \subseteq (\rho, e)$ then $\text{eval}_{\rho, e}(\rho', e')$ is defined: there exists $u$ such that $\rho', e' \upharpoonright u \subseteq v$.

2. $\text{eval}_{\rho, e}(\rho, e) = v$.

3. If $(\rho', e') \subseteq (\rho, e)$ and $(\rho', e'') \subseteq (\rho, e)$ then $\text{eval}_{\rho, e}(\rho' \sqcap \rho'', e' \sqcap e'') = \text{eval}_{\rho, e}(\rho', e') \sqcap \text{eval}_{\rho, e}(\rho''', e''')$.

Technically, the $\text{eval}_{\rho, e}$ is a meet-semilattice homomorphism. Using this property for the cases that involve environment extension, we can now prove Theorem 1:

Proof. Part (2) of Theorem 1 is immediate from the definition of $\mathbb{V}_{\ref e}$. For parts (1) and (3), we proceed by induction on the derivation of $\rho, e \in \mathbb{V}_{\ref v}$, using the hole propagation rules in Figure 7 whenever the evaluation would otherwise get stuck, and Equation 1 for the binder cases.

Finally we are ready to show that a least slice for a given $v$ exists and give an explicit characterisation of it by considering a basic property of meet-semilattice homomorphisms. Every meet-preserving mapping $f_s : A \to B$ is the upper adjoint of a unique Galois connection. The lower adjoint of $f_s$, sometimes (confusingly) written $f^c : B \to A$, which preserves joins, inverts $f_s$ in the following minimising way: for any output $b$ of $f_s$, the lower adjoint yields the smallest input $a$ such that $f^c(a) \supseteq b$. Extensionally, the lower adjoint satisfies $f^c(b) = \bigcap \{a \in A \mid f_s(a) \supseteq b\}$.

Corollary 1 (Existence of least slices). Suppose $\rho, e \in \mathbb{V}_{\ref v}$. Then there exists a function $\text{uneval}_{\rho, e}$ from $\text{Prefix}(v)$ to $\text{Prefix}(\rho, e)$ satisfying:

$$\text{uneval}_{\rho, e}(u) = \bigcap \{ (\rho', e') \in \text{Prefix}(\rho, e) \mid \text{eval}_{\rho, e}(\rho', e') \supseteq u \}$$

Proof. Immediate from Theorem 1.

So for any terminating computation $\rho, e \in \mathbb{V}_{\ref v}$, there is a total function, which we call $\text{uneval}_{\rho, e}$, from partial values to partial programs which, for any $u \subseteq v$, yields the least slice of $(\rho, e)$ for $u$. Extensionally, $\text{uneval}_{\rho, e}(u)$ is the meet of all the slices of $(\rho, e)$ for $u$. This smallest slice is, in the parlance of the slicing literature, a dynamic slice: it pertains to a single execution, namely $\rho, e \in \mathbb{V}_{\ref v}$.

But the fact that $\text{uneval}_{\rho, e}$ is uniquely determined by $\text{eval}_{\rho, e}$ does not give us an efficient algorithm for computing it. We will turn to this in the next section.

3.5 Differential slices

As discussed in Section 2, the difference between two slices can be useful for diagnosing the cause of a problem. To focus on a partial subterm of a partial value, we can use a pair $\Delta(u, v)$ where $u \subseteq v$. Here, the inner part $u$ shows the context and the outer part $v$ shows the additional part of the value that we are more interested in. Given an algorithm for computing least slices $\text{uneval}_{\rho, e}$, we can then simply define the differential slice as $\text{diff}_{\rho, e}(\Delta(u, v)) = \Delta(\text{uneval}_{\rho, e}(u), \text{uneval}_{\rho, e}(v))$. Thus, differential slicing is straightforward once we have a slicing algorithm.
Traces \( T := \square x | c \in T_1 \otimes_{\mathcal{E}_1, e_2} T_2 | (T_1, T_2) \)

\( \text{fst} T \mid T \mid \text{inl} T \mid \text{inr} T \)

\( \text{case} T \text{ of } \{ \text{inl}(x_1), T_1; \text{inr}(x_2), e_2 \} \)

\( \text{case} T \text{ of } \{ \text{inl}(x_1), e_1; \text{inr}(x_2), T_2 \} \)

\( \text{fun} f(x).e \mid T_1 T_2 \triangleq f(x).T \)

\( \text{roll} T \mid \text{unroll} T \)

Figure 8: Syntax of traces

4. Program Slicing as Backwards Execution

In Section 3, we show that for an arbitrary prefix \( v \) of the output of a computation, there is a least dynamic program slice. To calculate the least slice for \( v \), we could in principle consider every prefix of the program, and take the meet of those large enough to compute \( v \). Clearly such an approach would not lead to a practical algorithm. Instead what we would like to do is somehow infer backtracks from unneeded parts of the output to unneeded parts of the input.

As outlined in Sections 1 and 2, we use a trace, constructed during evaluation, to do precisely this. The key idea is to record enough information during execution to allow the computation to be “rewound” and, given some partial output that serves as a slicing criterion, a partial program to be reconstructed. We call this procedure unevaluation, since it computes a program from a value. In this section, we explain how we represent and construct traces, give a definition of unevaluation, and prove that for any terminating computation \( \rho, e \) the unevaluation algorithm, supplied with a suitable trace, computes the least slice of \( \rho, e \).

4.1 Abstract syntax

A trace-based approach carries substantial overhead, which we discuss in Section 6. But the advantages are also considerable: not only is program slicing readily possible, as we shall see shortly, but slicing can be closely integrated into the activity of debugging, as we saw with the examples from our Slicer tool.

This close integration of slicing, source browsing and debugging is made possible by representing the trace as an “unrolling” of the expression. Figure 8 gives the abstract syntax of traces, which closely mirrors that of expressions. We will explain the trace forms in more detail when giving the tracing semantics in Section 4.2 below. Traces also include a hole form \( \square \), as with expressions, this induces a partial order on traces, which we again denote by \( \sqsubseteq \), and a lattice Prefix(\( T \)) of prefixes of a given trace \( T \). The typing rules for traces are given in Figure 9; the judgment \( \Gamma \vdash T : \tau \) states that \( T \) has type \( \tau \) in \( \Gamma \). When we are not concerned with \( \tau \) but only with \( \Gamma \), we sometimes write this as \( \Gamma \vdash T \). The only potentially surprising typing rule is for application traces, where \( T \) may be typed in an arbitrary \( \Gamma \) extended with bindings for \( f \) and \( x \). Note that if \( S \sqsubseteq T \), then \( S \) and \( T \) have the same type.

