Functional Programs that Explain their Work

Roly Perera
University of Birmingham & MPI-SWS
rolyp@mpi-sws.org

Umut A. Acar
Carnegie Mellon University & MPI-SWS
umut@cs.cmu.edu

James Cheney
University of Edinburgh
jcheney@inf.ed.ac.uk

Paul Blain Levy
University of Birmingham
P.B.Levy@cs.bham.ac.uk

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: trace slices. We present techniques for specifying flexible and rich slicing criteria based on partial expressions, parts of which have been replaced by holes. We characterise program slices in an algorithm-independent fashion and show that a least slice for a given criterion exists. We then present an algorithm, called *unevaluation*, for computing least program slices from computations reified as traces. Observing a limitation of program slices, we develop a notion of *trace slice* as another form of explanation and present an algorithm for computing them. The unevaluation algorithm can be applied to any subtrace of a trace slice to compute a program slice whose evaluation generates that subtrace. This close correspondence between programs, traces, and their slices can enable the programmer to understand a computation interactively, in terms of the programming language in which the computation is expressed. We present an implementation in the form of a tool, discuss some important practical implementation concerns and present some techniques for addressing them.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and debugging—Tracing; D.3.4 [*Programming Languages*]: Processors—Debuggers

Keywords program slicing; debugging; provenance

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 a 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 [27], 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 were later extended to compute forward slices. Since it pertains to a specific execution, and explains the relationship between a computation and its result, dynamic 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 [4], software maintenance [5], and testing [3]. For a comprehensive overview of the slicing literature, we refer the reader to existing surveys [26, 28].

While slicing has been studied extensively in the context of imperative languages, there is comparatively less 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; it is not clear whether slicing techniques for imperative programs can be extended to handle higher-order values. Second, functional programs typically manipulate complex values such as recursive data types, whereas slicing techniques in imperative programs often perform slicing at the granularity of variables, which is too coarse a grain to be effective for functional programs.

Reps and Turnidge [21] 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 [6] 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 [19] present techniques that allow more flexible criteria but which only apply to first-order, lazy functional languages. To the best of our knowledge, the problem of slicing of strict, higher-order functional programs with rich criteria has remained an open problem.

In this paper, we develop techniques for computing program slices for strict, higher-order functional programs based on rich slicing criteria and propose techniques for more powerful examinations of computations via trace slices.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'12, September 9–15, 2012, Copenhagen, Denmark.
Copyright © 2012 ACM 978-1-4503-1054-3/12/09. . . $10.00

To enable expressing rich queries on the computation, we consider two forms of slicing criteria: partial values and differential partial values. We define a *partial value* as a value where some subvalues are replaced by a *hole*, written □. Intuitively, a partial value discards parts of a value that are of no interest to the user. For example, the partial value `(□,2)` specifies a pair where the first component is of no interest; the partial value `cons(□,cons(2,□))` specifies a list where the first element and the second tail are of no interest. While partial values help throw away parts of a value that are of no interest, they may not always be used to focus on a specific part of a value, because the "path" to the part of interest cannot be replaced by a hole, e.g., we cannot focus on the second part of a tuple without specifying there being a tuple. To make it possible to focus on parts of a value, we define *differential* partial values, which can highlight subvalues of interest. For example the differential partial value `cons(□,cons(2,□))` specifies a list where the second element is of interest.

For slicing criteria consisting of (differential) partial values, we define *program slices* in an algorithm-independent way and show that least program slices exists (Section 3). We define a *partial program* as a program where some subexpressions are replaced by holes and give an operational semantics for partial programs (expressions). We call a partial program a *program slice* for a partial value if executing the partial program yields that partial value as an output. We then exhibit a family of Galois connections between partial programs and partial values which guarantees the existence of a least program slice for a given partial value.

Our characterization of program slices shows that it is possible to compute least program slices but does not provide an algorithm that does so. To efficiently compute least program slices, we develop an *unevaluation* algorithm (Section 4). Unevaluation relies on traces that represent an evaluation by recording the evaluation derivation as an unrolled expression. Our use of traces are inspired by self-adjusting computation [1], but we structure them differently to enable their use in computation comprehension. Given a slicing criterion, our unevaluation algorithm uses the trace to "push" the criterion, a partial value, backward through the trace, discarding unused parts of the trace and rolling the remaining parts up into a least program slice for the criterion.

Our program slices can give valuable explanations about a computation, but as with any slicing technique, they can lose their effectiveness in complex computations. Intuitively, the reason for this is that each expression in a program slice must subsume the many different behaviors that the expression exhibits in an execution. Indeed, 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. For example, we show how program slices become ineffective when trying to understand a buggy mergesort implementation (Section 2.3).

Motivated by the limitations of program slices, we introduce the concept of *trace slicing* (Section 5). As the name suggest, the high-level idea is to compute a slice of a trace for a given slicing criterion by replacing parts of the trace that are not relevant to the criterion with holes. We call the resulting reduced trace a *partial trace*; a partial trace *explains* a partial value (slicing criterion) if the trace contains enough information to unevaluate the partial value. We then present a trace-slicing algorithm for computing the least partial trace which explains a given partial value. When computing a least trace slice, it is critical to compute least slices of any higher-order values; our trace-slicing algorithm defers to unevaluation to do this.

When used in combination, our techniques can offer an interactive and precise approach to understanding computations. The user can execute a program with some input and then compute a (least) program slice for a desired slicing criterion. If interested in understanding the computation in more depth, the user can then ask for 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. Explanations are expressed in the same programming language as the computation itself.

We present a tool, Slicer, that enables interactive trace 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.

Our contributions include the following:

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

Proofs that are omitted due to space restrictions can be found in the companion technical report [20].

2. Overview

We present an overview of our techniques considering examples with increasing sophistication. All the figures presented here were produced by our tool, 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 traces and hides the deeper levels under ellipses written as "...". The user can click on ellipses to see the hidden contents. This invites the user to think of the execution of an expression as an "unrolling" of that expression. Slicer can also display the (partial) value associated with a step in the computation, which is a feature that we often utilize; such partial values are printed in shaded (gray) boxes.

Slicer prints holes, written as □ throughout the formalism, as ⊞, because the user can ask for what is hidden behind the hole to be displayed.

2.1 Example: List Length

Consider using the standard `length` function to compute the length of a list with three elements:

```
let fun length xs =
      case xs of
        Nil -> 0
        Cons(x,xs) -> 1 + length xs'
in length (Cons(1,Cons(2,Cons(3,Nil))))
```

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 `length` that its output does not depend on the elements of the input

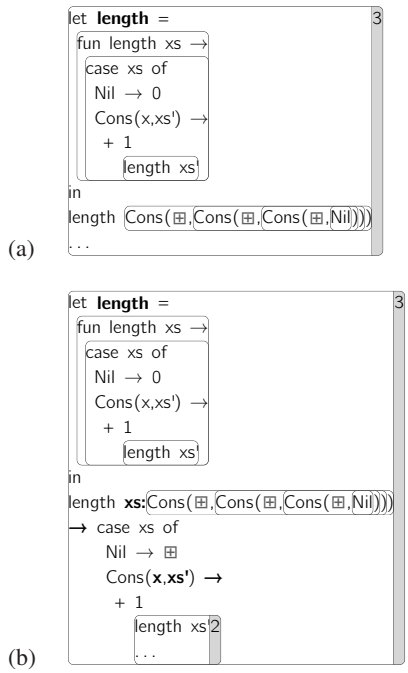

Figure 1. (a) slice of `length`; (b) expanded by one step

list. The slice also illustrates that all of the function `length` is relevant to the output, i.e., all branches were exercised to compute the result. To see the trace slice for the actual computation, we unroll the ellipses at the end of the program slice. The trace slice, shown in Figure 1(b) following the expression slice of `length`, illustrates how the result is actually computed, throwing away subtraces not contributing to the result. The trace slice is an unrolling of the `case` expression constituting the body of `length`, leading to a recursive call which returns 2.

2.2 Example: List Map

Consider applying the standard list primitive `map` to increment each element of the list `Cons(6,Cons(7,Cons(2,Nil)))` by one:

```
let fun map f xs =
      case xs of
        Nil -> Nil
        Cons y -> Cons (f (fst y), map f (snd y))
in map (fn x => x + 1) (Cons(6,Cons(7,Cons(2,Nil))))
```

After performing this computation in Slicer, we can ask for a trace slice with the partial value `Cons(⊞,Cons(8,⊞))`. Figure 2(a) shows the trace slice computed by Slicer. (Please ignore for now the shaded green boxes.) The trace slice starts with a slice of the definition of `map`, where the `Nil` branch is a hole. This indicates that the `Nil` branch was unused – indeed, the partial value does not end with `Nil` but with a hole. The trace slice continues with the application of `map` to the increment function and the input list, where parts of the input are replaced with holes. What is interesting here is that the way the list argument to map has been sliced shows that both the first `Cons` cell, without the head, and the second `Cons` cell, without the tail, contribute to the partial result. Indeed, making changes to any of these elements could change the parts of the result relevant to the criterion.

