Toward a Theory of Self-Explaining Computation

James Cheney[1], Umut A. Acar[2,3], and Roly Perera[1]

[1] University of Edinburgh
[2] Carnegie Mellon University
[3] INRIA-Rocquencourt

Abstract. Provenance techniques aim to increase the reliability of human judgments about data by making its origin and derivation process explicit. Originally motivated by the needs of scientific databases and scientific computation, provenance has also become a major issue for business and government data on the Web. However, so far provenance has been studied only in relatively restrictive settings: typically, for data stored in databases or scientific workflow systems, and processed by query or workflow languages of limited expressiveness. Long-term provenance solutions require an understanding of provenance in other settings, particularly the general-purpose programming or scripting languages that are used to glue different components such as databases, Web services and workflows together. Moreover, what is required is not only an account of mechanisms for recording provenance, but also a theory of what it means for provenance information to explain or justify a computation. In this paper, we begin to outline a such a theory of *self-explaining computation*. We introduce a model of provenance for a simple imperative language based on *operational derivations* and explore its properties.

1 Introduction

Scientific data (including both raw data and processed results) is now being published and shared online in unprecedented quantities. Understanding the significance, validity, or accuracy of this data depends on understanding its provenance. When data is not confined to a single user, system, or intended application, it is essential to make the origin, ownership history, processing steps, and context or assumptions about the data explicit, to avoid misinterpretation and aid reproducibility. Over the last decade, a wide variety of techniques aimed at addressing this *provenance problem* have been proposed, including new data formats [46, 47] and mechanisms for generating provenance to accompany computations.

Buneman, Khanna and Tan's 2001 paper "Why and Where: A Characterization of Data Provenance" [9] was among the first publications to investigate the problem of provenance in database systems. Although provenance was studied earlier by Wang and Madnick [56], Woodruff and Stonebraker [59] and Cui et al. [20], Buneman et al. [9] has had greater influence (at least measured in terms of citations) than these other works. We conjecture that one reason for this is that Buneman et al. went beyond proposing mechanisms for provenance: they also considered the question of the meaning of provenance. By considering and comparing two models, why-provenance and where-provenance, they made it clear that there might be many different models of provenance, with different advantages and disadvantages, and suitable for different applications.

Provenance techniques have since been studied in the context of databases [9, 29], scientific workflow systems [43, 7], operating systems [48], and inference systems [38] (including recent interest in the Semantic Web community, culminating in a W3C Working Group on Provenance [27, 47]). In each of these contexts, there is a large design space for provenance mechanisms, yet at the same time there is not a clear consensus on the requirements or policies that these mechanisms ought to satisfy. Sometimes even the specifications of the techniques are unclear, or illustrated mainly through intuitive examples.

Instead, a wide variety of informal motivations have been cited, usually not accompanied by precise definitions or proofs of correctness. Such motivations include:

- To record a complete derivation of a program or inference process execution. [23, 60, 48]
- To guarantee repeatability, replayability or reproducibility. [23, 45]
- To explain causal structure, history, influence or dependence. [40, 17, 13, 15]
- To show where result data has been copied from, how result records were composed from input records, or why results were produced. [9, 8, 29, 16]
- To validate a computation to ensure it is correct. [41]
- To diagnose and repair errors in computations involving components that are not believed to be reliable. [39]
- To facilitate efficient recomputation and comparison of computations, including recomputation from different inputs or in different computational environments. [23, 5, 17, 29]

As of 2010, there were over 400 research papers on provenance in computer systems to date [44]. However, not all of them observe Lamport's rule "State the Problem Before Describing the Solution" [35]: instead, many present a proposed solution and then argue that it is a solution on its own terms, without making the problem it solves explicit. This state of affairs should be compared with the state of security research twenty-five years ago, when a wide variety of (often proprietary) security solutions were being proposed without a clear understanding of what problems they solved (or were meant to solve). In an influential essay, Good [28] argued that foundational understanding (theories and mathematical models) were necessary for computer security. We believe provenance research is in a similar state today: there are few formal models or crisp definitions of the requirements for provenance or proofs that actual techniques achieve their purported goals.

In previous work [17], we highlighted several hazards implicit in this state of affairs, which we called *provenance failures*. A provenance failure is a loss or risk exposed by failure to properly manage provenance: for example, losses due to outdated information in online trading [11] or due to inaccurate scientific results [42]. Government agencies may also view leaks as provenance failures [55], and as of this writing (June 2013), it is widely reported [30] that intelligence agencies are combining massive computing resources with unfettered access to metadata about phone calls and Internet use. Whether one views these developments as essential tools for fighting terrorism or unacceptable hazards to privacy and individual liberty, one cannot deny that the problems of protecting and securing metadata are becoming just as important as those for raw data.

Since 2009, there has been a major effort to define standards for provenance on the World Wide Web [47]. Provenance techniques are now being widely advocated as a basis for trusting online data and scientific results. However, if these techniques are not placed on a firm foundation, then this effort is doomed to failure: if the problems to be solved by provenance are not formulated precisely, then proposed solutions will, at best, provide a false sense of security.

For the purposes of this paper, we consider provenance to be any information, usually not already provided by the system, describing some aspect of a system's run-time behavior (or of data flowing through it). Our view is that general-purpose systems, including programming languages, should be equipped with general-purpose notions of provenance that are (a) clearly specified, (b) suitable for a variety of typical applications, and (c) equipped with a formal correctness property relating the behavior of the real system to the provenance description.

The first two criteria are relatively easy to satisfy. The aim of this paper is to bring the third requirement into focus and study it. Many of the commonly-stated requirements for provenance amount to a form of *explanation* that adequately accounts for the behavior of the system [22]. However, the precise sense in which some auxiliary data (which we call provenance) actually explains a computational process is seldom explicitly stated. In this paper, we begin to outline a theory of *self-explaining computation*, in which the semantics of provenance and its relationship to the conventional semantics of a programming language (or behavior of a system) are the objects of study.

What are the open questions that a theory of self-explaining computation should address? These are just a few possibilities:

1. If a system's actual behavior is described by explicit records, how do these constitute explanations? What are different appropriate definitions of explanation and how are they related?
2. Provenance can be recorded according to several different strategies, ranging from coarse-grained to fine-grained. Fine-grained provenance seems more useful or "complete" but can easily grow to dwarf the raw data. How can we understand and quantify the tradeoff between granularity and usefulness?
3. The full provenance record often includes far too much information to be useful. How can we extract subsets of this information that correctly approximates the full record?
4. Some provenance techniques (e.g. minimal witnesses in why-provenance) are *extensional*, or invariant with respect to a conventional semantics of the system, and others are *intensional*, meaning that their behavior can be different

for conventionally-equivalent expressions (e.g. where-provenance). What are the advantages and disadvantages of these different approaches? How can we justify intensional provenance semantics?

The behavior of computer systems can be described programmatically. The study of the semantics of programming languages has explored a large number of alternative approaches to defining the meaning of programs, ranging from *denotational* techniques [53] that interpret program text as an abstract, mathematical object such as a function, to *operational* techniques [50] that explain the behavior of complex program constructs via rules that describe how to evaluate a program step-by-step. We take the view that the theory of self-explaining computation should build on programming language semantics, in order to ensure that the specifications of provenance techniques are clear, and in order to facilitate formalization and proof of correctness properties.

We focus on an operational approach to provenance in the context of an imperative core-language **IMP** [57]. We explore the implications of taking a large-step operational derivation (that is, an explicit natural semantics proof tree [33]) as a form of provenance. We define a semantics for programs that produces both a standard result and an operational derivation tree, which we view as recording all of the information that could be relevant to understanding the program and how it executed (at the **IMP** level of abstraction).

We then consider the problem of *formalizing* some of the requirements above and *extracting* information from traces in order to meet these requirements. For example, we give a candidate definition of source locations (inspired by where-provenance [9]) and then show how this can be extracted from derivations. We also describe the use of derivations for a form of incremental computation (loosely inspired by self-adjusting computation [5]), in order to demonstrate that derivations are expressive enough to meet this strong requirement. We have made additional contributions since the first version of this paper was written [3, 49], and we conclude with a discussion of these results and future steps.