The Slicer implementation associates a value to every trace node, as shown in our examples earlier, but this is not necessary in order to compute slices, so we omit the value annotations from the formalism.

4.2 Tracing semantics

We now define a tracing semantics for the reference language presented earlier. The rules, given in Figure 10, are identical to those for \( \Psi_{nat} \), except that they construct a trace as well as a value. Tracing evaluation for an expression \( \Gamma \vdash e : \tau \) in environment \( \rho \) for \( \Gamma \), written \( \rho, e \downarrow v, T \), yields both a value \( v : \tau \) and a trace \( \Gamma \vdash T : \tau \) describing how the value was computed.

Before explaining what the trace records, we dispense with a few preliminary properties. Where the proofs are straightforward inductions or closely analogous to those for the reference language they are omitted. First, tracing evaluation is deterministic:

Figure 9: Typing rules for traces
4.3 Program slicing via unevaluation

We are now able to define a deterministic program slicing algorithm, called unevaluation, which utilises the information in a trace in order to run a computation backwards and recover a prefix of the original program. The definition is given in Figure 11. For a

\[
\rho, e \Downarrow v, T
\]

\[
\rho, x \Downarrow \rho(x), x
\]

\[
\rho, e_1 \Downarrow c_1, T_1
\]

\[
\rho, e_2 \Downarrow c_2, T_2
\]

\[
\rho, e_1 \oplus e_2 \Downarrow c_1 \oplus c_2, T_1 \oplus c_2, T_2
\]

\[
\rho, e_1 \Downarrow c_1, T_1
\]

\[
\rho, e_2 \Downarrow T_2
\]

\[
\rho, \text{fun } f(x).e \Downarrow \rho, \text{fun } f(x).e\]

\[
\rho, e_1 \Downarrow v_1, T_1
\]

\[
\rho, e_2 \Downarrow v_2, T_2
\]

\[
\rho[ f \mapsto v_1 ][ x \mapsto v_2 ], e \Downarrow v, T
\]

\[
v_1 = \langle p', \text{fun } f(x).e\rangle
\]

\[
\rho, e_1 \Downarrow v_1, T_1
\]

\[
\rho, e_2 \Downarrow v_2, T_2
\]

\[
\rho, c \Downarrow (v_1, v_2), T
\]

\[
\rho, e \Downarrow (v_1, v_2), T
\]

\[
\rho, \text{fst } e \Downarrow v_1, \text{fst } T
\]

\[
\rho, \text{snd } e \Downarrow v_2, \text{snd } T
\]

\[
\rho, \text{inl } e \Downarrow \text{inl } T
\]

\[
\rho, \text{inr } e \Downarrow \text{inr } T
\]

\[
\rho, \text{case } e \Downarrow \{ \text{inl}(x_1).e_1 : \text{inr}(x_2).e_2 \}
\]

\[
\rho, \text{case } T \Downarrow \text{inl}(x_1).T_1 \text{inr}(x_2).T_2
\]

\[
\rho, \text{case } e \Downarrow \{ \text{inl}(x_1).e_1 : \text{inr}(x_2).e_2 \}
\]

\[
\rho, \text{case } T \Downarrow \{ \text{inl}(x_1).T_1 \text{inr}(x_2).T_2
\]

\[
\rho, \text{roll } e \Downarrow \text{roll } v, \text{roll } T
\]

\[
\rho, \text{unroll } e \Downarrow \text{unroll } v, \text{unroll } T
\]

\[
\rho, x \Downarrow [v_1, v_2], T
\]

\[
\rho, x \Downarrow [v_1, v_2], T
\]

\[
\rho, e \Downarrow [v_1, v_2], e
\]

\[
\rho, \text{case } e \Downarrow \{ \text{inl}(x_1).e_1 : \text{inr}(x_2).e_2 \}
\]

\[
\rho, \text{case } T \Downarrow \{ \text{inl}(x_1).T_1 \text{inr}(x_2).T_2
\]

\[
\rho, \text{case } e \Downarrow \{ \text{inl}(x_1).e_1 : \text{inr}(x_2).e_2 \}
\]

\[
\rho, \text{case } T \Downarrow \{ \text{inl}(x_1).T_1 \text{inr}(x_2).T_2
\]

\[
\rho, \text{roll } e \Downarrow \text{roll } v, \text{roll } T
\]

\[
\rho, \text{unroll } e \Downarrow \text{unroll } v, \text{unroll } T
\]

\[
\rho, x \Downarrow [v_1, v_2], T
\]

\[
\rho, x \Downarrow [v_1, v_2], T
\]

\[
\rho, e \Downarrow [v_1, v_2], e
\]

\[
\rho, \text{case } e \Downarrow \{ \text{inl}(x_1).e_1 : \text{inr}(x_2).e_2 \}
\]

\[
\rho, \text{case } T \Downarrow \{ \text{inl}(x_1).T_1 \text{inr}(x_2).T_2
\]

\[
\rho, \text{roll } e \Downarrow \text{roll } v, \text{roll } T
\]

\[
\rho, \text{unroll } e \Downarrow \text{unroll } v, \text{unroll } T
\]

\[
\rho, x \Downarrow [v_1, v_2], T
\]

\[
\rho, x \Downarrow [v_1, v_2], T
\]

\[
\rho, e \Downarrow [v_1, v_2], e
\]

\[
\rho, \text{case } e \Downarrow \{ \text{inl}(x_1).e_1 : \text{inr}(x_2).e_2 \}
\]

\[
\rho, \text{case } T \Downarrow \{ \text{inl}(x_1).T_1 \text{inr}(x_2).T_2
\]

\[
\rho, \text{roll } e \Downarrow \text{roll } v, \text{roll } T
\]

\[
\rho, \text{unroll } e \Downarrow \text{unroll } v, \text{unroll } T
\]

\[
\rho, x \Downarrow [v_1, v_2], T
\]

\[
\rho, x \Downarrow [v_1, v_2], T
\]

\[
\rho, e \Downarrow [v_1, v_2], e
\]

\[
\rho, \text{case } e \Downarrow \{ \text{inl}(x_1).e_1 : \text{inr}(x_2).e_2 \}
\]

\[
\rho, \text{case } T \Downarrow \{ \text{inl}(x_1).T_1 \text{inr}(x_2).T_2
\]

\[
\rho, \text{roll } e \Downarrow \text{roll } v, \text{roll } T
\]

\[
\rho, \text{unroll } e \Downarrow \text{unroll } v, \text{unroll } T
\]
the form $\rho_1[x \mapsto v_1]$, where $v_1$ is a partial value which is then injected into the sum type and used to slice the scrutinee.

