THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

A core calculus for provenance

Citation for published version:
Acar, UA, Ahmed, A, Cheney, J & Perera, R 2013, 'A core calculus for provenance', Journal of Computer Security, vol. 21, no. 6, pp. 919-969. https://doi.org/10.3233/JCS-130487

Digital Object Identifier (DOI):
10.3233/JCS-130487

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Journal of Computer Security

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 23. Sep. 2019

arXiv:1310.6299v2 [cs.PL] 3 Jan 2014

A Core Calculus for Provenance

Umut A. Acar Amal Ahmed James Cheney Roly Perera

January 6, 2014

Abstract

Provenance is an increasing concern due to the ongoing revolution in sharing and processing scientific data on the Web and in other computer systems. It is proposed that many computer systems will need to become provenance-aware in order to provide satisfactory accountability, reproducibility, and trust for scientific or other high-value data. To date, there is not a consensus concerning appropriate formal models or security properties for provenance. In previous work, we introduced a formal framework for provenance security and proposed formal definitions of properties called disclosure and obfuscation.

In this article, we study refined notions of positive and negative disclosure and obfuscation in a concrete setting, that of a general-purpose programing language. Previous models of provenance have focused on special-purpose languages such as workflows and database queries. We consider a higher-order, functional language with sums, products, and recursive types and functions, and equip it with a tracing semantics in which traces themselves can be replayed as computations. We present an annotation-propagation framework that supports many provenance views over traces, including standard forms of provenance studied previously. We investigate some relationships among provenance views and develop some partial solutions to the disclosure and obfuscation problems, including correct algorithms for disclosure and positive obfuscation based on trace slicing.

1 Introduction

Provenance, or meta-information about the origin, history, or derivation of an object, is now recognized as a central challenge in establishing trust and providing security in computer systems, particularly on the Web. Essentially, provenance management involves instrumenting a system with detailed monitoring or logging of auditable records that help explain how results depend on inputs or other (sometimes untrustworthy) sources. The security and privacy ramifications of provenance must be understood in order to safely meet the needs of users that desire provenance without introducing new security vulnerabilities or compromising the confidentiality of other users.

The lack of adequate provenance information can cause (and has caused) major problems, which we call *provenance failures* [19]. Essentially, a provenance failure can arise either from failure to *disclose* some key provenance information to users, or from failure to *obfuscate* some sensitive provenance information. As an example of failure to disclose provenance, in 2008 an undated, years-out-of-date story about United Airlines' 2002 near-bankruptcy was mistakenly put on Google News' main page, causing investors to panic about its financial stability, which in turn led to a significant decrease in its share price over the course of a few hours [12]. Another example is the 'Climategate' controversy [42], in which climate scientists were embarrassed (and widely criticized by climate change skeptics) when private emails that suggested poor data analysis practice were leaked. . As an example of failure to obfuscate, in 2003 a Word document about British intelligence prior to the invasion of Iraq was published with supposedly secret contributors' identities logged in its change history [47], revealing the influence of political advisors on the report.

Obfuscation is obviously closely related to traditional security concerns, such as confidentiality and anonymity. Disclosure is, in our view, also a security property, linked to the traditional security concern of availability. In securing provenance, we seek to disclose some important provenance information while keeping other aspects of provenance confidential. If all we cared about was obfuscation, then security would be easy to achieve by simply not providing any provenance. The tension between the two goals of disclosure and obfuscation makes the analysis of security for provenance a more challenging problem.

Provenance has predominantly been studied in the context of scientific computation and databases. A number of forms of provenance have been proposed for different computational models, including *why* and *where* provenance [11], *how*-provenance [28], and *dependency* provenance [17] in databases. In other settings, a variety of ad hoc techniques have been proposed, largely based on instrumenting various systems to record a graph diagramming the procedure calls or dependencies among data and processes [38, 8, 44]. However, almost all of this work assumes a cooperative setting in which users are not intentionally trying to subvert or forge provenance information. When the accuracy of information used by day traders, validity and public acceptance of scientific results, and independence of intelligence reports from political influence depends on provenance, there is a great deal at stake, so it is important to develop foundations for correctness and security of provenance in the face of attacks.