2 Background

To illustrate our approach, we employ a simple imperative programming language **IMP** [57], augmented with pairs as a simple form of data structure. The syntax of **IMP** expressions $e \in \mathrm{Exp}$, commands $c \in \mathrm{Comm}$ and values $v \in \mathrm{Val}$ is as follows:

$$\begin{aligned}
e &::= x \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid (e_1, e_2) \mid \texttt{fst}(e) \mid \texttt{snd}(e) \mid i \mid b \mid e_1 = e_2 \mid e_1 + e_2 \mid \cdots \\
c &::= \texttt{skip} \mid x := e \mid c_1; c_2 \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c \\
v &::= i \mid b \mid (v_1, v_2)
\end{aligned}$$

where $x \in \mathrm{Var}$ denotes variables, $i \in \mathbb{Z}$ denotes integers, and $b \in \mathbb{B}$ denotes boolean values. We will also write $\oplus$ for an arbitrary binary operation, including $+$, $=$, and possibly others.

$$\frac{}{\sigma, x \Downarrow \sigma(x)} \qquad \frac{\sigma, e_1 \Downarrow v_1 \quad \sigma[x := v_1], e_2 \Downarrow v_2}{\sigma, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2} \qquad \frac{i \in \mathbb{Z}}{\sigma, i \Downarrow i} \qquad \frac{\sigma, e_1 \Downarrow i_1 \quad \sigma, e_2 \Downarrow i_2}{\sigma, e_1 + e_2 \Downarrow i_1 + i_2}$$

$$\frac{\sigma, e_1 \Downarrow v_1 \quad \sigma, e_2 \Downarrow v_2}{\sigma, (e_1, e_2) \Downarrow (v_1, v_2)} \qquad \frac{\sigma, e \Downarrow (v_1, v_2)}{\sigma, \texttt{fst}(e) \Downarrow v_1} \qquad \frac{\sigma, e \Downarrow (v_1, v_2)}{\sigma, \texttt{snd}(e) \Downarrow v_1} \qquad \frac{b \in \{\texttt{true}, \texttt{false}\}}{\sigma, b \Downarrow b}$$

Fig. 1. Operational semantics derivation rules for **IMP** expressions

$$\frac{}{\sigma, \texttt{skip} \Downarrow \sigma} \qquad \frac{\sigma, e \Downarrow v}{\sigma, x := e \Downarrow \sigma[x := v]} \qquad \frac{\sigma, c_1 \Downarrow \sigma' \quad \sigma', c_2 \Downarrow \sigma''}{\sigma, c_1; c_2 \Downarrow \sigma''}$$

$$\frac{\sigma, e \Downarrow \texttt{true} \quad \sigma, c_1 \Downarrow \sigma'}{\sigma, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Downarrow \sigma'} \qquad \frac{\sigma, e \Downarrow \texttt{false} \quad \sigma, c_2 \Downarrow \sigma'}{\sigma, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Downarrow \sigma'}$$

$$\frac{\sigma, e \Downarrow \texttt{true} \quad \sigma, c \Downarrow \sigma' \quad \sigma', \texttt{while}\ e\ \texttt{do}\ c \Downarrow \sigma''}{\sigma, \texttt{while}\ e\ \texttt{do}\ c \Downarrow \sigma''} \qquad \frac{\sigma, e \Downarrow \texttt{false}}{\sigma, \texttt{while}\ e\ \texttt{do}\ c \Downarrow \sigma}$$

Fig. 2. Operational semantics derivation rules for **IMP** commands

The meaning of expressions and commands is defined via operational semantics rules as shown in Figures 1 and 2. Our semantics is essentially a standard large-step operational semantics. We consider the set of *stores* Store $=$ Var $\to$ Val and use functions $\sigma \in$ Store to store the values of variables. We write $[]$ for the empty store, $[x_1 := v_1, \ldots, x_n := v_n]$ for a store binding x_i to v_i, and $\sigma[x := v]$ for a store σ updated by replacing the value of x with v. More generally, we write $\sigma[\sigma']$ for σ updated with σ', that is, $\sigma[\sigma'](x) = \sigma'(x)$ if $x \in \mathrm{dom}(\sigma')$ and $\sigma(x)$ otherwise.

In this paper, we view the derivations as explicit data structures, that is, as ordered, ranked trees with nodes labeled with *judgments* J. The judgments we will consider are:

$$J ::= \sigma, e \Downarrow v \mid \sigma, c \Downarrow \sigma'$$

The judgment $\sigma, e \Downarrow v$ indicates that an expression e evaluates to value v in store σ. The judgment $\sigma, c \Downarrow \sigma'$ indicates that a command c evaluates in store σ to store σ'.

The rules in Figures 1 and 2 thus essentially define construction rules for valid derivations. We write $D :: \sigma, e \Downarrow v$ to indicate that D is a valid derivation whose root is labeled with $\sigma, e \Downarrow v$. We may also write patterns of the form

$$\frac{D_1 \quad \cdots \quad D_n}{J}$$

to describe a valid derivation tree whose root is labeled with J and whose immediate subderivations are $D_1, \ldots, D_n$. Figure 3 shows three sample operational semantics derivations.

$$\dfrac{\dfrac{\dfrac{}{[x=4,y=2],x \Downarrow 4} \quad \dfrac{}{[x=4,y=2],2 \Downarrow 2}}{[x=4,y=2],x=2 \Downarrow \texttt{false}} \quad \dfrac{}{[x=4,y=2],y:=4 \Downarrow [x=4,y=4]}}{[x=4,y=2],\texttt{if}\ x=2\ \texttt{then}\ x:=y*2\ \texttt{else}\ y:=4 \Downarrow [x=4,y=4]}$$

$$\dfrac{\dfrac{\dfrac{}{[x=4,y=2],x \Downarrow 4} \quad \dfrac{}{[x=4,y=2],2 \Downarrow 2}}{[x=4,y=2],x=2 \Downarrow \texttt{false}} \quad \dfrac{}{[x=4,y=2],y:=x \Downarrow [x=4,y=4]}}{[x=4,y=2],\texttt{if}\ x=2\ \texttt{then}\ x:=y*2\ \texttt{else}\ y:=x \Downarrow [x=4,y=4]}$$

$$\dfrac{\dfrac{\dfrac{}{[x=3,y=2],x \Downarrow 3} \quad \dfrac{}{[x=3,y=2],2 \Downarrow 2}}{[x=3,y=2],x=2 \Downarrow \texttt{false}} \quad \dfrac{}{[x=3,y=2],y:=x \Downarrow [x=3,y=3]}}{[x=3,y=2],\texttt{if}\ x=2\ \texttt{then}\ x:=y*2\ \texttt{else}\ y:=x \Downarrow [x=3,y=3]}$$

Fig. 3. Example derivation trees

For illustration purposes, we also give the standard denotational semantics of **IMP** programs. Recall that a denotational semantics assigns to each program expression or command a mathematical meaning. Here, we interpret expressions e as functions $\mathcal{E}[\![e]\!]- : \mathrm{Store} \to \mathrm{Val}_\bot$ from stores to values, and commands c as functions $\mathcal{C}[\![c]\!]- : \mathrm{Store} \to \mathrm{Store}_\bot$. Here, we use the standard notation $S_\bot$ to abbreviate $S \uplus \{\bot\}$, that is, the set S augmented with a special "undefined" value $\bot$. One can equivalently think of the interpretations as partial functions $\mathrm{Store} \rightharpoonup \mathrm{Val}$ or $\mathrm{Store} \rightharpoonup \mathrm{Store}$ respectively. The denotational semantics is defined in Figures 4 and 5.

Theorem 1 ([57]). *The denotational and operational semantics are equivalent in the sense that:*

1. *$\mathcal{E}[\![e]\!]\sigma = v$ holds if and only if there exists a derivation D of $\sigma, e \Downarrow v$, and*
2. *$\mathcal{C}[\![c]\!]\sigma = \sigma'$ holds if and only if there exists a derivation D of $\sigma, c \Downarrow \sigma'$.*

The proof is standard, but in the interest of precision we exhibit functions that witness the forward direction by constructing explicit derivations. These are shown in Figures 6 and 7. The function $\mathcal{E}^{\mathcal{D}}[\![e]\!]\sigma$ yields a pair (D, v) of a derivation of $D :: \sigma, e \Downarrow v$ along with the actual value v. Likewise, the function $\mathcal{C}^{\mathcal{D}}[\![c]\!]\sigma$ yields a pair (D, σ'), where $D :: \sigma, c \Downarrow \sigma'$. (The second components of the respective return values, v and σ', are redundant, but this formulation makes the definition more uniform).

In the rest of this paper, we explore the consequences of viewing the derivation obtained by evaluating an **IMP** expression or command as a form of provenance in its own right.