Unevaluation of the application of a primitive operation retrieves the values $v_1$ and $v_2$ previously cached in the trace and uses those to unevaluate the arguments. We treat all primitive operations as strict in both operands; it would be straightforward to extend the semantics and slicing rules to accommodate non-strict operations. There are also alternatives to the caching approach. One is to require that

- Monotonicity of explanation
- Computation of least slices

### 4.4 Correctness of tracing evaluation

Unevaluation is deterministic, which is again a straightforward induction, relying on the $v \not\sqsubseteq □$ side-conditions in Figure 11:

**Lemma 5** (Determinism of unevaluation). If $v, T \Downarrow 1 \rho, e$ and $v, T \Downarrow 1 \rho', e'$ then $(\rho, e) = (\rho', e')$.

Not every well-typed trace $T$ can be used to unevaluate a value $v$ of the same type. First, $T$ might have some strange (but well-typed) structure that could never be produced by evaluation, so that the required joins do not exist. Second, $T$ might have the right structure, but also some holes, and not enough trace is available to unevaluate $v$. So a key property of $T$ with respect to $v$ is whether it is able to guide the unevaluation of $v$. When $T$ has this property, we say that it $\text{explains}$ $v$. We can think of the unique $(\rho, e)$ such that $v, T \Downarrow 1 \rho, e$ as the “explanation” of $v$ which $T$ produces. Note that there is not a unique trace which explains $v$.  

**Definition 3** ($T$ explains $v$). For any value $v$ and trace $T$, we say that $T$ explains $v$ iff there exist $\rho, e$ such that $v, T \Downarrow 1 \rho, e$.

The key correctness property of tracing evaluation is as follows. If evaluating a program yields $v$ and $T$, then $T$ explains $v$. Before proving this, we first show that a trace $T$ of $v$ where $v, T \Downarrow 1 \rho, e$ gives rise to a monotonic function $\text{tr-uneval}_T$ from $\text{Prefix}(v)$ to $\text{Prefix}(\rho, e)$. In fact $\text{tr-uneval}_T$ also preserves meets and joins, but monotonicity is sufficient here.

**Definition 4** ($\text{tr-uneval}_T$, $\tau$). Suppose $T$ explains $v$. Then define $\text{tr-uneval}_T, \tau$ to be $\Downarrow 1^\tau$ domain-restricted to $\{u, T\} \mid u \sqsubseteq v$.

We omit the $v$ subscript when it is clear from the context that the argument to $\text{tr-uneval}_T, \tau$ is a prefix of $v$.

**Theorem 3** (Monotonicity of explanation).

*Suppose $T$ explains $v$. Then:

1. For any $u \sqsubseteq v$, $\text{tr-uneval}_T(u)$ is defined.
2. If $u \sqsubseteq u' \sqsubseteq v$ then $\text{tr-uneval}_T(u) \sqsubseteq \text{tr-uneval}_T(u')$.

Proof. See Appendix, Section A.1.

Monotonicity means that smaller values have smaller explanations. Now we establish that tracing evaluation to $v$ does indeed produce a trace able to explain $v$. Moreover, unevaluation after evaluation is deflationary: explanation of values are smaller than the programs which compute them. We state and prove these simultaneously. Again, we drop the $\rho, e$ subscript from $\text{tr-eval}_{\rho, e}$ when it is clear from the context that the argument is a prefix of $(\rho, e)$.

**Theorem 4** (Explanations are program prefixes).

*Suppose $\rho, e \Downarrow 1 v, T$. Then $T$ explains $v$. Moreover, for any $(\rho', e') \sqsubseteq (\rho, e)$:

\[
\text{tr-uneval}_T(\text{eval}(\rho', e')) \sqsubseteq (\rho', e')
\]

Proof. See Appendix, Section A.2.

### 4.5 Correctness of unevaluation

As we sketched in Section 3, the intuition is that a slice, or explanation, is “correct” if it can evaluate to at least the slicing criterion. We now show that we compute slices which have this property.

First we make the following observation. If we are able to unevaluate $v$ with $T$, then for any trace $U$ of a sub-computation of $T$ which was used to unevaluate an intermediate value $u \not\sqsubseteq □$, we must also have had $U \not\sqsubseteq □$, since otherwise unevaluation would have got stuck. But dually, we can also observe that if $U$ were used to unevaluate an intermediate value $u = □$, then $U$ was discarded in its entirety. In fact whenever $T$ suffices to unevaluate $v$, any larger trace is equally good.

**Lemma 6.** Suppose $S$ explains $v$. Then any $T \supseteq S$ explains $v$. Moreover, for any $u \sqsubseteq v$, we have $\text{tr-uneval}_S(u) = \text{tr-uneval}_T(u)$.

Proof. See Appendix, Section A.3.

It is also useful to have a lemma which composes some of our previous observations.

**Lemma 7.** Suppose $\rho, e \Downarrow 1 v, T$ and $(u, S) \sqsubseteq (v, T)$. If $S$ explains $u$ then $\text{tr-uneval}_T(u) \sqsubseteq (\rho, e)$.

Proof.

Suppose $\rho, e \Downarrow 1 v, T$ and $(u, S) \sqsubseteq (v, T)$ where $S$ explains $u$. Then:

- $\text{tr-uneval}_S(u) = \text{tr-uneval}_T(u)$ (Theorem 4)
- $\text{tr-uneval}_T(v) = \text{tr-uneval}_T(u)$ (Theorem 3)
- $(\rho, e)$

**Theorem 5** (Correctness of $\Downarrow 1$). Suppose $\rho, e \Downarrow 1 v, T$. If $(u, S) \sqsubseteq (v, T)$ and $S$ explains $u$ then $\text{eval}(\text{tr-uneval}_S(u)) \sqsubseteq u$.