The partial trace for the partial value `Cons(⊞,Cons(8,⊞))` 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 `Cons(⊞,Cons(8,⊞))`. Slicer returns a trace slice where parts contributing directly to the

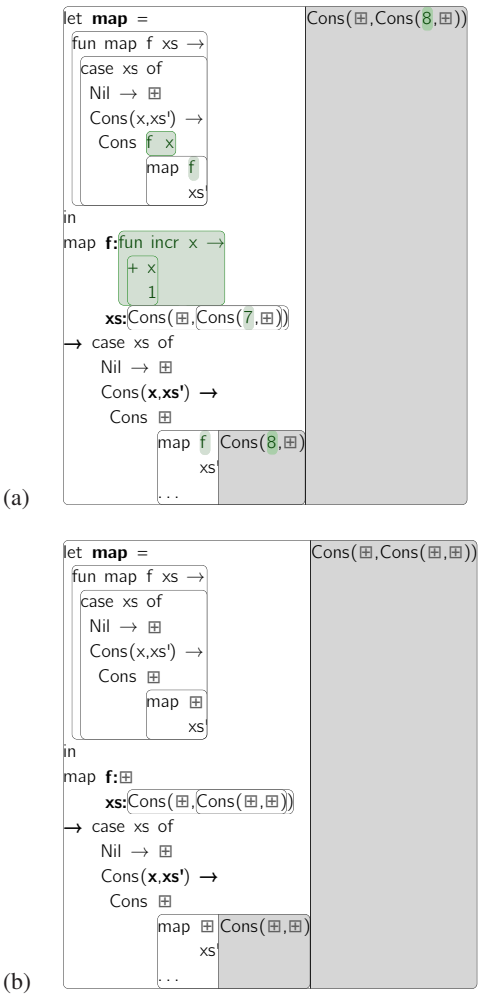

Figure 2. Slice of `map incr [6,7,2]`

second element are also highlighted in green, as shown in Figure 2(a). The highlighted parts show that the second element is computed by applying 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 sense that v replaces with actual values some holes of u. As we show in Section 5, the trace slice for v is greater 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 `Cons(⊞,Cons(8,⊞))`, we compute the delta between the trace slice for `Cons(⊞,Cons(8,⊞))` and then trace slice for `Cons(⊞,Cons(⊞,⊞))`. It is this difference that is highlighted in Figure 2(a). For the purposes of comparison, we show the trace slice for `Cons(⊞,Cons(⊞))` 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

Figure 3. Outermost merge phase of mergesort.

recursively, and merges the results to produce the sorted output list. The bug is in the merge function: the "else" branch should be `Cons(y, ...)` instead of `Cons(x, ...)`. Here is the buggy code for `merge`:

```
fun merge xs ys =
  case xs of
    Nil -> ys
    Cons (x,xs') ->
      case ys of
        Nil -> xs
        Cons (y,ys') -> case x < y of
                          True -> Cons(x,merge(xs',ys))
                          False -> Cons(x,merge(xs,ys'))
```

When used to sort the list `Cons(1,Cons(2,Cons(3,Nil)))` the buggy merge sort returns the list `Cons(1,Cons(3,Cons(3,Nil)))`. To identify the bug, we slice the trace with respect to the differential value `Cons(1,Cons(3,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 though they are least slices for the supplied criterion. We therefore inspect the actual trace slice. Figure 3 illustrates the outermost level of the trace slice including the outermost level of the merge function. As the figure illustrates, Slicer highlights precisely the offending statement of the `merge` function in the trace showing where the 3 in the output comes from `Cons` statements in the `False` branch of the `merge` function, the buggy expression.

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.

Types	τ	$::= \mathbf{1} \mid b \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \mu\alpha.\tau \mid \alpha$
Contexts	Γ	$::= \bullet \mid \Gamma, x : \tau$
Expressions	e	$::= x \mid () \mid c \mid e_1 \oplus e_2 \mid \texttt{fun}\ f(x).e \mid e_1\ e_2$
		$\mid (e_1, e_2) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid \texttt{inl}\ e \mid \texttt{inr}\ e$
		$\mid \texttt{case}\ e\ \texttt{of}\ \{\texttt{inl}(x_1).e_1; \texttt{inr}(x_2).e_2\} \mid$
		$\mid \texttt{roll}\ e \mid \texttt{unroll}\ e$
Values	v	$::= c \mid (v_1, v_2) \mid \texttt{inl}\ v \mid \texttt{inr}\ v \mid \texttt{roll}\ v$
		$\mid \langle \rho, \texttt{fun}\ f(x).e \rangle$
Environments	ρ	$::= \bullet \mid \rho[x \mapsto v]$

Figure 4. Reference language: abstract syntax

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 $\mu\alpha.\tau$, type variables α, and primitive types b. Variable contexts are defined inductively in the usual way. Expressions include the unit value $()$, standard introduction and elimination forms for 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 ρ is a well-formed environment for Γ. The signature Σ assigns to every primitive constant c 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 $\rho, e \Downarrow_{\mathsf{ref}} v$ states that expression e evaluates in closing environment ρ to value v. Values include the usual forms, plus closures $\langle \rho, \texttt{fun}\ f(x).e \rangle$. 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.

Lemma 1 (Type preservation for $\Downarrow_{\mathsf{ref}}$)**.** *If $\Gamma \vdash e : \tau$ and $\Gamma \vdash \rho$ and $\rho, e \Downarrow_{\mathsf{ref}} v$ then $\vdash v : \tau$.*

Lemma 2 (Determinism of $\Downarrow_{\mathsf{ref}}$)**.** *If $\rho, e \Downarrow_{\mathsf{ref}} v$ and $\rho, e \Downarrow_{\mathsf{ref}} v'$ then $v = v'$.*

The language used in our examples and implementation can be easily desugared into the reference language just described.

3.2 Characterizing program slices

In an imperative language, a program can often be sliced by simply deleting some of its statements. This approach is unsuitable for functional languages, which tend to be expression-based. We therefore introduce a new expression constructor $\Box$ called *hole*, which inhabits every type, and use holes to represent deleted sub-terms of an expression or value. For example we can "slice" the expression $\texttt{inl}\ (3 \oplus 4)$ by replacing its left sub-expression by $\Box$, obtaining $\texttt{inl}\ (\Box \oplus 4)$. The additional syntax rules are given at the top of Figure 7.

Introducing $\Box$ into the syntax gives rise to a partial order $\sqsubseteq$ on expressions. This follows immediately from the standard construction of expressions as sets of odd-length paths, where a *path* is an alternating sequence $\langle k_0, n_0, \ldots, k_{i-1}, n_{i-1}, k_i \rangle$ of constructors k and child indices n. In order to represent an expression, a set of such paths must satisfy two properties characteristic of "tree-

$\boxed{\rho, e \Downarrow_{\mathsf{ref}} v}$

$$\frac{}{\rho, x \Downarrow_{\mathsf{ref}} \rho(x)} \qquad \frac{}{\rho, c \Downarrow_{\mathsf{ref}} c} \qquad \frac{\rho, e_1 \Downarrow_{\mathsf{ref}} c_1 \qquad \rho, e_2 \Downarrow_{\mathsf{ref}} c_2}{\rho, e_1 \oplus e_2 \Downarrow_{\mathsf{ref}} c_1 \mathbin{\hat{\oplus}} c_2}$$

$$\frac{}{\rho, \mathtt{fun}\ f(x).e \Downarrow_{\mathsf{ref}} \langle \rho, \mathtt{fun}\ f(x).e \rangle}$$

$$\frac{\rho, e_1 \Downarrow_{\mathsf{ref}} v_1 \qquad \rho, e_2 \Downarrow_{\mathsf{ref}} v_2 \qquad \rho'[f \mapsto v_1][x \mapsto v_2], e \Downarrow_{\mathsf{ref}} v}{\rho, e_1\ e_2 \Downarrow_{\mathsf{ref}} v}\ v_1 = \langle \rho', \mathtt{fun}\ f(x).e \rangle$$

$$\frac{\rho, e_1 \Downarrow_{\mathsf{ref}} v_1 \qquad \rho, e_2 \Downarrow_{\mathsf{ref}} v_2}{\rho, (e_1, e_2) \Downarrow_{\mathsf{ref}} (v_1, v_2)} \qquad \frac{\rho, e \Downarrow_{\mathsf{ref}} (v_1, v_2)}{\rho, \mathtt{fst}\ e \Downarrow_{\mathsf{ref}} v_1}$$