Although a wide variety of models of provenance have been studied in different settings, there has been relatively little progress on developing a general understanding of provenance. By analogy with Abadi, Banerjee, Heintze and Riecke's *core calculus of dependency* [1], which elucidated the common ideas underlying different techniques such as information flow security, program slicing, and binding-time analysis, this article introduces a core calculus for provenance: that is, a calculus that illustrates and unifies the key ideas underlying a range of provenance techniques, including tracing, annotation-propagation, and connections to program slicing. Our main application of this framework is to explore the implications of the general definitions of disclosure and obfuscation introduced in our prior work, but we hope that our approach will also be useful for studying other aspects of provenance.

Prior work on provenance and security. Despite its apparent importance, there has been relatively little work on formal foundations of provenance, and work on provenance security has only begun to appear over the last five years.

Our previous work [17] appears to have been the first to explicitly relate information-flow security to a form of provenance, called *dependency provenance*. Provenance has been studied in language-based security by Cirillo et al. [21], who developed a form of authorization logic with notions of provenance for understanding information flow among concurrently executing objects, and by Swamy et al. [46, 45], who developed mechanisms for dependency-provenance tracking in a dependently-typed secure programming language called Fable. Both projects focus on specifying and enforcing security policies involving provenance tracking alongside many other concerns, and not on defining provenance semantics or extraction techniques. Work on secure auditing [34, 29] and expressive programming languages for security [45] is also related, but this work focuses on explicitly manipulating proofs of authorization or evidence about protocol or program runs rather than automatically deriving or securing provenance information in its own right.

There is also some work directly addressing security for provenance [20, 32, 15, 23, 6, 24]. Chong [20] gave (to our knowledge) the first candidate formal definitions of *data security* and *provenance security* using a trace semantics, based in part on earlier, unpublished work of ours on traces and provenance [16]. Hasan et al. [32] study security techniques for ensuring the integrity of a document that changes over time along with its provenance records. Davidson et al. [23] studied a notion of privacy for provenance in scientific workflows, focusing on complexity lower bounds. In their approach, the definition of privacy essentially says that for unknown components in a workflow (i.e. a simple dataflow diagram), an attacker should not be able to learn functional behavior; for example, should not be able to narrow down the possible output values for any input to less than a parameter k. Cheney [15] gave an abstract framework for provenance, proposed definitions of properties called *obfuscation* and *disclosure*, and discussed algorithms and complexity results for instances of this framework including finite automata, workflows, and the semiring model of database provenance [28]. Zhang et al. [50] develop tamper-detection techniques for provenance in databases. Blaustein et al. [6] studied the problem of rewriting provenance graphs to hide information while still satisfying some plausibility constraints. Dey et al. [24] studied provenance publishing policies, aimed at giving users greater control over what information is shown and hidden. They developed a system called ProPub equipped with well-defined publishing and hiding operators, along with constraints that such policies should satisfy. (In this respect, Dey et al.'s publishing operators, and work on "provenance views" [35] can be seen retroactively as addressing disclosure requirements subject to additional conciseness constraints.)

More recently, Lyle and Martin [36] gave a detailed comparative survey of topics in provenance and in security, pointing out many parallel developments, and Martin et al. [37] advocate study of provenance considered as a security control. Some other topics in security, such as non-repudiation [43], plausible deniability or differential privacy [26], also appear analogous to our disclosure and obfuscation properties, and this connection could be explored.

In this article, we build on prior work on provenance security by studying the disclosure and obfuscation properties of different forms of provenance in the context of a higher-order, pure, functional language. To illustrate what we mean by provenance, we present examples of programming with three different forms of provenance in Transparent ML (TML), a prototype implementation of the ideas of this article.