Proof. See Appendix, Section A.4.

### 4.6 Computation of least slices

It is now easy to see that the unevaluation of $u$ is the smallest program slice large enough to evaluate to $u$. Moreover, any program slice as large as the unevaluation of $u$ is large enough to evaluate to $u$.

**Corollary 2** (Computation of least slices). Fix a terminating computation $\rho, e \Downarrow 1 v, T$. For any $u \sqsubseteq v$ and $(\rho', e') \sqsubseteq (\rho, e)$ we have:

- $u \sqsubseteq \text{eval}(\rho', e') \iff \text{tr-uneval}_T(u) \sqsubseteq (\rho', e')$

Proof. For the $\implies$ direction, suppose $\rho', e' \Downarrow 1 u'$, $S$ with $u' \sqsupseteq u$. Note that $S \supseteq T$ and $u' \sqsubseteq v$ by monotonicity.

\[
\text{tr-uneval}_T(u) = \text{tr-uneval}_T(u') (S \subseteq T, \text{Lemma 6})
\]

\[
\sqsubseteq \text{tr-uneval}_{T}(u') (\text{Theorem 3})
\]

\[
\sqsubseteq (\rho', e') (\text{Theorem 4})
\]

For the $\impliedby$ direction, suppose $\text{tr-uneval}_T(u) \sqsubseteq (\rho', e')$. Then:

\[
\text{eval}(\rho', e') \sqsubseteq \text{eval}(\text{tr-uneval}_T(u)) \text{ (Theorem 1)}
\]

\[
\sqsubseteq u (\text{Theorem 5})
\]
5.2 Computation of least trace slices

Theorem 7. But if $v, T \not\subseteq \rho, S$ then we can slice $v, T \not\subseteq \rho, e$. The only non-trivial case is the application rule, because we invoke the $\not\subseteq$ judgment from the $\not\subseteq$ judgment. Then we use that $\not\subseteq$ is deterministic (Lemma 5).

The key correctness property of trace slicing with $v$ is that it yields a partial trace able to explain $v$. Moreover, the resulting trace is smaller than the original trace:

**Theorem 7 (Correctness of trace slicing).** If $v, T \not\subseteq \rho, S$ then $S$ explains $v$ and $S \not\subseteq T$.

**Proof.** See Appendix, Section A.5.

The fact that slicing produces a smaller trace means that if we can slice $T$ with $v$, then it must be that $T$ explains $v$. By determinism the judgments agree on environments.

**Corollary 3.** $v, T \not\subseteq \rho, S \Rightarrow \exists e, v, T \not\subseteq \rho, e$.

**Proof.** Suppose $v, T \not\subseteq \rho, S$. Then $S$ explains $v$ and $S \not\subseteq T$ by Theorem 7. But if $S \not\subseteq T$, then $T$ also explains $v$ by Lemma 6. Then there exist $\rho', e$ such that $v, T \not\subseteq \rho, e$. But then $\rho = \rho'$ by Theorem 6 and the fact that $\not\subseteq$ is deterministic (Lemma 8).

5.2 Computation of least trace slices

When we slice $T$ with $v$ to obtain $S$, although $S$ may be strictly smaller than $T$, by Lemma 6 the program slice obtained by unevaluating $v$ with $S$ is the same as would be obtained by unevaluating with $T$. But $S$ is the canonical explanation of $v$ compatible with $T$, in that it is the least prefix of $T$ which still explains $v$.

**Theorem 8 (Trace slicing computes the least trace explaining $v$).** Suppose $v, T' \not\subseteq \rho, S$, and any $T' \not\subseteq T$ that explains $v$. Then $S \not\subseteq T'$.

**Proof.** See Appendix, Section A.6.

6. Implementation and Tracing Strategies

We have completed a prototype implementation, in Haskell, of our slicing techniques, as a tool that we call Slicer. As with most dynamic program slicers or debuggers, Slicer records a trace of the computation, consuming space (memory) linear in the number of execution steps of the program. Slicing and debugging information is often so critical that programmers routinely pay the space and time cost of recording the trace. We briefly outline two strategies for controlling tracing costs and present preliminary experimental results.

Our first strategy relies on Haskell’s lazy evaluation to construct the trace lazily, which is possible because the trace is not needed until it is sliced. Since slicing can throw away a portion of the trace, laziness may successfully avoid the redundant work of building environment, but it is identical to the one obtained by slicing, so we disregard it. (Theorem 6 below.)

5.1 Correctness of trace slicing

Trace slicing is deterministic. The proof is a straightforward induction, relying on the $v \not\subseteq$ side-conditions in Figure 12.

**Lemma 8.** Suppose $v, T \not\subseteq \rho, S$ and $v, T \not\subseteq \rho', S'$. Then $\rho = \rho'$ and $S = S'$.

If $T$ explains $v$ then we can slice $T$ with $v$. Moreover the partial environment we obtain is the one we would obtain via unevaluation:

**Theorem 6.** $v, T \not\subseteq \rho, e \implies \exists S, v, T \not\subseteq \rho, S$.

**Proof.** Straightforward induction on the derivation of $v, T \not\subseteq \rho, e$. The only non-trivial case is the application rule, because we invoke the $\not\subseteq$ judgment from the $\not\subseteq$ judgment. Then we use that $\not\subseteq$ is deterministic (Lemma 5).

The key correctness property of trace slicing with $v$ is that it yields a partial trace able to explain $v$. Moreover, the resulting trace is smaller than the original trace:

**Theorem 7 (Correctness of trace slicing).** If $v, T \not\subseteq \rho, S$ then $S$ explains $v$ and $S \not\subseteq T$.

**Proof.** See Appendix, Section A.5.

The fact that slicing produces a smaller trace means that if we can slice $T$ with $v$, then it must be that $T$ explains $v$. By determinism the judgments agree on environments.