$$\frac{\rho, e \Downarrow_{\mathsf{ref}} (v_1, v_2)}{\rho, \mathtt{snd}\ e \Downarrow_{\mathsf{ref}} v_2} \qquad \frac{\rho, e \Downarrow_{\mathsf{ref}} v}{\rho, \mathtt{inl}\ e \Downarrow_{\mathsf{ref}} \mathtt{inl}\ v} \qquad \frac{\rho, e \Downarrow_{\mathsf{ref}} v}{\rho, \mathtt{inr}\ e \Downarrow_{\mathsf{ref}} \mathtt{inr}\ v}$$

$$\frac{\rho, e \Downarrow_{\mathsf{ref}} \mathtt{inl}\ v_1 \qquad \rho[x_1 \mapsto v_1], e_1 \Downarrow_{\mathsf{ref}} v}{\rho, \mathtt{case}\ e\ \mathtt{of}\ \{\mathtt{inl}(x_1).e_1; \mathtt{inr}(x_2).e_2\} \Downarrow_{\mathsf{ref}} v}$$

$$\frac{\rho, e \Downarrow_{\mathsf{ref}} \mathtt{inr}\ v_2 \qquad \rho[x_2 \mapsto v_2], e_2 \Downarrow_{\mathsf{ref}} v}{\rho, \mathtt{case}\ e\ \mathtt{of}\ \{\mathtt{inl}(x_1).e_1; \mathtt{inr}(x_2).e_2\} \Downarrow_{\mathsf{ref}} v}$$

$$\frac{\rho, e \Downarrow_{\mathsf{ref}} v}{\rho, \mathtt{roll}\ e \Downarrow_{\mathsf{ref}} \mathtt{roll}\ v} \qquad \frac{\rho, e \Downarrow_{\mathsf{ref}} v}{\rho, \mathtt{unroll}\ e \Downarrow_{\mathsf{ref}} v}$$

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

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash () : \mathtt{1}} \qquad \frac{}{\Gamma \vdash x : \tau}\ x : \tau \in \Gamma \qquad \frac{}{\Gamma \vdash c : b}\ c : b \in \Sigma$$

$$\frac{\Gamma \vdash e_1 : b_1 \qquad \Gamma \vdash e_2 : b_2}{\Gamma \vdash e_1 \oplus e_2 : \tau}\ \oplus : b_1 \times b_2 \to \tau \in \Sigma$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{fun}\ f(x).e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{fst}\ e : \tau_1}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{snd}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathtt{inl}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathtt{inr}\ e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case}\ e\ \mathtt{of}\ \{\mathtt{inl}(x_1).e_1; \mathtt{inr}(x_2).e_2\} : \tau}$$

$$\frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \mathtt{unroll}\ e : \tau[\mu\alpha.\tau/\alpha]} \qquad \frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \mathtt{roll}\ e : \mu\alpha.\tau}$$

$\boxed{\Gamma \vdash \rho}$

$$\frac{}{\bullet \vdash \bullet} \qquad \frac{\Gamma \vdash \rho \qquad \vdash v : \tau}{\Gamma, x : \tau \vdash \rho[x \mapsto v]}$$

$\boxed{\vdash v : \tau}$

$$\frac{}{\vdash () : \mathtt{1}} \qquad \frac{}{\vdash c : b}\ c : b \in \Sigma$$

$$\frac{\Gamma \vdash \rho \qquad \Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2}{\vdash \langle \rho, \mathtt{fun}\ f(x).e \rangle : \tau_1 \to \tau_2} \qquad \frac{\vdash v_1 : \tau_1 \qquad \vdash v_2 : \tau_2}{\vdash (v_1, v_2) : \tau_1 \times \tau_2}$$

$$\frac{\vdash v : \tau_1}{\vdash \mathtt{inl}\ v : \tau_1 + \tau_2} \qquad \frac{\vdash v : \tau_2}{\vdash \mathtt{inr}\ v : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash v : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \mathtt{roll}\ v : \mu\alpha.\tau}$$

Figure 6. Reference language: typing judgments

hood": prefix-closure, and deterministic extension. Prefix-closure means that if a path is in the set, then each of its prefixes is in the set; deterministic extension means that all paths agree about the value of k_i for a given position in the tree.

The partial order $\sqsubseteq$ is simply the inclusion order on these sets. For example, that $\mathtt{inl}\ (3 \oplus \Box)$ and $\mathtt{inl}\ (3 \oplus 4)$ are related by $\sqsubseteq$ comes about because the sets of paths that comprise the two expressions are similarly related:

$$\overbrace{\underbrace{\langle \mathtt{inl} \rangle, \langle \mathtt{inl}, \mathbf{0}, \oplus \rangle, \langle \mathtt{inl}, \mathbf{0}, \oplus, \mathbf{0}, 3 \rangle}_{\mathtt{inl}\ (3 \oplus \Box)}}^{} \sqsubseteq \overbrace{\underbrace{\langle \mathtt{inl} \rangle, \langle \mathtt{inl}, \mathbf{0}, \oplus \rangle, \langle \mathtt{inl}, \mathbf{0}, \oplus, \mathbf{0}, 3 \rangle, \langle \mathtt{inl}, \mathbf{0}, \oplus, \mathbf{1}, 4 \rangle}_{\mathtt{inl}\ (3 \oplus 4)}}^{}$$

(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 $\Box$. 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 $\mathsf{Prefix}(e)$.

Definition 1 (Prefix of e)**.** $\mathsf{Prefix}(e) = \{e' \mid e' \sqsubseteq e\}$

What is more, the set $\mathsf{Prefix}(e)$ 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 $\mathtt{inl}\ 3 \oplus 4$ and $\mathtt{inl}\ 3 \oplus 5$, 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:

$$\overbrace{\underbrace{\langle \mathtt{inl} \rangle, \langle \mathtt{inl}, \mathbf{0}, \oplus \rangle, \langle \mathtt{inl}, \mathbf{0}, \oplus, \mathbf{0}, 3 \rangle}_{\mathtt{inl}\ (3 \oplus \Box)}}^{} \sqcup \overbrace{\underbrace{\langle \mathtt{inl} \rangle, \langle \mathtt{inl}, \mathbf{0}, \oplus \rangle, \langle \mathtt{inl}, \mathbf{0}, \oplus, \mathbf{1}, 4 \rangle}_{\mathtt{inl}\ (\Box \oplus 4)}}^{} = \overbrace{\underbrace{\langle \mathtt{inl} \rangle, \langle \mathtt{inl}, \mathbf{0}, \oplus \rangle, \langle \mathtt{inl}, \mathbf{0}, \oplus, \mathbf{0}, 3 \rangle, \langle \mathtt{inl}, \mathbf{0}, \oplus, \mathbf{1}, 4 \rangle}_{\mathtt{inl}\ (3 \oplus 4)}}^{}$$

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

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, $\Downarrow_{\mathsf{ref}}$ is free to consume a hole, but only in order to produce a hole. We capture this informal notion by extending the $\Downarrow_{\mathsf{ref}}$ rules with the additional rules for propagating holes given in Figure 7. Hole itself evaluates to $\Box$, 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 $\Downarrow_{\mathsf{ref}}$ we shall mean the extended version of the rules.

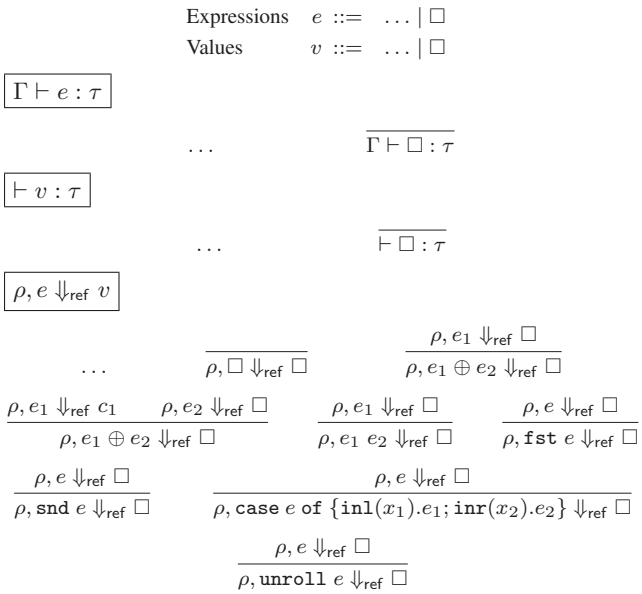

Figure 7. Additional rules for partial expressions

Since evaluation can produce partial values, clearly environments may now map variables to partial values. This gives rise to a partial order on environments. Specifically, we overload $\sqsubseteq$ so that $\rho \sqsubseteq \rho'$ iff there exists $\mathrm{dom}(\rho) = \mathrm{dom}(\rho')$ and $\forall x \in \mathrm{dom}(\rho).\rho(x) \sqsubseteq \rho'(x)$. For any Γ, we will write $\Box_\Gamma$ for the least partial environment for Γ, viz. the ρ such that $\rho(x) = \Box$ for every $x \in \mathrm{dom}(\Gamma)$. Again, the set $\mathsf{Prefix}(\rho)$ forms a finite distributive lattice where the join $\rho \sqcup \rho'$ is the partial environment $\{x \mapsto \rho(x) \sqcup \rho'(x) \mid x \in \mathrm{dom}(\rho)\}$, 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 $-[x \mapsto -]$ from $\mathsf{Prefix}(\rho) \times \mathsf{Prefix}(v)$ to $\mathsf{Prefix}(\rho[x \mapsto v])$ satisfies:

$$(\rho' \sqcap \rho'')[x \mapsto u \sqcap u'] = \rho'[x \mapsto u'] \sqcap \rho''[x \mapsto u''] \qquad (1)$$

and similarly for joins.