2.1 A note on the overhead and scale of provenance tracking

Our **IMP** language incorporates standard primitive operations found in most general-purpose programming languages, such as arithmetic and booleans. The

$$
\begin{aligned}
\mathcal{E}[\![e]\!] &: \text{Store} \to \text{Val} \\
\mathcal{E}[\![x]\!]\sigma &= \sigma(x) \\
\mathcal{E}[\![\texttt{let } x = e_1 \texttt{ in } e_2]\!]\sigma &= \mathcal{E}[\![e_2]\!]\sigma[x := \mathcal{E}[\![e_1]\!]\sigma] \\
\mathcal{E}[\![i]\!]\sigma &= i \\
\mathcal{E}[\![e_1 + e_2]\!]\sigma &= \mathcal{E}[\![e_1]\!]\sigma + \mathcal{E}[\![e_2]\!]\sigma \\
\mathcal{E}[\![(e_1, e_2)]\!]\sigma &= (\mathcal{E}[\![e_1]\!]\sigma, \mathcal{E}[\![e_2]\!]\sigma) \\
\mathcal{E}[\![\texttt{fst}(e)]\!]\sigma &= v_1 \qquad (\mathcal{E}[\![e]\!]\sigma = (v_1, v_2)) \\
\mathcal{E}[\![\texttt{snd}(e)]\!]\sigma &= v_2 \qquad (\mathcal{E}[\![e]\!]\sigma = (v_1, v_2)) \\
\mathcal{E}[\![b]\!]\sigma &= b \\
\mathcal{E}[\![e_1 = e_2]\!]\sigma &= \begin{cases} \texttt{true} & \mathcal{E}[\![e_1]\!]\sigma = \mathcal{E}[\![e_2]\!]\sigma \\ \texttt{false} & \mathcal{E}[\![e_1]\!]\sigma \neq \mathcal{E}[\![e_2]\!]\sigma \end{cases}
\end{aligned}
$$

Fig. 4. Denotational semantics of expressions

$$
\begin{aligned}
\mathcal{C}[\![c]\!] &: \text{Store} \to \text{Store} \\
\mathcal{C}[\![x := e]\!]\sigma &= \sigma[x := \mathcal{E}[\![e]\!]\sigma] \\
\mathcal{C}[\![c_1; c_2]\!]\sigma &= \mathcal{C}[\![e_2]\!](\mathcal{C}[\![e_1]\!]\sigma) \\
\mathcal{C}[\![\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2]\!]\sigma &= \begin{cases} \mathcal{C}[\![c_1]\!]\sigma & \mathcal{E}[\![e]\!]\sigma = \texttt{true} \\ \mathcal{C}[\![c_2]\!]\sigma & \mathcal{E}[\![e]\!]\sigma = \texttt{false} \end{cases} \\
\mathcal{C}[\![\texttt{while } e \texttt{ do } c]\!]\sigma &= \begin{cases} \mathcal{C}[\![\texttt{while } e \texttt{ do } c]\!](\mathcal{C}[\![c]\!]\sigma) & \mathcal{E}[\![e]\!]\sigma = \texttt{true} \\ \sigma & \mathcal{E}[\![e]\!]\sigma = \texttt{false} \end{cases}
\end{aligned}
$$

Fig. 5. Denotational semantics of commands

derivation trace model we propose above could be prohibitively expensive in raw computational terms if we instrument the program to generating a new derivation step node for each primitive operation. Furthermore, the space needed for such a trace is likely to be large, in direct proportion to the running time.

In this paper we do not consider this practical aspect of provenance, which is obviously important. Our goal is to understand what information, in principle, one might consider as a "most precise" form of provenance, in order to understand what is lost by adopting more practical techniques. Moreover, it may be that the time and space overhead of naive derivation-trace provenance can be avoided, either through finding a more compact representation of the trace, or using standard compression techniques to compress the trace (which may have a lot of redundancy). Naturally, for a deterministic program, one such compressed representation is the original program itself plus its input: this requires no run-time or space overhead for provenance tracking, but requires completely

$$
\begin{aligned}
\mathcal{E}^{\mathcal{D}}[\![e]\!] &: \text{Store} \to \text{Deriv} \times \text{Val} \\
\mathcal{E}^{\mathcal{D}}[\![x]\!]\sigma &= \left(\frac{}{\sigma, x \Downarrow \sigma(x)}, \sigma(x)\right) \\
\mathcal{E}^{\mathcal{D}}[\![i]\!]\sigma &= \left(\frac{}{\sigma, i \Downarrow i}, i\right) \\
\mathcal{E}^{\mathcal{D}}[\![b]\!]\sigma &= \left(\frac{}{\sigma, b \Downarrow b}, b\right) \\
\mathcal{E}^{\mathcal{D}}[\![e_1 \oplus e_2]\!]\sigma &= \mathsf{let}\ (D_1, i_1) = \mathcal{E}^{\mathcal{D}}[\![e_1]\!]\sigma\ \mathsf{in} \\
&\qquad \mathsf{let}\ (D_2, i_2) = \mathcal{E}^{\mathcal{D}}[\![e_2]\!]\sigma\ \mathsf{in}\ \left(\frac{D_1 \quad D_2}{\sigma, e_1 \oplus e_2 \Downarrow i_1 \oplus i_2}, i_1 \oplus i_2\right) \\
\mathcal{E}^{\mathcal{D}}[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!]\sigma &= \mathsf{let}\ (D_1, v_1) = \mathcal{E}^{\mathcal{D}}[\![e_1]\!]\sigma\ \mathsf{in} \\
&\qquad \mathsf{let}\ (D_2, v_2) = \mathcal{E}^{\mathcal{D}}[\![e_2]\!]\sigma[x := v_1]\ \mathsf{in}\ \left(\frac{D_1 \quad D_2}{\sigma, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2}, v_2\right) \\
\mathcal{E}^{\mathcal{D}}[\![(e_1, e_2)]\!]\sigma &= \mathsf{let}\ (D_1, v_1) = \mathcal{E}^{\mathcal{D}}[\![e_1]\!]\sigma\ \mathsf{in} \\
&\qquad \mathsf{let}\ (D_2, v_2) = \mathcal{E}^{\mathcal{D}}[\![e_2]\!]\sigma\ \mathsf{in}\ \left(\frac{D_1 \quad D_2}{\sigma, (e_1, e_2) \Downarrow (v_1, v_2)}, (v_1, v_2)\right) \\
\mathcal{E}^{\mathcal{D}}[\![\texttt{fst}(e)]\!]\sigma &= \mathsf{let}\ (D, (v_1, v_2)) = \mathcal{E}^{\mathcal{D}}[\![e]\!]\sigma\ \mathsf{in}\ \left(\frac{D}{\sigma, \texttt{fst}(e) \Downarrow v_1}, v_1\right) \\
\mathcal{E}^{\mathcal{D}}[\![\texttt{snd}(e)]\!]\sigma &= \mathsf{let}\ (D, (v_1, v_2)) = \mathcal{E}^{\mathcal{D}}[\![e]\!]\sigma\ \mathsf{in}\ \left(\frac{D}{\sigma, \texttt{snd}(e) \Downarrow v_2}, v_2\right)
\end{aligned}
$$

Fig. 6. Extracting derivations for expressions

$$
\begin{aligned}
\mathcal{C}^{\mathcal{D}}[\![c]\!] &: \text{Store} \to \text{Deriv} \times \text{Store} \\
\mathcal{C}^{\mathcal{D}}[\![x := e]\!]\sigma &= \mathsf{let}\ (D, v) = \mathcal{E}^{\mathcal{D}}[\![e]\!]\sigma\ \mathsf{in}\ \left(\frac{D}{\sigma, x := e \Downarrow \sigma[x := v]}, \sigma[x := v]\right) \\
\mathcal{C}^{\mathcal{D}}[\![c_1; c_2]\!]\sigma &= \mathsf{let}\ (D_1, \sigma') = \mathcal{C}^{\mathcal{D}}[\![c_1]\!]\sigma\ \mathsf{in} \\
&\qquad \mathsf{let}\ (D_2, \sigma'') = \mathcal{C}^{\mathcal{D}}[\![c_2]\!]\sigma'\ \mathsf{in}\ \left(\frac{D_1 \quad D_2}{\sigma, c_1; c_2 \Downarrow \sigma''}, \sigma''\right) \\
\mathcal{C}^{\mathcal{D}}[\![\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]\sigma &= \mathsf{let}\ (D, b) = \mathcal{E}^{\mathcal{D}}[\![e]\!]\sigma\ \mathsf{in} \\
&\quad \mathsf{let}\ (D', \sigma') = \mathsf{if}\ b\ \mathsf{then}\ \mathcal{C}^{\mathcal{D}}[\![c_1]\!]\sigma\ \mathsf{else}\ \mathcal{C}^{\mathcal{D}}[\![c_2]\!]\sigma\ \mathsf{in} \\
&\quad \left(\frac{D \quad D'}{\sigma, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Downarrow \sigma'}, \sigma'\right) \\
\mathcal{C}^{\mathcal{D}}[\![\texttt{while}\ e\ \texttt{do}\ c]\!]\sigma &= \mathsf{let}\ (D, b) = \mathcal{E}^{\mathcal{D}}[\![e]\!]\sigma\ \mathsf{in} \\
&\quad \mathsf{if}\ b \\
&\quad \mathsf{then}\ \mathsf{let}\ (D', \sigma') = \mathcal{C}^{\mathcal{D}}[\![c]\!]\sigma\ \mathsf{in} \\
&\qquad\quad \mathsf{let}\ (D'', \sigma'') = \mathcal{C}^{\mathcal{D}}[\![\texttt{while}\ e\ \texttt{do}\ c]\!]\sigma'\ \mathsf{in}\ \left(\frac{D \quad D' \quad D''}{\sigma, \texttt{while}\ e\ \texttt{do}\ c \Downarrow \sigma''}, \sigma''\right) \\
&\quad \mathsf{else}\ \left(\frac{D}{\sigma, \texttt{while}\ e\ \texttt{do}\ c \Downarrow \sigma}, \sigma\right)
\end{aligned}
$$

Fig. 7. Extracting derivations for commands

recomputing the program to perform provenance analysis. Exploring the tradeoff between time and space overhead of provenance tracking vs. provenance analysis is an important area for future work; here, we focus only on defining different provenance analyses in terms of derivation traces.