**Corollary 3.** $v, T \not\subseteq \rho, S \Rightarrow \exists e, v, T \not\subseteq \rho, e$.

**Proof.** Suppose $v, T \not\subseteq \rho, S$. Then $S$ explains $v$ and $S \not\subseteq T$ by Theorem 7. But if $S \not\subseteq T$, then $T$ also explains $v$ by Lemma 6. Then there exist $\rho', e$ such that $v, T \not\subseteq \rho, e$. But then $\rho = \rho'$ by Theorem 6 and the fact that $\not\subseteq$ is deterministic (Lemma 8).

5.2 Computation of least trace slices

When we slice $T$ with $v$ to obtain $S$, although $S$ may be strictly smaller than $T$, by Lemma 6 the program slice obtained by unevaluating $v$ with $S$ is the same as would be obtained by unevaluating with $T$. But $S$ is the canonical explanation of $v$ compatible with $T$, in that it is the least prefix of $T$ which still explains $v$.

**Theorem 8 (Trace slicing computes the least trace explaining $v$).** Suppose $v, T' \not\subseteq \rho, S$, and any $T' \not\subseteq T$ that explains $v$. Then $S \not\subseteq T'$.

**Proof.** See Appendix, Section A.6.

6. Implementation and Tracing Strategies

We have completed a prototype implementation, in Haskell, of our slicing techniques, as a tool that we call Slicer. As with most dynamic program slicers or debuggers, Slicer records a trace of the computation, consuming space (memory) linear in the number of execution steps of the program. Slicing and debugging information is often so critical that programmers routinely pay the space and time cost of recording the trace. We briefly outline two strategies for controlling tracing costs and present preliminary experimental results.

Our first strategy relies on Haskell’s lazy evaluation to construct the trace lazily, which is possible because the trace is not needed until it is sliced. Since slicing can throw away a portion of the trace, laziness may successfully avoid the redundant work of building
the parts thrown away. Our second strategy, which we call delayed tracing, is a form of controlled laziness. It takes advantage of the close correspondence between traces and expressions in our design: a trace is essentially an unfolding of an expression (via evaluation) and its subtraces are recursively unfoldings of subexpressions. This makes it possible to reduce the size of a trace dramatically by substituting it with the expression that generated it. When the trace is needed during slicing, we rerun the expression to generate the full trace and slice it there and keep only the slice. Delaying thus pushes the cost of tracing from a run to the slicing itself and can thus be helpful in the cases where slices are small or computed interactively on demand.

We implemented and compared three strategies for tracing and slicing: eager, lazy, and delayed tracing.

- The eager strategy involves adding strictness annotations to the datatype for traces, along with seqs used for implementing our eager evaluation semantics in Haskell, so that the trace is completely constructed before we begin slicing it.
- The lazy strategy uses Haskell’s default lazy evaluation order for the traces. This still has a runtime cost, because thunks are constructed that capture intermediate values that may ultimately be needed to reconstruct parts of the trace.
- The delayed strategy uses a new trace form called a delay to record the current environment and expression instead of the full trace. We insert delays during evaluation at function calls when a given recursion depth is exceeded. When we encounter a delay trace during slicing, we run the expression with tracing enabled, which may lead to a trace with additional delays. In our prototype implementation, we make no attempt to avoid multiple evaluations of an expression. We use an initial depth bound of 10, which doubles during re-tracing so that we collect more detailed traces as we get closer to our goal.

Table 1 shows preliminary timing measurements for eager, lazy and delayed tracing. The programs involved are sort, which mergesorts a list, rbtree, which builds a red-black tree from a list, rbtree-len, which builds a pair of a red-black tree and the length of a list, and vec-sum, which does a vector addition of two lists.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Trace(s)</th>
<th>Slice(s)</th>
<th>Total(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eager</td>
<td>0.12</td>
<td>0.21</td>
<td>0.33</td>
</tr>
<tr>
<td>lazy</td>
<td>0.21</td>
<td>0.38</td>
<td>0.60</td>
</tr>
<tr>
<td>delay</td>
<td>0.38</td>
<td>0.70</td>
<td>1.08</td>
</tr>
</tbody>
</table>

Table 1: Comparison of eager, lazy and delayed tracing strategies. Times are in seconds. Eval is the time to evaluate without tracing, and Trace and Slice are the additional time needed for tracing and slicing respectively. Slice/Trace is the ratio of number of nodes in the trace slice to the full trace.

### Program slicing

Biswas’s [5] and Ochoa et al.’s [16] work are the closest to our program slicing techniques. One major difference between Biswas’ work and our techniques is that we support more flexible slicing criteria, allowing arbitrary portions of the output to be selected and a least slice for that partial value to be computed. Biswas considers only slicing with respect to the entire output. Ochoa et al. present techniques for computing slices under more flexible slicing criteria but they consider only first-order lazy programs. Our techniques appear to be the first where a flexible slicing criterion can be used to compute least slices in the (strict) higher-order setting.

In terms of technique, Biswas’ and Ochoa et al.’s approach both rely on labelling parts of the program and propagating the labels through execution to determine which parts of the program contribute to the output. Both can be viewed as constructing an execution trace: Biswas constructs an implicit trace by propagating labels through the execution and Ochoa et al.’s approach constructs an explicit trace in form of a redex trail [19] so that expressions that are evaluated by the lazy evaluation strategy can be identified. Our techniques also rely on construction of a trace, but we structure our traces to reflect closely the syntax and the semantics of expressions. This allows us to “unevaluate” trace slices back to expressions and to handle higher-order programs. It also enables exploration of the computation, debugging and slicing to be interleaved in an interactive way.

### Provenance

Provenance concerns the auditing and analysis of the origins and computational history of data. Provenance is a growing field with several applications in databases [7, 8, 11, 12], security [9, 20] and scientific workflow systems [6, 10, 18]. The techniques employed sometimes rely on traces but have to date mainly been developed for languages of limited expressiveness (e.g., monotone query languages) rather than general-purpose languages and often without proper formal foundations. Provenance extraction seems to be an important future application area for language-based tracing and slicing techniques. Some efforts in this direction include Hidders et al. [13], who model workflows using a core database query language extended with nondeterministic, external function calls, and partially formalize a semantics of runs which are used to label the operational derivation tree for the computation.
Recent work on security and provenance [1] is also based on big-step traces. The authors present a “disclosure slicing” algorithm similar to our trace slicing algorithm, which is aimed at ensuring that a trace retains enough information to show how an output was produced. However, that work does not investigate program slicing or unevaluation, and so does not slice into function bodies. We believe the connection between provenance security and slicing is an intriguing area for further work.