Now suppose $\rho, e \Downarrow_{\mathsf{ref}} v$ and some partial output $u \sqsubseteq v$. We are now able to say what it is for a partial program $(\rho', e') \sqsubseteq (\rho, e)$ to be a "correct" slice of (ρ, e) for u. The idea is that if running (ρ', e') produces a value u' at least as big as u, then (ρ', 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 (ρ', e') is a *slice of* (ρ, e) *for* u.

Definition 2 (Slice of ρ, e for u)**.** Suppose $\rho, e \Downarrow_{\mathsf{ref}} v$ and $u \sqsubseteq v$. Then any $(\rho', e') \sqsubseteq (\rho, e)$ is a *slice of* (ρ, e) *for* u if $\rho', e' \Downarrow_{\mathsf{ref}} u'$ with $u' \sqsupseteq u$.

This makes precise our intuition above: a slice for partial output u 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 $\rho, e \Downarrow_{\mathsf{ref}} v$. Evaluation has several important properties if we restrict the domain of evaluation to just the prefixes of (ρ, e), which we will write as $\mathsf{eval}_{\rho,e}$. The first is that $\mathsf{eval}_{\rho,e}$ 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, $\mathsf{eval}_{\rho,e}$ is monotonic. Third, $\mathsf{eval}_{\rho,e}$ preserves meets. Since this third property implies the second, we just state the following.

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

1. *If* $(\rho', e') \sqsubseteq (\rho, e)$ *then* $\mathsf{eval}_{\rho,e}(\rho', e')$ *is defined: there exists* u *such that* $\rho', e' \Downarrow_{\mathsf{ref}} u$*.*
2. $\mathsf{eval}_{\rho,e}(\rho, e) = v$*.*
3. *If* $(\rho', e') \sqsubseteq (\rho, e)$ *and* $(\rho'', e'') \sqsubseteq (\rho, e)$ *then* $\mathsf{eval}_{\rho,e}(\rho' \sqcap \rho'', e' \sqcap e'') = \mathsf{eval}_{\rho,e}(\rho', e') \sqcap \mathsf{eval}_{\rho,e}(\rho'', e'')$*.*

Technically, $\mathsf{eval}_{\rho,e}$ 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 $\Downarrow_{\mathsf{ref}}$. For parts (1) and (3), we proceed by induction on the derivation of $\rho, e \Downarrow_{\mathsf{ref}} v$, 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 $f_* : A \to B$ is the *upper adjoint* of a unique Galois connection. The *lower adjoint* of f_*, sometimes (confusingly) written $f^* : B \to A$, which preserves joins, inverts f_* in the following minimising way: for any output b of f_*, the lower adjoint yields the smallest input a such that $f_*(a) \sqsupseteq b$. Extensionally, the lower adjoint satisfies $f^*(b) = \bigsqcap \{a \in A \mid f_*(a) \sqsupseteq b\}$.

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

$$\mathsf{uneval}_{\rho,e}(u) = \bigsqcap \left\{(\rho', e') \in \mathsf{Prefix}(\rho, e) \mid \mathsf{eval}_{\rho,e}(\rho', e') \sqsupseteq u\right\}$$

Proof. Immediate from Theorem 1. □

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

But the fact that $\mathsf{uneval}_{\rho,e}$ is uniquely determined by $\mathsf{eval}_{\rho,e}$ 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 $u \sqsubseteq v$. 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 $\mathsf{uneval}_{\rho,e}$, we can then simply define the differential slice as $\mathsf{diff}_{\rho,e}(\Delta(u, v)) = \Delta(\mathsf{uneval}_{\rho,e}(u), \mathsf{uneval}_{\rho,e}(v))$. Thus, differential slicing is straightforward once we have a slicing algorithm.

$$
\begin{array}{rrl}
\text{Traces} & T ::= & \Box \mid x \mid c \mid T_1 \oplus_{c_1,c_2} T_2 \mid (T_1, T_2) \\
& \mid & \texttt{fst}\ T \mid \texttt{snd}\ T \mid \texttt{inl}\ T \mid \texttt{inr}\ T \\
& \mid & \texttt{case}\ T\ \texttt{of}\ \{\texttt{inl}(x_1).T_1; \texttt{inr}(x_2).e_2\} \\
& \mid & \texttt{case}\ T\ \texttt{of}\ \{\texttt{inl}(x_1).e_1; \texttt{inr}(x_2).T_2\} \\
& \mid & \texttt{fun}\ f(x).e \mid T_1\ T_2 \triangleright f(x).T \\
& \mid & \texttt{roll}\ T \mid \texttt{unroll}\ T
\end{array}
$$

Figure 8. Syntax of traces

4. Program Slicing as Backwards Execution

In Section 3, we showed 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 backwards from unneeded parts of the output to unneeded parts of the input. To this end, we record, as a trace, certain information during execution to allow the computation to be "rewound" by an *unevaluation* algorithm that given slicing criterion (a partial output) reconstructs a the least partial program that evaluates to a result consistent with that criterion.

4.1 Abstract syntax

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 $\Box$; as with expressions, this induces a partial order on traces, which we again denote by $\sqsubseteq$, and a lattice $\mathsf{Prefix}(T)$ 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 τ in Γ. When we are not concerned with τ but only with Γ, 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 Γ' extended with bindings for f and x. Note that if $\Gamma \vdash T : \tau$ and $S \sqsubseteq T$, then $\Gamma \vdash S : \tau$.

The Slicer implemention associates every trace node with a value, 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 $\Downarrow_{\mathsf{ref}}$, except that they construct a trace as well as a value. Tracing evaluation for an expression $\Gamma \vdash e : \tau$ in environment ρ for Γ, 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:

Lemma 3 (Determinism of $\Downarrow$)**.** *If $\rho, e \Downarrow v, T$ and $\rho, e \Downarrow v', T'$ then $v = v'$ and $T = T'$.*

The tracing semantics and the reference semantics agree on values:

Theorem 2. $\rho, e \Downarrow_{\mathsf{ref}} v \iff \exists T. \rho, e \Downarrow v, T$

Tracing evaluation is both type-preserving and, when domain-restricted to the prefixes of a terminating computation (ρ, e), a meet- and join-preserving function. For the latter property, we do not state a theorem since we will use the reference semantics where possible in what follows.

$$\boxed{\Gamma \vdash T : \tau}$$

$$\frac{}{\Gamma \vdash \Box : \tau} \qquad \frac{}{\Gamma \vdash () : \texttt{1}} \qquad \frac{}{\Gamma \vdash x : \tau}\ x : \tau \in \Gamma$$

$$\frac{}{\Gamma \vdash c : b}\ c : b \in \Sigma$$

$$\frac{\vdash c_1 : b_1 \qquad \begin{array}{c}\Gamma \vdash T_1 : b_1 \\ \Gamma \vdash T_2 : b_2\end{array} \qquad \vdash c_2 : b_2}{\Gamma \vdash T_1 \oplus_{c_1,c_2} T_2 : \tau}\ \oplus : b_1 \times b_2 \to \tau \in \Sigma$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ f(x).e : \tau_1 \to \tau_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 : \tau_1 \to \tau_2 \\ \Gamma \vdash T_2 : \tau_1 \qquad \Gamma', f : \tau_1 \to \tau_2, x : \tau_1 \vdash T : \tau_2\end{array}}{\Gamma \vdash T_1\ T_2 \triangleright f(x).T : \tau_2}$$

$$\frac{\Gamma \vdash T_1 : \tau_1 \qquad \Gamma \vdash T_2 : \tau_2}{\Gamma \vdash (T_1, T_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash T : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\ T : \tau_1}$$

$$\frac{\Gamma \vdash T : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}\ T : \tau_2} \qquad \frac{\Gamma \vdash T : \tau_1}{\Gamma \vdash \texttt{inl}\ T : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash T : \tau_2}{\Gamma \vdash \texttt{inr}\ T : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash T : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash T_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}\ T\ \texttt{of}\ \{\texttt{inl}(x_1).T_1; \texttt{inr}(x_2).e_2\} : \tau}$$

$$\frac{\Gamma \vdash T : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash T_2 : \tau}{\Gamma \vdash \texttt{case}\ T\ \texttt{of}\ \{\texttt{inl}(x_1).e_1; \texttt{inr}(x_2).T_2\} : \tau}$$

$$\frac{\Gamma \vdash T : \mu\alpha.\tau}{\Gamma \vdash \texttt{unroll}\ T : \tau[\mu\alpha.\tau/\alpha]} \qquad \frac{\Gamma \vdash T : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \texttt{roll}\ T : \mu\alpha.\tau}$$