To ease exposition, we present the examples in terms of a hypothetical ML-like toplevel loop extended with labeled values, first-class traces, a type (Γ, τ) `trace` that consists of traces returning type τ evaluated in label context Γ, and with various functions that extract different forms of provenance from traces. The tracing and extraction features are formalized later in the article, and our prototype supports these examples, as well as the disclosure and obfuscation slicing algorithms presented later in the paper. The Slicer and LambdaCalc tools of Perera et al. [40, 39] employ similar ideas, and have been run on larger examples, but focus on slicing as a debugging and program understanding technique and does not yet support provenance extraction or disclosure and obfuscation slicing. Developing a unified and mature implementation supporting all of these ideas is left for future work; Perera et al. [40, 39] should be consulted for further implementation details.

1.1 Examples

Where-provenance. Where-provenance [11, 10] identifies at most one source location from which a part of the output was copied. For example, consider the following TML session:

```
- f [(1,2), (4,3), (5,6)];
val it = [(5,6), (3,4), (1,2)]
```

Without access to the source code, one can guess that f is doing something like

$$reverse \circ (map\ (\lambda(x, y).\mathtt{if}\ x < y\ \mathtt{then}\ (x, y)\ \mathtt{else}\ (y, x)))$$

However, by providing where-provenance information, the system can explain whether the numbers in the result were copied from the input or constructed in some other way:

```
- trace (f [(1@L1,2@L2),(4@L3,3@L4),(5@L5,6@L6)]);
it = <trace> : ({L1:int,...}, (int*int) list) trace
- where it;
val it = [(5@L5,6), (3@L4,4), (1@L1,2)]
```

This shows that f contrives to copy the first elements of the returned pairs but construct the second components.

Dependency provenance. Dependency provenance [17] is an approach that tracks a set of all source locations on which a result depends. For example, if we have:

```
- g [(1,2,3), (4,5,6)];
val it = [6,6] : int list
```

we again cannot tell much about what g does. By tracing and asking for dependency provenance, we can see:

```
- trace (g [(1@L1,2@L2,3@L3),(4@L4,5@L5,6@L6)]);
val it = <trace> : ({L1:int,...}, int list) trace
- dependency it;
val it = [6@{L1,L2,L3}, 6@{L1,L2,L3}]
```

This suggests that g is computing both elements of the result from the first triple and returning the result twice, without examining the rest of the list. We can confirm this as follows:

```
- trace (g ((1@L1,2@L2,3@L3)::[]@L));
val it = <trace> : ({L1:int,...}, int list) trace
- dependency it;
val it = [6@{L1,L2,L3}]
```

The fact that L does not appear in the output confirms that g does not look further into the list. While it appears that g may be computing 6 from $1, 2, 3$ by adding them together, the exact process by which g computes 6 from $1, 2, 3$ is not explicit in the dependency annotations; they are also consistent with the hypothesis that g multiplies $1, 2, 3$ together to compute 6 or even that g simply examines $1, 2, 3$ and then returns the constant 6.

Expression provenance. A third common form of provenance is an expression graph or tree that shows how a value was computed by primitive operations. For example, consider:

```
- (h 3, h 4, h 5)
val it = (6,24,120);
```

We might conjecture that h is actually the factorial function. By tracing h and extracting expression provenance, we can confirm this guess (at least for the given inputs):

```
- trace (h (4@L));
val it = <trace> : ({L:int}, int) trace
- expression it;
val it = 24@{L * (L-1) * (L-2) * (L-3)  * 1}
```

In this case, both where-provenance and dependency provenance would be uninformative since the result is not copied from, and obviously depends on, the input.

This kind of provenance is used extensively in *workflow* systems often used in e-science [33], where the main program is a high-level process coordinating a number of external (and often concurrent) program or RPC calls, for example, image-processing steps or bulk data transformations, which we could model by adding primitive image-processing operations and types to our language. Thus, even though the above examples use fine-grained primitive operations, this model is also useful for coarse-grained provenance-tracking.