**Execution monitoring.** An alternative to tracing is the execution monitoring of Kishon and Hudak [15]. A generic instrumented interpreter provides observation events to a monitor, which can use this information to calculate various properties of the execution. This has the advantage of avoiding creating large intermediate data structures like traces or redex trails. A disadvantage is that it is not possible to manipulate or transform the view of execution after the fact, short of building an explicit trace as we do.

**Execution indexing.** Execution indexing is a technique, sometimes implemented using execution monitoring, for setting up a correspondence or alignment between the execution traces of two different runs of a program. Recent applications of execution indexing include a form of differential slicing [14]. Their differential slices differ from ours in supporting comparison of distinct runs, unlike our technique which only allows two slices of the same computation to be compared. However, these techniques have been developed only for imperative languages.

8. Conclusions
We often treat computation as a “black box”, but debugging, comprehension and analysis problems require breaking this abstraction and looking inside the box. Opening the box exposes a great deal of complexity and a number of questions. What abstractions do we need to understand computation? How can we explain computation in a way that can be understood by the user or programmer? In this paper, we present theoretical and practical techniques for understanding strict, higher-order functional computations. We present techniques for reasoning about partial values and partial expressions (programs), and define least program slices in an algorithm independent way. We then present the unevaluation algorithm for computing the least program slices from computations refuted as traces. This result solves the problem of slicing of higher-order, strict functional programs to produce minimal program slices defined as expressions with holes (missing parts).

We observe a limitation of program slices and develop a technique, trace (computation) slicing, to calculate partial computations, allowing deeper and more accurate examination of a computation. Our techniques enable the user to interact with a computation and build explanations of it in the programming language the computation is expressed in. We present a practical implementation in the form of a tool (Slicer), which helps testing our approach and present examples. We also briefly describe a technique for reducing the overhead of tracing by avoiding unnecessary computations. This technique takes advantage of the connection between traces and expressions to delay evaluation of expressions that may be “sliced away” in the future.

References
A. Proofs

We write IH for "induction hypothesis".

A.1 Proof of Theorem 3

Proof. Suppose \( v, T \vdash u, \rho, e \). We only show part (1); part (2) follows from part (1) by determinism (Lemma 5). So suppose \( u \vdash v \). We proceed by induction on the derivation. In each case, if \( u = \square \) the conclusion is immediate. We present only the variable, function and application cases. The other cases are similar, but simpler.

**Case**

\[
\rho, x \not\vdash v, T \therefore x \not\vdash \square
\]

We can immediately derive

\[
u, x \not\vdash v, T \therefore u \not\vdash \square\]

noting that \( \square \vdash v, T \therefore \square \).

**Case**

\[
\rho, x \not\vdash v, T \therefore (v', f \leftarrow u_1)[x \mapsto u_2] \vdash e' \]

\[
u, \rho, x \not\vdash v, T \therefore u_2 \not\vdash e' \]

We want to derive

\[
u, x \not\vdash v, T \therefore u_2 \not\vdash (v', f \leftarrow u_1)[x \mapsto u_2] \vdash e' \]

by Equation 1, with \( u_2 \not\vdash e' \) by the IH. Similarly, since \( u_1 \vdash v_1 \) and \( u_2 \not\vdash \square \), we have \( \rho, x \not\vdash v_1 \) and \( u_2 \not\vdash \square \). Note that \( u_1 \vdash v_1 \) and \( u_2 \not\vdash \square \) by Equation 1, and also \( e' \not\vdash e \).

**Second premise.** By the IH, \( T_2 \vdash u_2 \) and then:

\[
\rho, x \not\vdash v_1 \]

We want to derive

\[
u, x \not\vdash v_1 \]

by induction on the derivation of \( u_2 \), \( v_1 \not\vdash \square \) and \( e \). The proof is straightforward and similar to a proof of determinism, so we present only the hole, variable, function and application cases.

**Case**

\[
\rho, x \not\vdash v, T \therefore \square \not\vdash \square
\]

We can immediately derive

\[
u, x \not\vdash v, T \therefore \square \not\vdash \square\]

**Case**

\[
u, x \not\vdash v, T \therefore \rho, x \not\vdash v, T \therefore \rho, x \not\vdash \square
\]

We want to derive

\[
u, x \not\vdash v, T \therefore \rho, x \not\vdash \square \]

by induction on the derivation of \( u_2 \), \( \rho, x \not\vdash \square \) and \( e \). Then since \( S \vdash T \), we have \( \rho, x \not\vdash \square \) by the IH. Similarly, since \( S \vdash T \), we have \( \rho, x \not\vdash \square \) by the IH. Therefore, we have \( \rho, x \not\vdash \square \) by the IH.

**A.3 Proof of Lemma 6

Proof. Suppose \( S \vdash u \), \( u \vdash v \) and \( T \vdash S \). Note that if \( \Gamma \vdash S \) then \( \Gamma \vdash T \). By Theorem 3, \( S \vdash u \) by induction on the derivation of \( u, S \vdash \rho, e \). The proof is straightforward and similar to a proof of determinism, so we present only the hole, variable, function and application cases.

**Case**

\[
\square, S \not\vdash \square, T \therefore \square \not\vdash \square
\]

We can immediately derive

\[
\square, S \not\vdash \square, T \therefore \square \not\vdash \square\]

**Case**

\[
u, x \not\vdash v, T \therefore \rho, x \not\vdash \square
\]

We can immediately derive

\[
u, x \not\vdash v, T \therefore \rho, x \not\vdash \square\]

We want to derive

\[
u, x \not\vdash v, T \therefore \rho, x \not\vdash \square
\]

by induction on the derivation of \( u_2 \), \( \rho, x \not\vdash \square \) and \( e \). Then since \( S \vdash T \), we have \( \rho, x \not\vdash \square \) by the IH. Therefore, we have \( \rho, x \not\vdash \square \) by the IH. Therefore, we have \( \rho, x \not\vdash \square \) by the IH.
A.4 Proof of Theorem 5

Proof. Suppose \( \rho \models u \vdash v, T \) where \( \Gamma \vdash T \), and \((u, S) \subseteq (v, T)\) with \( u, S \models^{\mathit{ref}} \rho' \models e' \). We want that \( \rho', e' \models^{\mathit{ref}} u' \) with \( u' \not\models u \).