Figure 9. Typing rules for traces

Lemma 4 (Type preservation for $\Downarrow$)**.** *If $\Gamma \vdash e : \tau$ and $\Gamma \vdash \rho$ with $\rho, e \Downarrow v, T$, then $\vdash v : \tau$ and $\Gamma \vdash T : \tau$.*

With these properties in mind, we can explain the tracing evaluation judgment. The idea is that tracing evaluation equips every value with a trace which describes how it was computed, where the trace takes the form of an unrolling of the original expression, akin to inlining the definition of functions into every call site.

Least elements are preserved, so the trace of the expression $\Box$ is the trace $\Box$. The trace of a variable x is the corresponding trace form x; in general the trace of $\Gamma \vdash e : \tau$ is not closed but is instead also typed in Γ. The traces of other nullary expressions are just the corresponding nullary trace form. For non-nullary forms, such as projections, pairs, and `case` expressions, the general pattern is to produce a trace which looks like the original expression except that any executed sub-expressions have been inflated into their traces. For example a trace of the form $\texttt{case}\ T\ \texttt{of}\ \{\texttt{inl}(x_1).T_1; \texttt{inr}(x_2).e_2\}$ records the scrutinee and the taken branch unrolled into their respective executions. The non-taken branch e_2 is kept in the trace to be consistent with our notion of a trace as an unrolled expression.

Primitive operations are special because they are black boxes: we have no information about how their outputs relate to their inputs. For subsequent slicing, we therefore record not only operand traces, but also the values of the operands. This allows the unevaluation algorithm to proceed through a primitive operation into the operand computations. Alternative approaches to primitives are discussed in Section 4.3 below.

The main point at which traces diverge from expressions is with function calls. An application trace $T_1\ T_2 \triangleright f(x).T$ records not only the evaluation T_1 and T_2 of the function and argument, but also the evaluation T of the function body, which may contain free occurrences of f and x. The notation $f(x).T$ denotes that f and x are re-bound at this point in the trace.

$\boxed{\rho, e \Downarrow v, T}$

$$\frac{}{\rho, \square \Downarrow \square, \square} \qquad \frac{}{\rho, x \Downarrow \rho(x), x} \qquad \frac{}{\rho, c \Downarrow c, c}$$

$$\frac{\rho, e_1 \Downarrow c_1, T_1 \qquad \rho, e_2 \Downarrow c_2, T_2}{\rho, e_1 \oplus e_2 \Downarrow c_1 \mathbin{\hat{\oplus}} c_2, T_1 \oplus_{c_1,c_2} T_2} \qquad \frac{\rho, e_1 \Downarrow \square, T_1}{\rho, e_1 \oplus e_2 \Downarrow \square, \square}$$

$$\frac{\rho, e_1 \Downarrow c_1, T_1 \qquad \rho, e_2 \Downarrow \square, T_2}{\rho, e_1 \oplus e_2 \Downarrow \square, \square}$$

$$\frac{}{\rho, \texttt{fun}\ f(x).e \Downarrow \langle \rho, \texttt{fun}\ f(x).e \rangle, \texttt{fun}\ f(x).e}$$

$$\frac{\rho, e_1 \Downarrow v_1, T_1 \qquad \rho, e_2 \Downarrow v_2, T_2 \qquad \rho'[f \mapsto v_1][x \mapsto v_2], e \Downarrow v, T}{\rho, e_1\ e_2 \Downarrow v, T_1\ T_2 \triangleright f(x).T}\ v_1 = \langle \rho', \texttt{fun}\ f(x).e \rangle$$

$$\frac{\rho, e_1 \Downarrow \square, T_1}{\rho, e_1\ e_2 \Downarrow \square, \square} \qquad \frac{\rho, e_1 \Downarrow v_1, T_1 \qquad \rho, e_2 \Downarrow v_2, T_2}{\rho, (e_1, e_2) \Downarrow (v_1, v_2), (T_1, T_2)}$$

$$\frac{\rho, e \Downarrow (v_1, v_2), T}{\rho, \texttt{fst}\ e \Downarrow v_1, \texttt{fst}\ T} \qquad \frac{\rho, e \Downarrow \square, T}{\rho, \texttt{fst}\ e \Downarrow \square, \square} \qquad \frac{\rho, e \Downarrow (v_1, v_2), T}{\rho, \texttt{snd}\ e \Downarrow v_2, \texttt{snd}\ T}$$

$$\frac{\rho, e \Downarrow \square, T}{\rho, \texttt{snd}\ e \Downarrow \square, \square} \qquad \frac{\rho, e \Downarrow v, T}{\rho, \texttt{inl}\ e \Downarrow \texttt{inl}\ v, \texttt{inl}\ T}$$

$$\frac{\rho, e \Downarrow v, T}{\rho, \texttt{inr}\ e \Downarrow \texttt{inr}\ v, \texttt{inr}\ T}$$

$$\frac{\rho, e \Downarrow \texttt{inl}\ v_1, T \qquad \rho[x_1 \mapsto v_1], e_1 \Downarrow v, T_1}{\begin{array}{l}\rho, \texttt{case}\ e\ \texttt{of}\ \{\texttt{inl}(x_1).e_1; \texttt{inr}(x_2).e_2\} \Downarrow \\ v, \texttt{case}\ T\ \texttt{of}\ \{\texttt{inl}(x_1).T_1; \texttt{inr}(x_2).e_2\}\end{array}}$$

$$\frac{\rho, e \Downarrow \texttt{inr}\ v_2, T \qquad \rho[x_2 \mapsto v_2], e_2 \Downarrow v, T_2}{\begin{array}{l}\rho, \texttt{case}\ e\ \texttt{of}\ \{\texttt{inl}(x_1).e_1; \texttt{inr}(x_2).e_2\} \Downarrow \\ v, \texttt{case}\ T\ \texttt{of}\ \{\texttt{inl}(x_1).e_1; \texttt{inr}(x_2).T_2\}\end{array}}$$

$$\frac{\rho, e \Downarrow \square, T}{\rho, \texttt{case}\ e\ \texttt{of}\ \{\texttt{inl}(x_1).e_1; \texttt{inr}(x_2).e_2\} \Downarrow \square, \square}$$

$$\frac{\rho, e \Downarrow v, T}{\rho, \texttt{roll}\ e \Downarrow \texttt{roll}\ v, \texttt{roll}\ T} \qquad \frac{\rho, e \Downarrow \texttt{roll}\ v, T}{\rho, \texttt{unroll}\ e \Downarrow v, \texttt{unroll}\ T}$$

$$\frac{\rho, e \Downarrow \square, T}{\rho, \texttt{unroll}\ e \Downarrow \square, \square}$$

Figure 10. Tracing semantics: call-by-value evaluation

Finally the hole propagation rules, which apply when for example a scrutinee evaluates to $\square$, always produce a hole trace.

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 value $\vdash v : \tau$ and trace $\Gamma \vdash T : \tau$, the judgment $v, T \Downarrow^{-1} \rho, e$ states that T can be used to unevaluate v to partial environment $\Gamma \vdash \rho$ and partial expression $\Gamma \vdash e : \tau$. Later we will show that these side-conditions are satisfied whenever T is produced by tracing evaluation to v (Theorem 4 below) or by slicing another trace with respect to v (Theorem 7, Section 5).

A key part of the definition which for convenience is omitted from Figure 11 is as follows. Every time a rule takes the join of two values or environments there is an implicit side-condition requiring that the joins exist. For example, the application rule has a side-condition stating that $v_1 \sqsubseteq v_1{}'$ and $\langle \rho, \texttt{fun}\ f(x).e \rangle \sqsubseteq v_1{}'$ have a least upper bound, plus another side-condition stating that ρ_1 and ρ_2 have a least upper bound.