Another important observation about the overhead and scalabilty of our approach is that our approach is parametric over the primitive operations: they may be (as in our examples) fine-grained, machine arithmetic operations, but they could just as well be coarser-grained, macroscopic steps. Consider, for example, an alternative variant of **IMP** in which the primitive operations include entire external programs. In other words, instead of performing all of our numerically-intensive computation explicitly using **IMP**-level arithmetic, we can consider it as a scripting language for orchestrating larger computational steps that are treated as primitive operations from the point of view of **IMP**'s provenance records. Another interesting area for future work could be to understand how to combine efficient coarse-grained provenance with more-precise, on-demand fine-grained provenance tracking.

3 Finding sources of copied data

As noted in the introduction, our goal is to use operational derivations as a starting point for formalizing various requirements on provenance. We start with the notion of where-provenance [9, 8]. Essentially, where-provenance is intended to track the sources of data copied from the input of a computation to the output. We will define where-provenance for while-programs in two stages: first, we will define where-provenance for straight-line code, and then we will lift the definition to arbitrary programs by erasing derivations to straight-line programs.

Since we have been using abstract syntax trees for values and expression trees, it seems natural to employ *paths* that can be used to address parts of expressions and values. We write $\mathrm{paths}(v)$ for the set of paths that are valid for a value. Specifically, $\mathrm{paths} : \mathrm{Val} \to \{1, 2\}^*$ is defined as:

$$\begin{aligned} \mathrm{paths}(b) = \mathrm{paths}(i) &= \{\epsilon\} \\ \mathrm{paths}((v_1, v_2)) &= 1 \cdot \mathrm{paths}(v_1) \cup 2 \cdot \mathrm{paths}(v_2) \end{aligned}$$

Here, if P is a set of paths, we write $i \cdot P$ for $\{i \cdot p \mid p \in P\}$. Similarly, we use paths of the form $x.p$ to point to parts of variable values in stores. We write $v[p]$ for the value located at path p in v, and we write $v[p := v']$ for the result of replacing the value at path p in v with v'. We extend these notations to environments and environment paths in the obvious way.

Now we first consider the problem of identifying the source path (if any) of a path in the result of an expression.

Definition 1. *Suppose* $\mathcal{E}[\![e]\!]\sigma = v$ *and* $p \in \mathrm{paths}(v)$*. A* source path q *is a path such that* $\sigma[q] = v[p]$ *and for any* v'*, we have if* $\mathcal{E}[\![e]\!]\sigma[q := v'][p] = v'$*.*

In other words, a source path q points to an input value $\sigma[q]$ that is a copy of the value $v[p]$ at result path p: if we change the input σ at q to v' then the change is mirrored at the output v'' at p. (Note that v'' may also differ at other places besides p; consider the expression (x, x).)

This definition of source path is based on the denotational semantics, and so for example two denotationally equivalent expressions such as $x + 0$ and x have the same source path behavior. Because of this, in general it appears difficult to determine source paths exactly: for example, if the primitive operations can encode Boolean formulas, then we can reduce the Boolean satisfiability problem to the problem of determining whether a Boolean variable is always an exact copy of a part of the input. With richer primitive operations such as arithmetic, determining whether source path relationships exist can become undecidable, reducing from Diophantine equation satisfiability.

Nevertheless, we can safely under-approximate the source paths of an expression, as shown in the $\mathsf{src}\,(e, p)$ function:

$$\begin{aligned}
\mathsf{src} &: \mathrm{Var} \times \mathrm{Path} \to \mathrm{Path}_\perp \\
\mathsf{src}\,(e, \perp) &= \perp \\
\mathsf{src}\,(x, p) &= x.p \\
\mathsf{src}\,(i, \epsilon) &= \perp \\
\mathsf{src}\,(b, \epsilon) &= \perp \\
\mathsf{src}\,(e_1 \oplus e_2, \epsilon) &= \perp \\
\mathsf{src}\,((e_1, e_2), \epsilon) &= \perp \\
\mathsf{src}\,((e_1, e_2), i \cdot p) &= \mathsf{src}\,(e_i, p) \\
\mathsf{src}\,(\mathtt{fst}(e), p) &= \mathsf{src}\,(e, 1 \cdot p) \\
\mathsf{src}\,(\mathtt{snd}(e), p) &= \mathsf{src}\,(e, 2 \cdot p) \\
\mathsf{src}\,(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2, p) &= \begin{cases} \mathsf{src}\,(e_1, q) & \text{if } \mathsf{src}\,(e_2, p) = x.q \\ \mathsf{src}\,(e_2, p) & \text{otherwise} \end{cases}
\end{aligned}$$

The cases for constants and primitive functions are obvious. For pair expressions, if the path is ϵ, then we return $\perp$ since the pair value was created by the pair expression. If the pair is $i \cdot p$ for some $i \in \{1, 2\}$, then we find the source of p in the appropriate subderivation. For projection operations `fst` or `snd`, we find the source of $i \cdot p$ where $i = 1$ or $i = 2$ respectively. For `let`-binding, we first find the source of p in the second subderivation. There are then two cases: either the source path is of the form $x.q$ where x was the bound variable, or it is $\perp$ or some other path $y.q$. In the first case, we find the source path of $x.q$ in e_1; in the second, we just return the source path we have already found (or $\perp$).

Theorem 2. *If $D :: \sigma, e \Downarrow v$ and $p \in \mathrm{paths}(v)$ and $\mathsf{src}\,(e, p) = q \neq \perp$ then q is a source path for p.*

Next, we consider commands. The definition of source path above extends naturally to *straight-line* code involving only sequential composition and assignment:

$$s ::= \mathtt{skip} \mid x := e \mid s_1; s_2$$

Again, source paths for commands can be extracted syntactically:

$$\begin{aligned}
\mathsf{src} &: \mathrm{Var} \times \mathrm{Path} \to \mathrm{Path}_\perp \\
\mathsf{src}\,(s, \perp) &= \perp \\
\mathsf{src}\,(\mathtt{skip}, x.p) &= x.p \\
\mathsf{src}\,(x := e, x.p) &= \mathsf{src}\,(e, p) \\
\mathsf{src}\,(y := e, x.p) &= x.p \qquad (x \neq y) \\
\mathsf{src}\,(s_1; s_2, q) &= \mathsf{src}\,(s_1, \mathsf{src}\,(s_2, q))
\end{aligned}$$

The idea is similar to where-provenance for expressions. The assignment command is handled similar to a `let`. Sequential composition is handled by composing src on subexpressions.

Theorem 3. *If $D :: \sigma, c \Downarrow \sigma'$ and $p \in \mathrm{paths}(\sigma')$ and $\mathsf{src}\,(c, p) = q \neq \perp$ then q is a source path for p.*

However, the above notion of source path does not transfer directly to commands with control-flow. For example, in a conditional $\mathtt{if}\ x = 1\ \mathtt{then}\ y = x\ \mathtt{else}\ y = 2$ there is no source path for the value of y, even in the case where $x = 1$ and y seems to be copied from x. As a compromise, we consider a weaker notion, based on the idea of "freezing" the control-flow of a derivation to obtain a straight-line program.

$$\begin{aligned}
\mathsf{freeze}\left(\frac{}{\sigma, \mathtt{skip} \Downarrow \sigma}\right) &= \mathtt{skip} \\
\mathsf{freeze}\left(\frac{D}{\sigma, x := e \Downarrow \sigma'}\right) &= x := e \\
\mathsf{freeze}\left(\frac{D_1 \quad D_2}{\sigma, c_1; c_2 \Downarrow \sigma''}\right) &= \mathsf{freeze}\,(D_1)\,;\mathsf{freeze}\,(D_2) \\
\mathsf{freeze}\left(\frac{D \quad D'}{\sigma, \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \Downarrow \sigma'}\right) &= \mathsf{freeze}\,(D') \\
\mathsf{freeze}\left(\frac{D :: \sigma, e \Downarrow \mathtt{false}}{\sigma, \mathtt{while}\ e\ \mathtt{do}\ c}\right) &= \mathtt{skip} \\
\mathsf{freeze}\left(\frac{D :: \sigma, e \Downarrow \mathtt{true} \quad D' \quad D''}{\sigma, \mathtt{while}\ e\ \mathtt{do}\ c}\right) &= \mathsf{freeze}\,(D')\,;\mathsf{freeze}\,(D'')
\end{aligned}$$

The function $\mathsf{freeze}\,(D)$ gives a straight-line code approximation of the program based on its derivation. We have:

Theorem 4. *If $D :: \sigma, c \Downarrow \sigma'$ then $\sigma, \mathsf{freeze}\,(D) \Downarrow \sigma'$.*

Note, however, that $\mathsf{freeze}\,()$ is still an intensional concept: two derivations of equivalent programs on equal inputs need not have the same straight-line approximation, as illustrated by $D_1 :: [x := 1], \mathtt{if}\ x = 1\ \mathtt{then}\ x := 1\ \mathtt{else}\ \mathtt{skip} \Downarrow [x := 1]$ and $[x := 1], x \Downarrow [x := 1]$. Moreover, the above theorem does not uniquely characterize the behavior of $\mathsf{freeze}\left(\right)$; for example, an alternative definition that simply collects the assignments needed to map σ to σ' would also have the given property. Thus, $\mathsf{freeze}\left(\right)$ represents an intuitive tradeoff between concreteness (avoiding control-flow) and faithfulness to the shape of the original derivation.