We proceed by induction on the derivation of \((u, S) \subseteq (v, T)\), presenting only the hole, variable, function and application cases. Recall Equation 1.

Case

\[ \square, S \models^{\mathit{ref}} \rho' \vdash e' \]

We can immediately derive \( \square, T \models^{\mathit{ref}} \rho, \square \).

Case

\[ u, x \models^{\mathit{ref}} \rho, x' \vdash u' \]

We can immediately derive

\[ \square, x' \models^{\mathit{ref}} \rho, x \]

noting that \((\square, x') \models^{\mathit{ref}} (\rho, x)\).

Case

\[ (\rho, \mathit{fun}(f)(x), e), \mathit{fun}(f)(x) \models^{\mathit{ref}} \rho, \mathit{fun}(f)(x) \]

We can immediately derive

\[ (\rho, \mathit{fun}(f)(x), e), \mathit{fun}(f)(x) \models^{\mathit{ref}} (\rho, \mathit{fun}(f)(x), e) \]

Case

\[ u, S \models^{\mathit{ref}} \rho'[f \mapsto u_1][x \mapsto v_1], e' \vdash u_2, S_2 \models^{\mathit{ref}} \rho_2, e_2' \]

We proceed by induction on the derivation of \( u, S \models^{\mathit{ref}} \rho' \models e' \) and inversion on the derivation of \( \rho', e' \models \rho, e \) with \( \rho, e \models^{\mathit{ref}} u \).

First we show that each premise of the unevaluation derivation yields a prefix of the evaluation of the corresponding derivation. Since \((u, S) \subseteq (v, T)\), we have \((\rho'[f \mapsto u_1][x \mapsto v_2], e') \models (\rho'[f \mapsto v_1][x \mapsto v_2], e)\) by Lemma 7. Then since \((u_2, S_2) \subseteq (v_2, T_2)\), we have \((\rho_2, e_2') \models (\rho, e_2)\) by Lemma 7. And then since \((u_1, T_1) \models (\rho, e_1)\) by Lemma 7.

We now want to derive

\[ \rho'[f \mapsto u_1][x \mapsto u_2], e' \models^{\mathit{ref}} u' \]

\[ \rho'[f \mapsto u_1][x \mapsto u_2], e' \models u' \]

with \( u' \not\models u \).

First premise. Noting that \( \rho_1 \subseteq \rho \) and \( \rho_2 \subseteq \rho \) and \( e_1' \subseteq e_1 \), we have

\[ u_1 \models (\rho'[f \mapsto v_1][x \mapsto v_2], e') \]

\[ \mathit{eval}(\rho_1 \cup \rho_2, e_1') \models u_1' \] (IH)

\[ \mathit{eval}(\rho_1 \cup \rho_2, e_1') \models u_1' \] (Theorem 1)

\[ (\rho'[f \mapsto v_1][x \mapsto v_2], e') \models u_1' \] (Theorem 1)

Second premise. Similarly, noting that \( e_2' \subseteq e_2 \) we have

\[ u_2 \models \mathit{eval}(\rho_2, e_2') \] (IH)

\[ \mathit{eval}(\rho_1 \cup \rho_2, e_1') \models u_2' \] (Theorem 1)

\[ \mathit{eval}(\rho_2, e_2') \models u_2' \] (Theorem 1)

Third premise. Using the inequalities established for the first and second premises, we note that \( \rho'[f \mapsto u_1][x \mapsto u_2] \subseteq \rho'[f \mapsto u_1][x \mapsto v_2] \subseteq \rho'[f \mapsto v_1][x \mapsto v_2] \). Then we have:

\[ \mathit{eval}(\rho'[f \mapsto u_1][x \mapsto u_2], e') \] (IH)

\[ \mathit{eval}(\rho'[f \mapsto u_1][x \mapsto v_2], e') \] (Theorem 1)

We can immediately derive

\[ \square, \Gamma \models^{\mathit{ref}} \rho, \square \]

Since \( S = x = T \) we can immediately derive

\[ v, x \models^{\mathit{ref}} (\square, \Gamma) \]

The case for a primitive constant \( c \) is similar to the variable case.

A.5 Proof of Theorem 7

Proof. Suppose \( v, T \not\models \rho, S \). We want that there exist \( \rho', e \) such that \( v, S \models^{\mathit{ref}} \rho', e \), and also that \( S \subseteq T \). We proceed by induction on the derivation.

Case

\[ v, x \models^{\mathit{ref}} (\square, \Gamma) \]

We can immediately derive

\[ \square, \Gamma \models^{\mathit{ref}} \rho, \square \]

Case

\[ v, x \not\models^{\mathit{ref}} (\square, \Gamma) \]

Since \( S = x = T \) we can immediately derive

\[ v, x \models^{\mathit{ref}} (\square, \Gamma) \]

Both premises are immediate from the IH, with \( S_1 \subseteq T_1 \) and \( S_2 \subseteq T_2 \). The cases for pairs \((T_1, T_2)\), projections \( \mathit{fst} \) and \( \mathit{snd} \), injections \( \mathit{inl} \) and \( \mathit{inr} \), and \( \mathit{roll} \) and \( \mathit{unroll} \) are similar.

Case

\[ (\rho, \mathit{fun}(f)(x), e), \mathit{fun}(f)(x) \models^{\mathit{ref}} \rho, \mathit{fun}(f)(x) \]

We can immediately derive

\[ (\rho, \mathit{fun}(f)(x), e), \mathit{fun}(f)(x) \models^{\mathit{ref}} \rho, \mathit{fun}(f)(x) \]

Case

\[ v, T \not\models \rho[f \mapsto v_1][x \mapsto v_2], S \]

We want to derive:

\[ c_2, S_2 \not\models \rho_2, e_2 \]

\[ v, S_2 \not\models \rho_2 \]

\[ v, S_2 \not\models \rho_2 \]

Both premises are immediate from the IH, with \( S_1 \subseteq T_1 \) and \( S_2 \subseteq T_2 \). The cases for pairs \((T_1, T_2)\), projections \( \mathit{fst} \) and \( \mathit{snd} \), injections \( \mathit{inl} \) and \( \mathit{inr} \), and \( \mathit{roll} \) and \( \mathit{unroll} \) are similar.