$\boxed{v, T \Downarrow^{-1} \rho, e \text{ where } \Gamma \vdash T : \tau}$

$$\frac{}{\square, T \Downarrow^{-1} \square_\Gamma, \square} \qquad \frac{}{v, x \Downarrow^{-1} \square_{\Gamma.x \mapsto v}, x}\ v \neq \square$$

$$\frac{}{c, c \Downarrow^{-1} \square_\Gamma, c} \qquad \frac{c_2, T_2 \Downarrow^{-1} \rho_2, e_2 \qquad c_1, T_1 \Downarrow^{-1} \rho_1, e_1}{v, T_1 \oplus_{c_1,c_2} T_2 \Downarrow^{-1} \rho_1 \sqcup \rho_2, e_1 \oplus e_2}\ v \neq \square$$

$$\frac{}{\langle \rho, \texttt{fun}\ f(x).e \rangle, \texttt{fun}\ f(x).e' \Downarrow^{-1} \rho, \texttt{fun}\ f(x).e}$$

$$\frac{v, T \Downarrow^{-1} \rho[f \mapsto v_1][x \mapsto v_2], e \qquad v_2, T_2 \Downarrow^{-1} \rho_2, e_2 \qquad v_1 \sqcup \langle \rho, \texttt{fun}\ f(x).e \rangle, T_1 \Downarrow^{-1} \rho_1, e_1}{v, T_1\ T_2 \triangleright f(x).T \Downarrow^{-1} \rho_1 \sqcup \rho_2, e_1\ e_2}\ v \neq \square$$

$$\frac{v_2, T_2 \Downarrow^{-1} \rho_2, e_2 \qquad v_1, T_1 \Downarrow^{-1} \rho_1, e_1}{(v_1, v_2), (T_1, T_2) \Downarrow^{-1} \rho_1 \sqcup \rho_2, (e_1, e_2)}$$

$$\frac{(v_1, \square), T \Downarrow^{-1} \rho, e}{v_1, \texttt{fst}\ T \Downarrow^{-1} \rho, \texttt{fst}\ e}\ v_1 \neq \square \qquad \frac{(\square, v_2), T \Downarrow^{-1} \rho, e}{v_2, \texttt{snd}\ T \Downarrow^{-1} \rho, \texttt{snd}\ e}\ v_2 \neq \square$$

$$\frac{v, T \Downarrow^{-1} \rho, e}{\texttt{inl}\ v, \texttt{inl}\ T \Downarrow^{-1} \rho, \texttt{inl}\ e} \qquad \frac{v, T \Downarrow^{-1} \rho, e}{\texttt{inr}\ v, \texttt{inr}\ T \Downarrow^{-1} \rho, \texttt{inr}\ e}$$

$$\frac{v, T_1 \Downarrow^{-1} \rho_1[x_1 \mapsto v_1], e_1 \qquad \texttt{inl}\ v_1, T \Downarrow^{-1} \rho, e}{\begin{array}{l}v, \texttt{case}\ T\ \texttt{of}\ \{\texttt{inl}(x_1).T_1; \texttt{inr}(x_2).e_2\} \Downarrow^{-1} \\ \rho_1 \sqcup \rho, \texttt{case}\ e\ \texttt{of}\ \{\texttt{inl}(x_1).e_1; \texttt{inr}(x_2).\square\}\end{array}}\ v \neq \square$$

$$\frac{v, T_2 \Downarrow^{-1} \rho_2[x_2 \mapsto v_2], e_2 \qquad \texttt{inr}\ v_2, T \Downarrow^{-1} \rho, e}{\begin{array}{l}v, \texttt{case}\ T\ \texttt{of}\ \{\texttt{inl}(x_1).e_1; \texttt{inr}(x_2).T_2\} \Downarrow^{-1} \\ \rho_2 \sqcup \rho, \texttt{case}\ e\ \texttt{of}\ \{\texttt{inl}(x_1).\square; \texttt{inr}(x_2).e_2\}\end{array}}\ v \neq \square$$

$$\frac{v, T \Downarrow^{-1} \rho, e}{\texttt{roll}\ v, \texttt{roll}\ T \Downarrow^{-1} \rho, \texttt{roll}\ e}$$

$$\frac{\texttt{roll}\ v, T \Downarrow^{-1} \rho, e}{v, \texttt{unroll}\ T \Downarrow^{-1} \rho, \texttt{unroll}\ e}\ v \neq \square$$

Figure 11. Slicing rules: unevaluation

Unevaluation traverses the trace and folds it back into an expression from which the unneeded bits have been discarded. As with evaluation, least elements are preserved, which simply means that "holes map to holes": unevaluating the value $\square$ produces the expression $\square$ and $\square_\Gamma$, the least environment for Γ. Unevaluating the trace of a variable x with v yields x as an expression and the least environment for Γ mapping x to v, which we write as $\square_{\Gamma.x \mapsto v}$.

The general pattern for non-nullary trace constructors is that the traces of the sub-computations are unevaluated and the resulting partial environments joined. For example we unevaluate a pair trace with a pair (v_1, v_2) by unevaluating the respective components with v_1 and v_2 and joining the results. When binders are involved, the fact that traces are typed allows us to safely extract partial values for the bound variables. For example with a `case` trace for `inl`, the selected branch is unevaluated, producing a partial environment of 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.

Unevaluating the application of a primitive operation retrieves the values c_1 and c_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 every primitive operation provide its own adjoint slicing operation, although this places additional burden on the implementer.

The application rule is the most interesting. For a trace $T_1\ T_2 \triangleright f(x).T$, we unevaluate T to obtain a slice e of the original function body, plus an environment $\rho[f \mapsto v_1][x \mapsto v_2]$ where ρ is a slice

of the environment in which the closure was captured, and v_1 and v_2 are slices describing the usage of f and x respectively inside T. Since T contains all recursive uses of the function, v_1 (which is a partial closure) captures how much of f was used below this step of the computation. We then join v_1 with $\langle \rho, \texttt{fun}\ f(x).e \rangle$ to merge in information about the usage of the function at the present step, and use it to unevaluate T_1.

4.4 Correctness of tracing evaluation

Unevaluation is deterministic, which is again a straightforward induction, relying on the $v \neq \square$ 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 *explains* v. We can think of the unique (ρ, 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 ρ, 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 $\mathsf{tr\text{-}uneval}_{v,T}$ from $\mathsf{Prefix}(v)$ to $\mathsf{Prefix}(\rho, e)$. In fact $\mathsf{tr\text{-}uneval}_{v,T}$ also preserves meets and joins, but monotonicity is sufficient here.

Definition 4 ($\mathsf{tr\text{-}uneval}_{v,T}$)**.** Suppose T explains v. Then define $\mathsf{tr\text{-}uneval}_{v,T}$ to be $\Downarrow^{-1}$ 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 $\mathsf{tr\text{-}uneval}_{v,T}$ is a prefix of v.

Theorem 3 (Monotonicity of explanation)**.**
Suppose T explains v. Then:

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

Proof. See Appendix (supplementary material). □

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 ρ, e subscript from $\mathsf{tr\text{-}eval}_{\rho,e}$ when it is clear from the context that the argument is a prefix of (ρ, e).

Theorem 4 (Explanations are program prefixes)**.**
Suppose $\rho, e \Downarrow v, T$. Then T explains v. Moreover, for any $(\rho', e') \sqsubseteq (\rho, e)$:

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

Proof. See Appendix (supplementary material). □

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 $u \neq \square$, we must also have had $U \neq \square$, since otherwise unevaluation would have got stuck. But dually, we can also observe that if U were used to unevaluate an intermediate value $u = \square$, 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 $T \sqsupseteq S$ explains v. Moreover, for any $u \sqsubseteq v$, we have $\mathit{tr\text{-}uneval}_S(u) = \mathit{tr\text{-}uneval}_T(u)$.*

Proof. See Appendix (supplementary material). □

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

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

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

$$\begin{array}{lll} & \mathsf{tr\text{-}uneval}_S(u) & \\ = & \mathsf{tr\text{-}uneval}_T(u) & (T \sqsupseteq S \text{ and Lemma 6}) \\ \sqsubseteq & \mathsf{tr\text{-}uneval}_T(v) & \text{(Theorem 3)} \\ \sqsubseteq & (\rho, e) & \text{(Theorem 4)} \end{array}$$

□

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

Proof. See Appendix (supplementary material). □

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 $\rho, e \Downarrow v, T$. For any $u \sqsubseteq v$ and $(\rho', e') \sqsubseteq (\rho, e)$ we have:*

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

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

$$\begin{array}{lll} \mathsf{tr\text{-}uneval}_T(u) = \mathsf{tr\text{-}uneval}_S(u) & & (S \sqsubseteq T, \text{Lemma 6}) \\ \sqsubseteq \mathsf{tr\text{-}uneval}_S(u') & & \text{(Theorem 3)} \\ \sqsubseteq (\rho', e') & & \text{(Theorem 4)} \end{array}$$

For the $\impliedby$ direction, suppose $\mathsf{tr\text{-}uneval}_{v,T}(u) \sqsubseteq (\rho', e')$. Then:

$$\begin{array}{ll} \mathsf{eval}(\rho', e') \sqsupseteq \mathsf{eval}(\mathsf{tr\text{-}uneval}_T(u)) & \text{(Theorem 1)} \\ \sqsupseteq u & \text{(Theorem 5)} \end{array}$$

□

5. Trace Slicing

As discussed in Section 2, program slices omit information that can be essential for understanding how a program computed a result. Indeed, it is common that distinct criteria on the output (for example picking out different elements of a list) yield the same program slice. This is often because the program is computing some aggregate property of its input, such as an average or total order.