A running example. Figure 1 graphically illustrates these three forms of provenance on a single example: a simple function mapped over a list. This corresponds to the following TML sessions:

```
- val y = 2@L;
- fun f x = if x = y then y else x+1;
- val xs = [1@L1,2@L2,3@L3];
- val t = trace (map f xs);
val t = <trace> : ({L1:int,L2:int,L3:int}, int list) trace
- where t;
val it = [2@{},2@{L},4@{}]
- dependency t;
val it = [2@{L1,L},2@{L2,L},4@{L3,L}];
- expression t;
val it = [2@{L1+1},2@{L},4@{L3+1}];
```

Note that this illustrates much of the power of TML, including higher-order, recursive functions and sum, product and recursive types. We use this as a running example throughout the paper.

Provenance security. The three models of provenance above represent useful forms of provenance that might increase users' trust or confidence that they understand the results of a program. However, if the underlying data, or the structure of the computation, is sensitive, then making this information available may lead to inadvertent vulnerabilities, by making it possible for users to infer sensitive information that they cannot observe directly. This is a particular problem if we wish to disclose part of the result of a program, and provenance that justifies part of the result, while keeping other parts of the program's execution, input, or output confidential.

As a simple example, consider a program `if` $x \neq 1$ `then` (y, y) `else` (z, w). Suppose we wish to disclose some information about the computation to an untrusted recipient Alice (not necessarily a malicious attacker), that none of

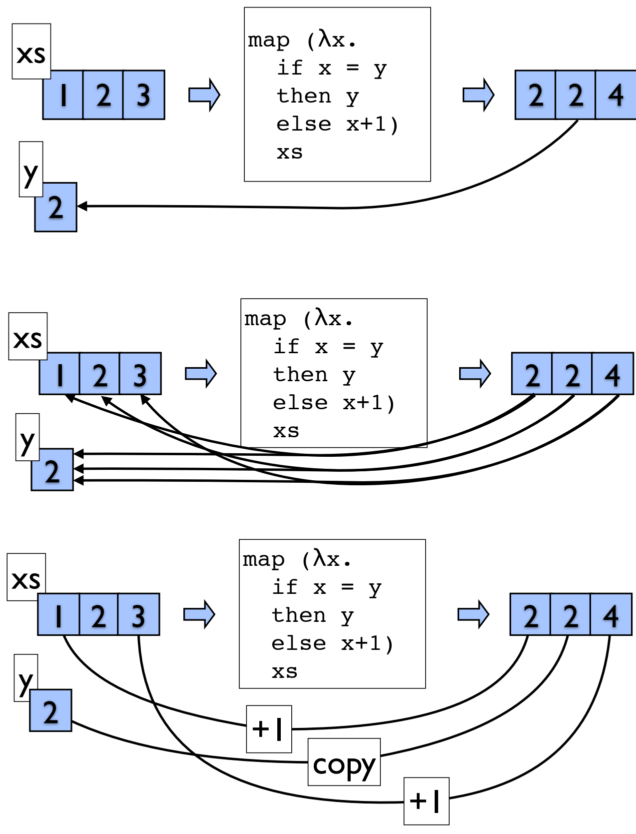

Figure 1: Illustration of various forms of provenance

x, y are visible to Alice, that z, w are visible to Alice, and only the value of x is confidential. If z and w happen to both equal a common value, say 42, then the result is $(42, 42)$. It is safe to disclose this result to Alice, because (without inspecting x or y) she cannot be certain whether $x = 1$, since there are two scenarios consistent with the result: $x = 1$ and $y = 42$ or $x \neq 1$ and y arbitrary. However, any of the above forms of provenance make it possible to distinguish which branch was taken because the two different copies of 42 in z and w will have different provenance. Thus, if the provenance information is released then a principal can infer that the second branch was taken, and hence, $x = 1$. In technical terms, we cannot *disclose* any of the above forms of provenance for the result while *obfuscating* the fact that $x = 1$.