Given a derivation $D :: \sigma, e \Downarrow v$ and path p in the result value v, we can then define the source path of p in a general **IMP** program c as $\mathsf{src}\left(\mathsf{freeze}\left(D\right), p\right)$.

Actually, very little of the derivation is needed to compute sources. Inspecting each rule, we never need to examine the input store of any judgment and we seldom need to inspect the return value: we only do this for `while`, and we could potentially avoid this by inferring whether the loop test holds from the structure of the subtree (i.e., a while-subderivation with only one child must correspond to a loop test that evaluates to false). So, in general, if we only want to extract source information then all we really need is the straight-line approximation of the derivation (i.e., $\mathsf{freeze}\left(D\right)$), not the (usually much larger) full derivation with explicit store, expression, and value annotations. The straight-line approximation $\mathsf{freeze}\left(D\right)$ might be viewed as an interesting form of provenance in its own right. We can extract more than just source information from it; for example, we can determine whether an output value was computed by adding two inputs.

A straight-line program could also be viewed as a DAG, following many conventional approaches to provenance such as OPM [46]. Clearly, we could extract an OPM-style DAG from a straight-line program. Moreover, as argued by Cheney [13] and Moreau [45], provenance DAGs can be viewed as a model of computation for the purpose of analyzing the causality or reproducibility of the computation they represent. However, the DAG approximation corresponding to $\mathsf{freeze}\left(D\right)$ does not necessarily provide enough information for full recomputation. In the next section, we consider the related issue of using the derivation as a basis for efficient recomputation based on caching.

4 Dependence and change propagation

Another common motivation for provenance is to understand how parts of the result depend on intermediate computation steps or source data. The notion of dependence plays an important role in programming languages, particularly dependency tracking [1, 2], information flow security [51] and change propagation [5, 4]. As argued in [12, 15], we believe that this is a good starting point for understanding how provenance should link results to the source data they depend on.

Analyzing dependence requires us to consider not just how an expression did evaluate but how its evaluation might change if the inputs were modified. If we expect provenance to explain the results, then what metric should we use to compare different explanations? We believe that an explanation should have *predictive value* in the sense that it can be used to effectively predict how the result might change if the inputs were modified. Of course, the original program also provides this ability, but full recomputation may involve redoing subcomputations where nothing has changed. Thus, a further requirement is that the explanation be concise in the sense that it avoids details of uninteresting parts of the computation that do not change.

Derivation trees already provide all of the information needed to predict the results of changes. In fact, for a deterministic language, the root judgment of a derivation tree already contains the whole program, and we can simply rerun this on any new input and compare the old derivation and result value with the new ones. However, we argue that this does not provide a satisfying explanation. Derivation trees are verbose and it is not easy to propagate changes through them. For example, in Figure 3 if we change the value of x from 4 to 3, the structure of the derivation does not change. A large number of parts of the derivation need to change, because there are many copies of the value of x in the store and return values. In some sense, all we really need to know about the result is that it is a copy of x, and the control flow depended on the fact that $x = 2$ was false. This gives us enough information to predict the result of any change to x that maintains the invariant $x \neq 2$.

To make this precise, consider the function $\mathcal{E}^{\Delta}(D, \delta)$ that takes a derivation D of $\sigma, e \Downarrow v$ and a partial environment δ and constructs the new value v' resulting from evaluating e on $\sigma[\delta]$. Here, δ is an environment that provides new values for some of the variables in σ. We write $\sigma[\delta]$ to indicate the environment that takes values $\delta(x)$ if $x \in \mathrm{dom}(\delta)$ and $\sigma(x)$ otherwise. We also consider an analogous function $\mathcal{C}^{\Delta}(D, \delta)$ that propagates changes through commands.

In Figure 8, we define functions $\mathcal{E}^{\Delta}(-,-)$: Deriv $\times$ Store $\rightarrow$ Val and $\mathcal{C}^{\Delta}(-.-)$: Deriv $\times$ Store $\rightarrow$ Store$_{\perp}$ that attempt to reuse values cached in subderivations wherever possible. Specifically, whenever we can detect that the changed values in δ do not overlap with the free variables of an expression or command, we simply reuse the cached value (for an expression) or return δ (for a command). The following lemma shows that this is safe:

Lemma 1. *If* $\mathrm{dom}(\delta) \cap FV(e) = \emptyset$ *and* $\sigma, e \Downarrow v$ *then* $\sigma[\delta], e \Downarrow v$*. Moreover, if* $\mathrm{dom}(\delta) \cap FV(c) = \emptyset$ *and* $\sigma, c \Downarrow \sigma'$ *then* $\sigma[\delta], c \Downarrow \sigma'[\delta]$*.*

The first rule for expressions says that we can reuse a cached subexpression provided none of its variables have changed in value (that is, $FV(e) \cap \mathrm{dom}(\delta) = \emptyset$). The next few rules essentially just replay evaluation. The rule for `let` deserves discussion: essentially, we recompute the bound expression and compare its value with the previous value cached in the trace. If the values are equal, we recompute the body of the `let` using δ, otherwise, we add the new binding for x to δ. This makes it possible to use cached subderivations more often than if we always added x to δ.

For commands, the rules follow a similar pattern. The first rule indicates that it is safe to skip recomputation of a command whose free variables have not been changed. Assignment follows a pattern similar to `let`. However, we need to recompute subexpressions using the cached stores when the control flow changes, for example if the change affects the result of a conditional test. The rules for conditionals require re-starting evaluation when the control flow changes (we use the denotational semantics for brevity). For example, if a conditional

$$
\begin{aligned}
\mathcal{E}^{\Delta}\left(D :: \sigma, e \Downarrow v, \delta\right) &= v \qquad (\mathrm{dom}(\delta) \cap FV(e) = \emptyset) \\
\mathcal{E}^{\Delta}\left(\frac{}{\sigma, x \Downarrow v}, \delta\right) &= \delta(x) \quad (x \in \mathrm{dom}(\delta)) \\
\mathcal{E}^{\Delta}\left(\frac{D_1 \quad D_2}{\sigma, e_1 \oplus e_2 \Downarrow v}, \delta\right) &= \mathcal{E}^{\Delta}\left(D_1, \delta\right) \oplus \mathcal{E}^{\Delta}\left(D_2, \delta\right) \qquad (\oplus \in \{=, +, \ldots\} \\
\mathcal{E}^{\Delta}\left(\frac{D_1 \quad D_2}{\sigma, (e_1, e_2) \Downarrow (v_1, v_2)}, \delta\right) &= \left(\mathcal{E}^{\Delta}\left(D_1, \delta\right), \mathcal{E}^{\Delta}\left(D_2, \delta\right)\right) \\
\mathcal{E}^{\Delta}\left(\frac{D}{\sigma, \mathtt{fst}(e) \Downarrow v}, \delta\right) &= \mathsf{let}\ (v_1', v_2') = \mathcal{E}^{\Delta}\left(D, \delta\right)\ \mathsf{in}\ v_1' \\
\mathcal{E}^{\Delta}\left(\frac{D}{\sigma, \mathtt{snd}(e) \Downarrow v}, \delta\right) &= \mathsf{let}\ (v_1', v_2') = \mathcal{E}^{\Delta}\left(D, \delta\right)\ \mathsf{in}\ v_2' \\
\mathcal{E}^{\Delta}\left(\frac{D_1 :: \sigma, e_1 \Downarrow v \quad D_2}{\sigma, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2}, \delta\right) &= \begin{cases} \mathcal{E}^{\Delta}\left(D_2, \delta\right) & (\mathcal{E}^{\Delta}\left(D_1, \delta\right) = v) \\ \mathcal{E}^{\Delta}\left(D_2, \delta[x := \mathcal{E}^{\Delta}\left(D_1, \delta\right)]\right) & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 8. Update propagation for expressions