In this section, we show how given a trace T which explains v, we can calculate a least prefix S of T which still preserves

$\boxed{v, T \searrow \rho, S \text{ where } \Gamma \vdash T : \tau}$

$$\frac{}{\Box, T \searrow \Box_\Gamma, \Box} \qquad \frac{}{v, x \searrow \Box_{\Gamma.x \mapsto v}, x}\ v \neq \Box \qquad \frac{}{c, c \searrow \Box_\Gamma, c}$$

$$\frac{c_2, T_2 \searrow \rho_2, S_2 \qquad c_1, T_1 \searrow \rho_1, S_1}{v, T_1 \oplus_{c_1,c_2} T_2 \searrow \rho_1 \sqcup \rho_2, S_1 \oplus_{c_1,c_2} S_2}\ v \neq \Box$$

$$\frac{}{\langle \rho, \texttt{fun}\ f(x).e\rangle, \texttt{fun}\ f(x).e' \searrow \Box_\Gamma, \texttt{fun}\ f(x).\Box}$$

$$\frac{\begin{array}{c} v, T \searrow \rho[f \mapsto v_1][x \mapsto v_2], S \qquad v, T \Downarrow^{-1}\ \_, e \\ v_2, T_2 \searrow \rho_2, S_2 \qquad v_1 \sqcup \langle \rho, \texttt{fun}\ f(x).e\rangle, T_1 \searrow \rho_1, S_1 \end{array}}{v, T_1\ T_2 \triangleright f(x).T \searrow \rho_1 \sqcup \rho_2, S_1\ S_2 \triangleright f(x).S}\ v \neq \Box$$

$$\frac{v_2, T_2 \searrow \rho_2, S_2 \qquad v_1, T_1 \searrow \rho_1, S_1}{(v_1, v_2), (T_1, T_2) \searrow \rho_1 \sqcup \rho_2, (S_1, S_2)}$$

$$\frac{(v_1, \Box), T \searrow \rho, S}{v_1, \texttt{fst}\ T \searrow \rho, \texttt{fst}\ S}\ v_1 \neq \Box \qquad \frac{(\Box, v_2), T \searrow \rho, S}{v_2, \texttt{snd}\ T \searrow \rho, \texttt{snd}\ S}\ v_2 \neq \Box$$

$$\frac{v, T \searrow \rho, S}{\texttt{inl}\ v, \texttt{inl}\ T \searrow \rho, \texttt{inl}\ S} \qquad \frac{v, T \searrow \rho, S}{\texttt{inr}\ v, \texttt{inr}\ T \searrow \rho, \texttt{inr}\ S}$$

$$\frac{\begin{array}{c} v, T_1 \searrow \rho_1[x_1 \mapsto v_1], S_1 \qquad \texttt{inl}\ v_1, T \searrow \rho, S \\ v, \texttt{case}\ T\ \texttt{of}\ \{\texttt{inl}(x_1).T_1; \texttt{inr}(x_2).e_2\} \searrow \\ \rho_1 \sqcup \rho, \texttt{case}\ S\ \texttt{of}\ \{\texttt{inl}(x_1).S_1; \texttt{inr}(x_2).\Box\} \end{array}}\ v \neq \Box$$

$$\frac{\begin{array}{c} v, T_2 \searrow \rho_2[x_2 \mapsto v_2], S_2 \qquad \texttt{inr}\ v_2, T \searrow \rho, S \\ v, \texttt{case}\ T\ \texttt{of}\ \{\texttt{inl}(x_1).e_1; \texttt{inr}(x_2).T_2\} \searrow \\ \rho_2 \sqcup \rho, \texttt{case}\ S\ \texttt{of}\ \{\texttt{inl}(x_1).\Box; \texttt{inr}(x_2).S_2\} \end{array}}\ v \neq \Box$$

$$\frac{v, T \searrow \rho, S}{\texttt{roll}\ v, \texttt{roll}\ T \searrow \rho, \texttt{roll}\ S} \qquad \frac{\texttt{roll}\ v, T \searrow \rho, S}{v, \texttt{unroll}\ T \searrow \rho, \texttt{unroll}\ S}\ v \neq \Box$$

Figure 12. Trace slicing

the "explanatory power" of T with respect to v, in that S retains sufficient information to unevaluate v. Least slices can provide more specific information about which parts of the computation contribute to a part of the output. The missing proofs can be found in the companion technical report [20].

The judgment $v, T \searrow \rho, S$, defined in Figure 12, states that slicing $\Gamma \vdash T : \tau$ with respect to a partial value $\vdash v : \tau$ yields partial environment $\Gamma \vdash \rho$ and partial trace $S \sqsubseteq T$. As with unevaluation, there are side-conditions, which we omit from the figure for convenience, on the rules which take joins, asserting that the joins exist.

The rules are similar in flavour to those for unevaluation, but sub-computations are sliced, rather than unevaluated back to expressions. The significant differences are in the function and application cases. For function traces, we produce the least environment for Γ and the least function trace smaller than T, namely $\texttt{fun}\ f(x).\Box$. For application traces, we both slice *and* unevaluate T: we slice to obtain a *trace* slice S for the function body, and we unevaluate to obtain *expression* slice e for the function body, so that it can be merged into v_1 and used as the slicing criterion for T_1. Unevaluation of the function body also yields a partial environment, but it is identical to the one obtained by slicing, so we disregard it. (See Theorem 6 below.)

5.1 Correctness of trace slicing

Trace slicing is deterministic. The proof is a straightforward induction, relying on the $v \neq \Box$ side-conditions in Figure 12.

Lemma 8. *Suppose $v, T \searrow \rho, S$ and $v, T \searrow \rho', S'$. 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. $v, T \Downarrow^{-1} \rho, e \implies \exists S.v, T \searrow \rho, S.$

Proof. Straightforward induction on the derivation of $v, T \Downarrow^{-1} \rho, e$. The only non-trivial case is the application rule, because we invoke the $\Downarrow^{-1}$ judgment from the $\searrow$ judgment. Then we use that $\Downarrow^{-1}$ 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 $v, T \searrow \rho, S$ then S explains v and $S \sqsubseteq T$.*

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. $v, T \searrow \rho, S \implies \exists e.v, T \Downarrow^{-1} \rho, e.$

Proof. Suppose $v, T \searrow \rho, S$. Then S explains v and $S \sqsubseteq T$ by Theorem 7. But if $S \sqsubseteq T$, then T also explains v by Lemma 6. Then there exist ρ', e such that $v, T \Downarrow^{-1} \rho', e$. But then $\rho = \rho'$ by Theorem 6 and the fact that $\searrow$ 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 $v, T' \searrow \rho, S$, and any $T \sqsubseteq T'$ that explains v. Then $S \sqsubseteq T$.*

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 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 fact that, in our design, a trace is essentially a recursive unfolding of an expression. 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, but after slicing, retain only the slice. Delaying thus pushes the cost of tracing from a run to slicing, and can thus be helpful in the cases where slices are small or computed interactively on demand. For comparison we also implemented a third strategy, *eager*. The three strategies are summarised below:

- The *eager* strategy involves adding strictness annotations to the datatype for traces, along with *seq*s used for implementing our eager evaluation semantics in Haskell, so that the trace is completely constructed before we begin slicing it.

Test Eval(s) Slice/Trace	Strategy	Trace(s)	Slice(s)	Total (s)
`sort`	eager	0.17	0.48	0.65
0.07	lazy	0.12	0.21	0.33
447K/500K	delay	0.01	0.38	0.39
`rbtree`	eager	0.88	2.41	3.29
0.29	lazy	0.66	1.31	1.97
1.55M/1.61M	delay	<0.01	2.48	2.49
`rbtree-len`	eager	0.9	0.02	0.92
0.29	lazy	0.67	0.04	0.71
9K/1.61M	delay	<0.01	0.01	0.01
`vec-sum`	eager	0.16	<0.01	0.16
0.13	lazy	0.13	<0.01	0.13
20/220K	delay	<0.01	<0.01	<0.01

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.