First premise. By the IH, \( S \) explains \( v \) and \( S \not\models T \). By Lemma 6, \( T \) also explains \( v \), with \( \mathit{tr-uneval}(v) = \mathit{tr-uneval}_S(v) \). Note that then \( e = e' \). Moreover, \( \rho'[f \mapsto v_1][x \mapsto v_2] = \rho'[f \mapsto v_1][x \mapsto v_2] \) by Theorem 6 and the deterministic of \( \mathit{v} \). Then \( \rho' = \rho \) and \( v_1' = v_1 \) and \( v_2' = v_2 \) by Equation 1.

Second premise. \( S \) explains \( v_2 = v_2' \), with \( S_2 \subseteq T_2 \) immediately by the IH.
Third premise. S1 explains \( v_1 \updownarrow (\rho, \text{fun } f(x).e) = v_1' \updownarrow (\rho', \text{fun } f(x).e') \) immediately by the IH.

**Case**

\[
\begin{align*}
\text{case } & \frac{\frac{v, T_1 \vdash \rho_1[x \mapsto v_1], S_1 \in \text{inl } v_1, T \vdash \rho, S}{v, \text{case } T \in \{\text{inl } (x_1), T_1; \text{inr } (x_2), e_2\} \vdash v \neq \Box}}{v_1' \updownarrow (\rho', \text{fun } f(x).e), S_1 \vdash v, T \vdash \rho_1, e_1} \\
\text{we want to show that case } & \frac{S \in \{\text{inl } (x_1), T_1; \text{inr } (x_2), e_2\} \vdash v_1' \updownarrow \rho_1, e_1 \neq \Box}{v_1' \updownarrow \rho_1'} \quad \text{and } v_1' \updownarrow \rho_1 \neq \Box} \\
\text{case } & \frac{S \in \{\text{inl } (x_1), T_1; \text{inr } (x_2), e_2\} \vdash v \neq \Box}{v, \text{case } S \in \{\text{inl } (x_1), T_1; \text{inr } (x_2), e_2\} \vdash v_1' \updownarrow \rho_1', e_1 \neq \Box} \\
\text{case } & \frac{S \in \{\text{inl } (x_1), T_1; \text{inr } (x_2), e_2\} \vdash \rho_1' \neq \Box}{v, \text{case } S \in \{\text{inl } (x_1), T_1; \text{inr } (x_2), e_2\} \vdash v \neq \Box} \\
\text{case } & \frac{S \in \{\text{inl } (x_1), T_1; \text{inr } (x_2), e_2\} \vdash \rho_1 \neq \Box}{v, \text{case } S \in \{\text{inl } (x_1), T_1; \text{inr } (x_2), e_2\} \vdash v \neq \Box}
\end{align*}
\]

The conclusion is immediate.

**A.6 Proof of Theorem 8**

Proof. Suppose \( v, T' \vdash \rho, S, \) and any \( T \subseteq T' \) such that \( v, T \vdash \rho_1, e \). We proceed by induction on the derivation of \( v, T \vdash \rho_1, e \) and inversion on the derivation of \( v, T' \vdash \rho_1, S \), using a stronger inductive hypothesis, viz. that \( (\rho', S) \subseteq (\rho, T) \). Note that if \( \Gamma \vdash T' \) then \( \Gamma \vdash T \).

**Case**

\[
\begin{align*}
\frac{\Box, T \vdash \rho_1, \Box}{\Box, T' \vdash \rho_1, \Box} \\
\frac{\Box, T \vdash \rho_1, \Box}{\Box, T' \vdash \rho_1, \Box}
\end{align*}
\]

The conclusion is immediate.

**Case**

\[
\begin{align*}
\frac{v, T \vdash \rho_1, \Box}{v, x \vdash \rho_1, x} & \quad v \neq \Box \\
\frac{v, T \vdash \rho_1, \Box}{v, x \vdash \rho_1, x} & \quad v \neq \Box
\end{align*}
\]

Again the conclusion is immediate.

**Case**

\[
\begin{align*}
\frac{c, e \vdash \rho_1, c}{c, e \vdash \rho_1, c} \\
\frac{c, e \vdash \rho_1, c}{c, e \vdash \rho_1, c}
\end{align*}
\]

Again the conclusion is immediate.

**Case**

\[
\begin{align*}
\frac{c_2, T_2 \vdash \rho_2, e_2 \quad c_1, T_1 \vdash \rho_1, e_1}{v, T_1 \uparrow c_1, c_2, T_2 \vdash \rho_1, e_1} & \quad v \neq \Box \\
\frac{c_2, T_2 \vdash \rho_2, e_2 \quad c_1, T_1 \vdash \rho_1, e_1}{v, T_1 \uparrow c_1, c_2, T_2 \vdash \rho_1, e_1} & \quad v \neq \Box
\end{align*}
\]

Since \( T_1 \subseteq T_1' \), we have \( \{\rho_1', T_1'\} \subseteq (\rho_1, T_1) \) by the IH. Similarly, \( T_2 \subseteq T_2' \) and so \( \{\rho_2', T_2'\} \subseteq (\rho_2, T_2) \) by the IH. Then \( \{\rho_1', \rho_2', T_1' \uparrow c_1, c_2, T_2' \} \subseteq \{\rho_1, \rho_2, T_1 \uparrow c_1, c_2, T_2\} \). The cases for pairs \( (T_1, T_2) \), projections \( \text{fst } \) and \( \text{snd } \), injections \( \text{inl } \) and \( \text{inr } \), and \( \text{roll } \) and \( \text{unroll } \) are similar.

**Case**

\[
\frac{\frac{\{\rho, \text{fun } f(x).e, \text{fun } f(x).e'\} \vdash \rho_1, \text{fun } f(x).e}{\{\rho, \text{fun } f(x).e, \text{fun } f(x).e'\} \vdash \rho_1, \text{fun } f(x).e}}
\]

The conclusion is immediate.