To study these problems systematically, we introduce a single, general model of provenance that can be instantiated in different ways to obtain the other models mentioned above (among many others). Essentially, our approach is to record a detailed trace of evaluation, whose structure corresponds closely to that of a large-step operational semantics derivation. Other forms of provenance can then be extracted by traversing the trace, and the trace itself can be viewed as a form of provenance. Of course, naively recording such a detailed trace may be prohibitively expensive, and in this article we are not advocating that such traces be explicitly constructed in practical systems, only that they are useful as a formalism for understanding different forms of provenance and their security properties.

1.2 Summary

Contributions. In this article, we build on, and refine, the provenance security framework previously introduced by Cheney [15]. We introduce a core language with replayable execution traces for a call-by-value, higher-order functional language, and make the following technical contributions:

- Refined definitions of obfuscation and disclosure (Sec. 2).
- A core calculus defining traced execution for a pure functional programming language (Sec. 3).
- A generic provenance extraction framework that includes several previously-studied forms of provenance as instances (Sec. 4).
- An analysis of disclosure and obfuscation guarantees provided by different forms of provenance, including techniques based on slicing execution traces (Sec. 5).

This article is a revised and expanded version of a conference paper [3]. Compared with the conference paper, this article includes detailed proofs, a more complete discussion of related work (including work published recently that was not covered in the conference paper), and additional examples and discussion of technical points. In addition, we encountered a problem with proving correctness of the disclosure slicing algorithm proposed in [3]; specifically, Lemma 2 in the conference version had a subtle problem, which we avoid through a reformulation of the disclosure slicing algorithm.

This article is also closely related to work on using traces for program slicing [40] published in ICFP 2012. The two papers present different aspects of a single research project; the trace model and some aspects of the slicing algorithms are closely related. However, the two papers make distinct contributions, and the system in the ICFP paper incorporates simplifications that are appropriate pragmatic choices for its application area (program slicing) but not appropriate for security analysis. Because of these differences we have chosen not to attempt to develop a unified presentation or implementation, to ensure that focus in this article remains on provenance security. We summarize the key differences below.

- In the ICFP paper, traces and slicing are defined in terms of an ad hoc semantics over partial values, and justified by a Galois connection between them. Here, we instead define slices for disclosure and obfuscation in terms of a standard operational semantics. One important consequence of these different choices is that unique minimal disclosure slices do not exist, whereas unique minimal backwards slices do exist in the ICFP paper.
- The slicing algorithms in the ICFP paper differ from those given here in certain technical details: specifically, we use value patterns and the $\eqsim_p$ equivalence relation instead of partial values, and we use $\Diamond$-patterns to support slicing of primitive operations instead of tagging primitive operations with the values of their inputs in traces.

- The ICFP paper did not present provenance extraction or explore the connection to provenance security that is the focus of this paper.
- The ICFP paper presents work on program slicing, differential slicing, and implementation techniques, topics that are beyond the scope of this article.

Outline. Section 2 briefly recapitulates the framework introduced by Cheney [15] and refines some definitions. We present the (standard) syntax and tracing semantics of TML in Section 3. In Section 4 we introduce a framework for querying and extracting provenance views from traces, including the three models discussed above. Section 5 presents our main results about disclosure, obfuscation, and trace slicing. Section 6 presents related work and Section 7 concludes.

2 Background

We recapitulate the main components of the provenance security framework of Cheney [15]. The framework assumes a given set of *abstract traces* $\mathcal{T}$, together with a collection $\mathcal{Q}$ of possible *trace queries* $Q : \mathcal{T} \to \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$ is the set of Boolean truth values. These represent properties of traces that the system designer may want to protect or that legitimate users or attackers of the system may want to learn. In the previous paper, we considered refinements to take into account the knowledge of the principals about the possible system behaviors. In this article, we consider a single principal and assume that all traces $\mathcal{T}$ are considered possible, for simplicity.