$$
\begin{aligned}
\mathcal{C}^{\Delta}\left(D :: \sigma, c \Downarrow \sigma', \delta\right) &= \delta \qquad (\mathrm{dom}(\delta) \cap FV(c) = \emptyset) \\
\mathcal{C}^{\Delta}\left(\frac{D :: \sigma, e \Downarrow v}{\sigma, x := e \Downarrow \sigma'}, \delta\right) &= \begin{cases} \delta & (\mathcal{E}^{\Delta}\left(D, \delta\right) = v) \\ \delta[x := \mathcal{E}^{\Delta}\left(D, \delta\right)] & \text{otherwise} \end{cases} \\
\mathcal{C}^{\Delta}\left(\frac{D_1 \quad D_2}{\sigma, c_1; c_2 \Downarrow \sigma'}, \delta\right) &= \mathcal{C}^{\Delta}\left(D_2, \mathcal{C}^{\Delta}\left(D_1, \delta\right)\right) \\
\mathcal{C}^{\Delta}\left(\frac{D :: e \Downarrow \mathtt{true} \quad D_1}{\sigma, \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \Downarrow \sigma'}, \delta\right) &= \mathsf{if}\ \mathcal{E}^{\Delta}\left(D, \delta\right)\ \mathsf{then}\ \mathcal{C}^{\Delta}\left(D_1, \delta\right)\ \mathsf{else}\ \mathcal{C}[\![c_2]\!](\sigma[\delta]) \\
\mathcal{C}^{\Delta}\left(\frac{D :: \sigma, e \Downarrow \mathtt{false} \quad D_2}{\sigma, \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \Downarrow \sigma'}, \delta\right) &= \mathsf{if}\ \mathcal{E}^{\Delta}\left(D, \delta\right)\ \mathsf{then}\ \mathcal{C}[\![c_1]\!](\sigma[\delta])\ \mathsf{else}\ \mathcal{C}^{\Delta}\left(D_2, \delta\right) \\
\mathcal{C}^{\Delta}\left(\frac{D :: \sigma, e \Downarrow \mathtt{true} \quad D' \quad D''}{\sigma, \mathtt{while}\ e\ \mathtt{do}\ c \Downarrow \sigma'}, \delta\right) &= \mathsf{if}\ \mathcal{E}^{\Delta}\left(D, \delta\right)\ \mathsf{then}\ \mathcal{C}^{\Delta}\left(D'', \mathcal{C}^{\Delta}\left(D', \delta\right)\right)\ \mathsf{else}\ \delta \\
\mathcal{C}^{\Delta}\left(\frac{D :: \sigma, e \Downarrow \mathtt{false}}{\sigma, \mathtt{while}\ e\ \mathtt{do}\ c \Downarrow \sigma}, \delta\right) &= \mathsf{if}\ \mathcal{E}^{\Delta}\left(D, \delta\right)\ \mathsf{then}\ \mathcal{C}[\![\mathtt{while}\ e\ \mathtt{do}\ c]\!](\sigma[\delta])\ \mathsf{else}\ \delta
\end{aligned}
$$

Fig. 9. Update propagation for commands

test changes from true to false, then we cannot use the subderivation stored for the then-branch; we have to execute the else-branch "from scratch" using ordinary evaluation on the updated store $\sigma[\delta]$. Composition and `while` also follow predictable patterns; here, we use the denotational semantics for commands as shorthand for computing commands "from scratch".

Theorem 5. *If $D :: \sigma, e \Downarrow v$ then $\sigma[\delta], e \Downarrow \mathcal{E}^{\Delta}(D, \delta)$. Similarly, if $D :: \sigma, c \Downarrow \sigma'$ and $\sigma[\delta], c \Downarrow \sigma''$ then $\sigma'' = \sigma'[\mathcal{C}^{\Delta}(D, \delta)]$.*

Note that the second part needs to be stated carefully because there is no guarantee that recomputing a command on a changed input will terminate.

The correctness theorem above essentially states that the functions $\mathcal{E}^{\Delta}(-,-)$ and $\mathcal{C}^{\Delta}(-,-)$ can be used to correctly compute the updated result. We could go further, and augment these functions to calculate the new derivation as well, or the changed part of the derivation. The latter could serve as a rough measure of the amount of "work" needed to recompute; obviously, in many cases the changed part of the derivation will be much smaller than the whole derivation, just as the changed part of the store obtained by $\mathcal{C}^{\Delta}(-,-)$ can be smaller than the whole result store.

Propagating updates through computations efficiently is a subtle issue with a large, still-growing literature (particularly for self-adjusting computation in functional programming [5, 4]). Our goal here is not to introduce a new approach to incremental recomputation that we claim will be more efficient, but only to establish a formal link between derivations-as-provenance and the notions of trace used in incremental recomputation. In particular, the $\mathcal{C}^{\Delta}(D, \delta)$ function highlights one qualitative difference between replaying the whole expression from scratch and derivation-based change propagation: only by recording some information about what happened in a previous run can we avoid fully recomputing each part of the program.

We could also push this idea further in several ways: we could allow finer-grained changes such as updates that change a specific path in a variable's value, not just the whole value; we could consider techniques for controlling the cost of caching by marking subexpressions with checkpointing annotations; we could improve the precision of update propagation for commands by static analysis of assignments; or we could incrementally recompute both the new value and its derivation (or the difference between derivations). Many of these ideas have already been explored in the context of self-adjusting computation, and it is intriguing to consider the possibility of unifying the notion of traces used in efficient self-adjusting computation systems with that needed for provenance.

5 Discussion

Pragmatic concerns, such as ease of use and extensibility, are often cited for employing operational semantics instead of denotational semantics. In particular, extensions such as nondeterminism, concurrency, additional type constructors, object-oriented features, and higher-order functions can be added to an operational semantics comparatively easily. Following the recipe in this paper, each such extension comes equipped with one or more standard notions of "operational derivation" which could be used as a form of provenance. However, the reality is not quite so simple: for example, adding sum types or collection types poses problems for our use of paths to address parts of result values. We discuss the ramifications of these extensions in the rest of this section.

5.1 Sum Types

Functional languages such as ML and Haskell support algebraic datatypes, based on type-theoretic *sum types*. The type $\tau_1 + \tau_2$ represents the disjoint union of types τ_1 and τ_2. Its introduction forms are injection functions $\texttt{inl} : \tau_1 \to \tau_1 + \tau_2$ and $\texttt{inr} : \tau_2 \to \tau_1 + \tau_2$, and its elimination form is a case construct that performs pattern matching.

$$\begin{aligned} e &::= \cdots \mid \texttt{inl}(e) \mid \texttt{inr}(e) \mid \texttt{case}\ e\ \texttt{of}\ \{x.e_1 \mid y.e_2\} \\ v &::= \cdots \mid \texttt{inl}(v) \mid \texttt{inr}(v) \end{aligned}$$

Sum types and the associated programming constructs can be handled similarly to booleans and conditionals:

$$\frac{\sigma, e \Downarrow v}{\sigma, \texttt{inl}(e) \Downarrow \texttt{inl}(v)} \qquad \frac{\sigma, e \Downarrow \texttt{inl}(v) \quad \sigma[x := v], e_1 \Downarrow v_1}{\sigma, \texttt{case}\ e\ \texttt{of}\ \{x.e_1 \mid y.e_2\} \Downarrow v_1}$$

$$\frac{\sigma, e \Downarrow v}{\sigma, \texttt{inr}(e) \Downarrow \texttt{inr}(v)} \qquad \frac{\sigma, e \Downarrow \texttt{inr}(v) \quad \sigma[y := v], e_2 \Downarrow v_2}{\sigma, \texttt{case}\ e\ \texttt{of}\ \{x.e_1 \mid y.e_2\} \Downarrow v_2}$$

Sum types complicate the issue of how to refer to parts of the input or output. A naive approach would simply be to add `inl` and `inr` as possible path steps, so that the path $1.\texttt{inl}.2$ refers to 42 in the value $(\texttt{inl}(17, 42), 0)$. However, this leads to problems with the definition of source path, since changes to the input might change the structure of the output in ways that invalidate paths involving `inl` or `inr`.

5.2 Higher-order functions and other control abstractions

Modern programming languages increasingly support first-class higher-order functions, either explicitly (as in functional languages such as ML, Haskell, or Scheme, and more recently in object-oriented languages such as C#, Java or Scala), or implicitly via other constructs such as function objects or inner classes (available in older versions of Java).

$$\begin{aligned} e &::= \cdots \mid \lambda x.e \mid e_1\ e_2 \\ v &::= \cdots \mid \langle \lambda x.e, \sigma \rangle \end{aligned}$$

Here, $\langle \lambda x.e, \sigma \rangle$ is a *closure* packaging a function body up with the environment in which it was constructed. We extend the operational semantics as follows (in the standard way):

$$\frac{\sigma, e_1 \Downarrow \langle \lambda x.e, \sigma' \rangle \quad \sigma, e_2 \Downarrow v_2 \quad \sigma'[x := v_2], e \Downarrow v}{\sigma, e_1\ e_2 \Downarrow v} \qquad \frac{}{\sigma, \lambda x.e \Downarrow \langle \lambda x.e, \sigma \rangle}$$

Higher-order functions pose a significant challenge to provenance tracking, because now the control flow (corresponding to the shape of the derivation tree) depends on evaluation: when evaluating a first-class function call, we first evaluate the function part to find the body of the function, which is in general not known until run time. Also, similarly to sums, it is difficult to use paths to refer to "parts" of closure values.

5.3 Collection Types

Now consider an extension to the language to permit simple collections (such as sets, lists, or bags), as in Nested Relational Calculus [10]:

$$\begin{aligned} e &::= \cdots \mid \emptyset \mid \{e\} \mid e_1 \cup e_2 \\ v &::= \cdots \mid \emptyset \mid \{v_1, \ldots, v_n\} \end{aligned}$$

The operational semantics of collection operations can be defined as follows:

$$\frac{}{\sigma, \emptyset \Downarrow \emptyset} \qquad \frac{\sigma, e_1 \Downarrow v_1 \quad \sigma, e_2 \Downarrow v_2}{\sigma, e_1 \cup e_2 \Downarrow v_1 \cup v_2} \qquad \frac{\sigma, e \Downarrow v}{\sigma, \{e\} \Downarrow \{v\}}$$

We could also add a set comprehension operation $\bigcup_{x \in e_0} e$ to obtain an expressive comprehension query language, but the problems we want to discuss do not require this. (Naturally, this would introduce additional complications due to variable binding).