- 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 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. The list lengths are 1000 for `sort`, `rbtree`, and `rbtree-len` and 10000 for `vec-sum`. For `vec-sum` and `sort`, we slice the first element of the result list, for `rbtree` we slice the root value, and for `rbtree-len` we slice the second (length) component of the result. The trace slices for `vec-sum` and `rbtree-length` are small compared to the full trace, while for the other two examples, the trace slice is almost as large as the original trace. We used GHC 6.12.1 with optimization level -O2 running on a MacBook Pro (2.8 GHz Intel Core Duo, 4GB RAM).

The timing results show that the lazy strategy successfully reduces tracing costs compared to eager by around 30%. Slicing costs are also reduced, though, so the total overhead of tracing and slicing is almost 50% less than the eager strategy. Delayed tracing almost eliminates initial tracing costs, and when the resulting slice is small, the slicing cost is also negligible. When a slice is close to the full trace, however, delaying is more expensive than lazy slicing, because expressions may be re-evaluated multiple times in our prototype implementation. Preliminary memory profiling suggests that lazy uses much less memory for both tracing and slicing on the benchmarked programs than eager, while delay usually uses less memory for both tracing and slicing in all examples except tree, often by more than an order of magnitude.

These measurements are consistent with the brief conceptual discussion of the strategies, suggesting that a careful implementation that uses lazy and delayed tracing can further reduce overhead.

7. Related Work

We mentioned in the introduction some of the related work from the large literature on program slicing and related techniques; here we discuss more closely-related work, as well as other related work on provenance, debugging, and execution indexing.

Program slicing. Biswas's [6] and Ochoa *et al.*'s [19] work are the closest to our program slicing techniques. The main difference between Biswas' work and ours is that we support more flexible slicing criteria that permit arbitrary portions of the output to be thrown out or selected—Biswas considers only slicing with respect to the entire output. Ochoa *et al.* present techniques for computing slices under more flexible slicing criteria, but consider only first-order lazy programs. Our techniques appear to be the first where flexible criteria can be used in a strict, higher-order setting. We also realize a limitation of program slices and propose computation slices as a fine-grained techniques for understanding computations.

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.* construct an explicit trace in form of a redex trail [23] so that expressions that are lazily evaluated can be identified. Our techniques also rely on construction of a trace, but our traces reflect closely the syntax of expressions. This allows us to "unevaluate" trace slices back to expressions and to handle higher-order programs in a simple way.

Provenance. Provenance concerns the auditing and analysis of the origins and computational history of data. Provenance is a growing field with applications in databases [8, 9, 13, 14], security [10, 25] and scientific workflow systems [7, 12, 22]. 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. [15], 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 [2] by some of the present authors is also based on big-step techiques similar to those presented here. The authors present a "disclosure slicing" algorithm similar to our trace slicing algorithm, which ensures that a trace retains enough information to show how an output was produced. However, that work does not investigate program slicing or unevaluation, and is unable to slice higher-order values.

View ML [24] has similar high-level goals to ours but is technically quite different to our approach. VML allows the programmer to define special functions called *views* that carry some intensional information in the form of a datatype constructor with arguments showing how parameters were set. These can later be inspected or pattern-matched. This is a useful form of user-defined provenance, but is distinct from that provided by detailed tracing. On the other hand, views could provide a mechanism for abstraction/granularity control with larger traces, potentially addressing some of the issues of scale that we leave as future work.

Debugging. Debugging techniques often involve tracing. The problem of scale – in terms of both resource usage and human cognitive capacity – has received some attention in this area. Nilsson's "piecemeal" tracing for lazy languages [18] builds a trace in the form of an evaluation-dependency tree, but allows the trace to be

started at selected, suspected functions, akin to setting a breakpoint in a traditional debugger. Claessen *et al.'s* work on Hat [11], based on redex trails, explores efficient storage and visualisation techniques and demonstrates that trace-based approaches are feasible for large, multi-module programs. Both these systems also support the declaration of "trusted" components for which no internal trace is recorded.

Execution monitoring. An alternative to tracing is the *execution monitoring* of Kishon and Hudak [17]. 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 [16]. 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 useful information to the programmer but also presents several implementation and user-interface challenges. In this paper, we focused on foundations. We presented a novel algorithm-independent characterisation of the problem of calculating a least dynamic program slice, showed how to support fine-grained differential slicing criteria in the presence of higher-order functions, and showed how to slice reified computations, or traces, as well as programs. Our techniques, realised in our tool Slicer, enable the user to interact with a computation and understand it in terms of the programming language the program was expressed in.

The main challenge that lies ahead for our approach is scaling it to large programs. In the present paper, we briefly described techniques for reducing the overhead of tracing by delaying tracing until it is needed for slicing. Previous work in the area of functional debugging has explored several complementary techniques which may be of use in making trace-based approaches more scalable, including piecemeal construction of traces, "trusted" components for which tracing is disabled, and offline storage. Investigating the applicability of these and other techniques we defer to future work.

References

[1] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[2] U. A. Acar, A. Ahmed, J. Cheney, and R. Perera. A core calculus for provenance. In *Proceedings of the First Conference on Principles of Security and Trust (POST)*, pages 410–429. Springer, 2012.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *POPL*, pages 177–189. ACM, 1983.

[4] D. C. Atkinson and W. G. Griswold. Implementation techniques for efficient data-flow analysis of large programs. In *ICSM*, pages 52–61. IEEE, 2001.

[5] T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, pages 206–222, London, UK, 1993. Springer-Verlag.

[6] S. Biswas. *Dynamic Slicing in Higher-Order Programming Languages*. PhD thesis, University of Pennsylvania, 1997.

[7] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.

[8] P. Buneman, J. Cheney, and S. Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Transactions on Database Systems*, 33(4):28, November 2008.

[9] P. Buneman, S. Khanna, and W. Tan. Why and where: A characterization of data provenance. In *ICDT*, number 1973 in LNCS, pages 316–330, 2001.

[10] A. Cirillo, R. Jagadeesan, C. Pitcher, and J. Riely. Tapido: Trust and authorization via provenance and integrity in distributed objects. In *ESOP*, volume 4960 of *LNCS*, pages 208–223, 2008.

[11] K. Claessen, C. Runciman, O. Chitil, J. Hughes, and M. Wallace. Testing and tracing lazy functional programs using Quickcheck and Hat. In *In 4th Summer School in Advanced Functional Programming, number 2638 in LNCS*, pages 59–99. Springer LNCS, 2003.

[12] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, pages 1345–1350, New York, NY, USA, 2008.

[13] J. N. Foster, T. J. Green, and V. Tannen. Annotated XML: queries and provenance. In *PODS*, pages 271–280, 2008.

[14] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[15] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. Van den Bussche. A formal model of dataflow repositories. In *DILS*, volume 4544 of *LNCS*, pages 105–121, 2007.

[16] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song. Differential slicing: Identifying causal execution differences for security applications. In *IEEE Symposium on Security and Privacy*, 2011.

[17] A. Kishon and P. Hudak. Semantics directed program execution monitoring. *J. Funct. Prog.*, 5(4):501–547, 1995.

[18] H. Nilsson. Tracing piece by piece: affordable debugging for lazy functional languages. In *Proceedings of the 1999 ACM SIGPLAN international conference on Functional programming*, pages 36–47, Paris, France, Sept. 1999. ACM Press.

[19] C. Ochoa, J. Silva, and G. Vidal. Dynamic slicing of lazy functional programs based on redex trails. *Higher Order Symbol. Comput.*, 21(1-2):147–192, 2008.

[20] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. Technical Report MPI-SWS-2012-003, Max Planck Institute for Software Systems, July 2012.

[21] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glck, and P. Thiemann, editors, *Partial Evaluation*, volume 1110 of *LNCS*, pages 409–429. Springer-Verlag, 1996.

[22] Y. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005.

[23] J. Sparud and C. Runciman. Complete and partial redex trails of functional computations. In *IFL 1997*, number 1467 in LNCS, pages 160–177. Springer-Verlag, 1998.

[24] E. Sumii and H. Bannai. VMλ: A functional calculus for scientific discovery. In Z. Hu and M. Rodrguez-Artalejo, editors, *Functional and Logic Programming*, volume 2441 of *Lecture Notes in Computer Science*, pages 290–304. Springer Berlin / Heidelberg, 2002.

[25] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *IEEE Symposium on Security and Privacy*, pages 369–383, 2008.

[26] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[27] M. Weiser. Program slicing. In *ICSE*, pages 439–449, 1981.

[28] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30:1–36, March 2005.