The prospect of using paths to refer to parts of data structures is significantly complicated in the presence of collections, especially unordered collections. For lists, we have similar issues to those for sums. For sets, it is technically possible to refer to set elements via their values, although this can be unwieldy when sets are nested. But for multisets, paths are no longer sufficient to address each part of a value if we view multiset expressions as equal modulo reordering of elements.

As a simple example, consider a multiset expression $\{x - y\} \cup \{z + 1\}$ which evaluates to $\{1, 1\}$. If we want to ask for the provenance of one of the output elements, there is no location scheme for pure multiset values that lets us distinguish between the two copies of '1' in the output. This means that in order to support correct source tracking we need to impose some kind of location structure on multisets. Of course, we can avoid this problem in a simple way, by treating collection values as lists and using integer indices. However, this indexing approach becomes rather complex if we wish to propagate changes from the input to the output, because the paths are not stable with respect to changes that affect the sizes of subcollections.

For example, suppose we have an expression $\{a\} \cup x \cup \{d\}$. If we first evaluate it with $x = \{b\}$ then the result will be $\{a, b, d\}$ where $v[3] = d$. But if we update x to be $\{b, c\}$, then the result is $v' = \{a, b, c, d\}$, where $v'[3] = c$ and $v'[4] = d$. We would like to be able to say that changing x from $\{b\}$ to $\{b, c\}$ did not affect the d value at $v[3]$. Intuitively, the d value in $v[3]$ is the same as the d value at $v'[4]$, but they have different paths. Thus, we need to maintain a partial mapping $\{(1, 1), (3, 4)\}$ relating the paths in v to identical parts of v'. Now consider an expression such as $\{a\} \cup x \cup \{b\} \cup x \cup \{c\}$. If x is changed from $\{d\}$ to $\{d, e\}$, then we need to reindex both the b and c elements, that is, the partial mapping would be $\{(1, 1), (3, 4), (5, 7)\}$.

These examples illustrate that paths based on positional indices are not stable under changes to the input. Although indexing can be made to work using partial mappings, it is notationally heavy, and it seems much cleaner instead to use collections that maintain explicit, unique labels for their elements. These labels can be used in paths, and have two further advantages over indices: we can still treat labeled multisets as equivalent up to reordering, and we do not have to keep track of partial mappings among results. These advantages also hold for lists and sets, for which it is technically possible to use numbers or element values in paths instead of labels. On the other hand, using labels requires generating fresh labels for newly-created collections, which essentially makes the rules for creating collections nondeterministic, which in turn complicates matters for the same reason as other forms of nondeterminism. We are currently investigating this problem.

5.4 Nondeterminism

We model a simple form of nondeterminism by adding a coin-flip expression `flip` whose semantics is as follows:

$$\frac{}{\sigma, \texttt{flip} \Downarrow \texttt{true}} \qquad \frac{}{\sigma, \texttt{flip} \Downarrow \texttt{false}}$$

Obviously, nondeterminism makes it impossible to predict the result of a program or its derivation.

Nondeterminism has little impact on the source-location extraction function: a value constructed by `flip` simply has source $\bot$. However, nondeterminism has interesting consequences for the update-propagation semantics. Basically, the question is how we should deal with changes to computations involving `flip`. Should all coin flips that might affect the result of a computation be re-done? In this case, we would constrain the caching rules to apply only if e does not contain `flip`.

Or should we avoid this as much as possible, to try to preserve the structure of the derivation as much as we can? In this case, we would allow caching to reuse the results of coin-flips stored in the derivation. Only if a subexpression is re-evaluated (e.g. as the result of a control flow change) would new random coin flips be performed.

Either choice may be sensible, depending on the situation. If we want change propagation to simulate what actually happened as much as possible (e.g. to track down the source of an error), we would opt for the first design, reusing as many cached coin flips as possible. If we want change propagation to simulate all possible behaviors of the program, while reusing deterministic subcomputations, then we need the second semantics. If we interpret `flip` probabilistically instead of nondeterministically, there may be additional choices.

5.5 Arrays, references, dynamic allocation, and concurrency

It is interesting to note that most approaches to provenance have focused on relatively high-level languages or abstractions such as database queries, scientific workflows, or operating system calls. In contrast, most work on program slicing [54], and much work on information flow in language-based security [51], has focused on small imperative languages similar to **IMP** extended with features such as arrays, references, and dynamic memory allocation. These features are, of course, still essential parts of programming most real-world systems written in C, C++, or Java, and are also present in many scripting or numerical computation languages used by scientists, such as Python, Matlab or R. Thus, to understand provenance in general, we will need to understand provenance for these features. Since these are exactly the features that tend to make program analysis, typechecking, and debugging difficult, we can predict with some confidence that they will pose challenges for provenance as well.

The approach taken in this paper should provide at least a starting point for understanding provenance in the presence of arrays, references or dynamic memory allocation. Concurrency may also be tackled using operational techniques. However, simply using some kind of derivation as a form of provenance leaves a lot of questions unanswered, especially in the case of concurrency: How can we efficiently record a full operational derivation? Is this even desirable in general, or can we formulate specifications that make it clear that we can make do with less? How can we recover a full operational derivation of a concurrent execution given that each concurrently-executing part only has access to part of the full derivation?

6 Related work

There is some prior work discussing requirements or design philosophies for provenance systems (e.g. [32, 37, 39, 31]). This work, like a great deal of work on provenance, has invoked informal motivations such as that provenance should identify data that are "relevant to", "caused" or "influenced" an output and provide "repeatability", "transparency", or "explanation". There has been little attempt to define these terms carefully or formally with respect to the semantics of the programs or systems being studied. As argued previously in some of our prior work [15, 8, 17], we believe that while these informal motivations are important, they are not enough on their own to explain what provenance is and why it is challenging to define, collect and manage it. There is a danger that different users and implementers may interpret these terms differently, leading to miscommunication and confusion. As was once common in computer security, there are many provenance mechanisms being designed without adequate understanding of the policies that they are meant to satisfy.

Some work on provenance in databases has considered whether certain forms of provenance can be extracted from others, including some negative results. For example, the semiring provenance model of Green et al. [29] can express some other models such as why-provenance and lineage, but cannot express where-provenance and vice versa (as discussed in [24, 16]). However, the design space for techniques to track and manage provenance for data and computations that span databases, workflows, or general-purpose programming languages, remains largely unexplored. Our workshop paper [6] gives one approach to a unified model of provenance for database queries and workflows. A subsequent paper [3] investigates provenance extraction and security issues for a trace model for functional programs. We are interested in developing analogous techniques for database query languages, building on language-integrated query techniques [18].

In a previous paper [15], we identified a connection between dependence in information-flow security and provenance, and developed a new form of provenance based on dependence tracking. Our paper [17] explored connections to programming languages, security, incremental, and bidirectional computation research [1, 5, 25]. More recently, we have investigated foundations for provenance security [14], building on work by Chong [19]. We believe language-based provenance security to be a fruitful area for future work, possibly extending the derivation trace model in this paper.

Our recent work [3, 49] considers traces and trace slicing for a pure, call-by-value calculus with product, sum and recursive types, and recursive functions, using a conventional large-step semantics; paths become unwieldy in this setting, and we adopt an alternative approach based on partial values. Other features such as exceptions, laziness, and first-class continuations (call/cc) pose similar, and possibly greater, challenges from the point of view of provenance. These challenges may require us to abandon the idea of using large-step derivations in favor of the small-step operational techniques typically used for these features [50].

Aside from a few papers on provenance techniques in concurrency calculi [52, 21], the theory of self-explaining computation in the presence of computational effects, concurrency, or laziness is unexplored. General approaches to the denotational or operational semantics of effects [36, 34] provide an intriguing starting point for the study of provenance in the presence of effects. Ideas from concurrency theory, particularly Winskel's *event structures* [58], may be a good place to start in understanding the meaning of provenance in concurrent or distributed settings.

7 Conclusions

To date, research on provenance has focused on particular classes of systems or computational models, such as databases, workflow management systems or operating systems. Real scientific data and processing pipelines are typically not confined to a single kind of system but instead use a combination of these systems as well as ad hoc programs written in general-purpose or scripting languages that glue these different systems together. We therefore argue here that provenance needs to be understood for general-purpose programming languages.

In this paper, we have proposed a simple (perhaps too simple) model of provenance in general-purpose programming language: the provenance trace of a computation is simply a full operational derivation, i.e. a "proof" that the program executed and produced a given value. This approach has both advantages and drawbacks. It seems reasonable to expect that we can extract any other form of provenance from such a trace: even in the presence of nondeterminism, the trace records all inputs, outputs and intermediate choices. Thus, we can extract other forms of provenance, such as source location information, as well as adapting traces to changes to the input. However, this generality (at least, if interpreted naively) comes at a high cost: the memory and processing overhead of storing such full traces for nontrivial programs appears prohibitive, strongly motivating compression or slicing techniques that can mitigate this cost while still providing detailed provenance. Another possible drawback, compared to, for example, the elegant semiring framework used in relational databases [29, 24], is the absence of strong semantic properties that can be used to optimize programs in the presence of provenance.

Nevertheless, our contribution helps to frame the problem of provenance management in general-purpose languages, by proposing an idealized "most general" form of provenance that can be used to define and compare other, more practical techniques. Many other questions remain to be investigated in developing a theory of self-explaining computation, including:

– Can we compress or slice the full derivation trace efficiently enough to make it a practical approach? If not, what are the limits of efficient provenance for general-purpose programs?
– How can we extend derivation traces to handle complex features such as concurrency, side-effects or collections?
– Can we identify intermediate forms of provenance that retain a high degree of generality while remaining efficiently implementable?
– Can we develop an appropriate compositional model of provenance building on denotational semantics (and admitting standard program equivalences)?

Acknowledgments Effort sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-13-1-3006. The U.S Government is authorized to reproduce and distribute reprints for Governmental purpose notwithstanding any copyright notation thereon. Cheney is supported by a Royal Society University Research Fellowship, by the EU FP7

DIACHRON project, and EPSRC grant EP/K020218/1. Parts of this research were done while Acar and Perera were at Max-Planck Institute for Software Systems, Kaiserslautern, Germany, and while Perera was a PhD student at the University of Birmingham.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL*, pages 147–160. ACM Press, 1999.
2. M. Abadi, B. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *ICFP*, pages 83–91. ACM Press, 1996.
3. U. A. Acar, A. Ahmed, J. Cheney, and R. Perera. A core calculus for provenance. In P. Degano and J. D. Guttman, editors, *Proceedings of the 1st International Conference on Principles of Security and Trust (POST)*, volume 7215 of *LNCS*, pages 410–429. Springer-Verlag, 2012.
4. U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Prog. Lang. Sys.*, 32(1):3:1–3:53, 2009.
5. U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, 2006.
6. U. A. Acar, P. Buneman, J. Cheney, N. Kwasnikowska, J. Van den Bussche, and S. Vansummeren. A graph model of data and workflow provenance. In *TAPP*, 2010. Online informal proceedings: `http://www.usenix.org/event/tapp10`.
7. S. Bowers, T. M. McPhillips, S. Riddle, M. K. Anand, and B. Ludäescher. Kepler/pPOD: Scientific workflow and provenance support for assembling the tree of life. In Freire et al. [26], pages 70–77.
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. Springer, 2001.
10. P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.
11. S. Carey and G. Rogow. UAL shares fall as old story surfaces online. Wall Street Journal, September 2008. `http://online.wsj.com/article/SB122088673-738010213.html`.
12. J. Cheney. Program slicing and data provenance. *IEEE Data Engineering Bulletin*, pages 22–28, December 2007. Invited paper.
13. J. Cheney. Causality and the semantics of provenance. In *Proceedings of the 2010 Workshop on Developments in Computational Models*, 2010.
14. J. Cheney. A formal framework for provenance security. In *CSF*, pages 281–293. IEEE, 2011.
15. J. Cheney, A. Ahmed, and U. A. Acar. Provenance as dependency analysis. *Mathematical Structures in Computer Science*, 21(6):1301–1337, 2011.
16. J. Cheney, L. Chiticariu, and W. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.
17. J. Cheney, S. Chong, N. Foster, M. Seltzer, and S. Vansummeren. Provenance: A future history. In *OOPSLA Companion (Onward! 2009)*, pages 957–964, 2009.

18. J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *ICFP*, 2013. To appear.
19. S. Chong. Towards semantics for provenance security. In *Workshop on the Theory and Practice of Provenance*, 2009. Informal online proceedings: `http://www.usenix.org/events/tapp09/`.
20. Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.
21. M. Dezani-Ciancaglini, R. Horne, and V. Sassone. Tracing where and who provenance in linked data: A calculus. *Theor. Comput. Sci.*, 464:113–129, 2012.
22. P. Dourish. *Computers and Design in Context*, chapter Accounting for System Behaviour: Representation, Reflection and Resourceful Action, pages 145–170. MIT Press, 1997.
23. I. Foster, J. Vockler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *SSDBM*, pages 1–10, July 2002.
24. J. N. Foster, T. J. Green, and V. Tannen. Annotated XML: queries and provenance. In *PODS*, pages 271–280, 2008.
25. J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007.
26. J. Freire, D. Koop, and L. Moreau, editors. *Second International Provenance and Annotation Workshop*, volume 5272 of *LNCS*. Springer, 2008.
27. Y. Gil, J. Cheney, P. Groth, O. Hartig, S. Miles, L. Moreau, P. P. da Silva, S. Coppens, D. Garijo, J. M. Gomez, P. Missier, J. Myers, S. Sahoo, and J. Zhao. Provenance XG final report, December 2010. http://www.w3.org/2005/Incubator/prov/XGR-prov-20101214/.
28. D. I. Good. The foundations of computer security: we need some. http://www.ieee-security.org/CSFWweb/goodessay.html, 1986.
29. T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40. ACM, 2007.
30. G. Greenwald and E. MacAskill. NSA Prism program taps in to user data of Apple, Google and others. The Guardian, June 2013. http://www.guardian.co.uk/world/2013/jun/06/us-tech-giants-nsa-data.
31. P. Groth, Y. Gil, J. Cheney, and S. Miles. Requirements for provenance on the web. *International Journal of Digital Curation*, 7(1):39–56, 2012.
32. P. T. Groth, S. Miles, and S. Munroe. Principles of high quality documentation for provenance: A philosophical discussion. In *IPAW*, pages 278–286, 2006.
33. G. Kahn. Natural semantics. In *STACS*, pages 22–39, 1987.
34. O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *ICFP*, 2013. To appear.
35. L. Lamport. State the problem before describing the solution. *SIGSOFT Softw. Eng. Notes*, 3:26–26, January 1978.
36. P. B. Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantic Structures in Computation*. Springer, 2004.
37. C. A. Lynch. When documents deceive: trust and provenance as new factors for information retrieval in a tangled web. *J. Am. Soc. Inf. Sci. Technol.*, 52(1):12–17, 2001.
38. D. L. McGuinness and P. Pinheiro da Silva. Explaining answers from the semantic web: the inference web approach. *Web Semant.*, 1:397–413, October 2004.

39. S. Miles, P. Groth, M. Branco, and L. Moreau. The requirements of using provenance in e-science experiments. *Journal of Grid Computing*, 5:1–25, 2007. 10.1007/s10723-006-9055-3.
40. S. Miles, P. T. Groth, S. Munroe, S. Jiang, T. Assandri, and L. Moreau. Extracting causal graphs from an open provenance data model. *Concurrency and Computation: Practice and Experience*, 20(5):577–586, 2008.
41. S. Miles, S. C. Wong, W. Fang, P. T. Groth, K.-P. Zauner, and L. Moreau. Provenance-based validation of e-science experiments. *J. Web Sem.*, 5(1):28–38, 2007.
42. G. Miller. A scientist's nightmare: Software problem leads to five retractions. *Science*, 314(5807):1856–1857, December 2006.
43. P. Missier, K. Belhajjame, J. Zhao, M. Roos, and C. A. Goble. Data lineage model for Taverna workflows with lightweight annotation requirements. In Freire et al. [26], pages 17–30.
44. L. Moreau. The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2(2-3):99–241, 2010.
45. L. Moreau. Provenance-based reproducibility in the semantic web. *J. Web Sem.*, 9(2):202–221, 2011.
46. L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. T. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. G. Stephan, and J. V. den Bussche. The open provenance model core specification (v1.1). *Future Generation Comp. Syst.*, 27(6):743–756, 2011.
47. L. Moreau and P. Missier (eds.). PROV-DM: The PROV data model. W3C Recommendation, April 2013. http://www.w3.org/TR/2013/REC-prov-dm-20130430/.
48. K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference*, pages 43–56. USENIX, June 2006.
49. R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. In *ICFP*, pages 365–376. ACM, 2012.
50. G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
51. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
52. I. Souilah, A. Francalanza, and V. Sassone. A formal model of provenance in distributed systems. In *Workshop on the Theory and Practice of Provenance*, 2009.
53. J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.* MIT Press, 1981.
54. F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.
55. S. Varghese. UK government gets bitten by Microsoft Word. Sydney Morning Herald, July 2003. `http://www.smh.com.au/articles/2003/07/02/-1056825430340.html`.
56. Y. R. Wang and S. E. Madnick. A polygen model for heterogeneous database systems: The source tagging perspective. In *VLDB*, pages 519–538, 1990.
57. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.
58. G. Winskel. Events, causality and symmetry. *Comput. J.*, 54(1):42–57, 2011.
59. A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, pages 91–102, 1997.
60. Y. Zhao, M. Wilde, and I. T. Foster. Applying the virtual data provenance model. In *IPAW*, pages 148–161, 2006.