Fix a set Ω of the possible *provenance views*, and a function $P : \mathcal{T} \to \Omega$ mapping each trace to a provenance view of the trace. We may write $(\Omega, P : \mathcal{T} \to \Omega)$ or just (Ω, P) for a provenance view. Also, we typically write $q : \Omega \to \mathbb{B}$ for a provenance query, that is, a query on a provenance view.

Given this framework, we proposed the following definitions:

Definition 2.1 (Disclosure). *A query $Q : \mathcal{T} \to \mathbb{B}$ is* disclosed *by a provenance view (Ω, P) if for every $t, t' \in \mathcal{T}$, if $P(t) = P(t')$ then $Q(t) = Q(t')$.*

In other words, disclosure means that there can be no traces t, t' that have the same provenance view but where one satisfies the query and the other does not.

Definition 2.2 (Obfuscation). *A query $Q : \mathcal{T} \to \mathbb{B}$ is* obfuscated *by a provenance view (Ω, P) if for every t in $\mathcal{T}$, there exists $t' \in \mathcal{T}$ such that $P(t) = P(t')$ and $Q(t) \neq Q(t')$.*

Thus, obfuscation is not exactly the opposite of disclosure; instead, it means that for every trace there is another trace with the same provenance view but different Q-value. This means that a principal that has access to the provenance view but not the trace cannot be certain whether or not Q is satisfied by the underlying trace.

In the previous paper, we gave several examples of instances of this framework. Here, for illustration, we just review one such instance, given by regular languages and finite automata.

Example 2.3 (Strings as traces). *Consider the regular sublanguage of $\{a, b\}^+$ consisting of nonempty strings; these can be viewed as traces of an automaton or other sequential process. Some views of the traces of the automaton include a transducer T_1 replacing each symbol with a, a transducer T_2 that deletes all of the as, and a transducer T_3 that deletes alternating symbols. A query over these traces can test whether the number of bs is even; this is obfuscated by T_1, disclosed by T_2, and neither fully obfuscated nor fully disclosed by T_3.*

When finite automata are used for queries and transducers for provenance views, we showed that disclosure is decidable for all queries and views and that obfuscation is decidable for all queries and views whose range is finite. It is unknown whether obfuscation is decidable in the general case.

The definitions above turn out to be too strong for our purposes; in this paper we will also consider some weaker versions of disclosure and obfuscation.

Definition 2.4. *A query* $Q : \mathcal{T} \to \mathbb{B}$ *is* positively disclosed *by provenance view* (Ω, P) *via query* $q : \Omega \to \mathbb{B}$ *if for every* t*, if* $q(P(t)) = 1$ *then* $Q(t) = 1$.

A query $Q : \mathcal{T} \to \mathbb{B}$ *is* negatively disclosed *by provenance view* (Ω, P) *via query* $q : \Omega \to \mathbb{B}$ *if for every* t*, if* $q(P(t)) = 0$ *then* $Q(t) = 0$.

In other words, positive disclosure means that there is a query q on the provenance that safely overapproximates Q on the underlying trace. If $q(P(t))$ holds then we know $Q(t)$ holds but otherwise we may not learn anything about t. Dually, negative disclosure means that if $q(P(t))$ is false then we know $Q(t)$ is also false, but otherwise learn nothing.

Example 2.5. *Suppose* $\mathcal{T} = \Omega = \{a, b\}^*$*. Define query* $Q(t)$ *to be true if and only if* t *is not of the form* $u \cdot abab \cdot v$ *for strings* u, v*, and let* $P(a_1 a_2 \cdots a_n) = a_2 a_4 \cdots a_{\lfloor \frac{n}{2} \rfloor}$*, the function that deletes alternate letters of its argument. Finally, let* $q(t)$ *be a query on* Ω *that is true if and only if* t *has no substrings of the form* aa *or* bb*. Then* Q *is positively disclosed by* P *via* q*, for if* $P(t)$ *has no* aa *or* bb *substring, then* t *can have no* $abab$ *substring. However,* Q *is not negatively disclosed by* P *(for any* q'*), because, for example,* $P(abab) = bb = P(bbbb)$*.*

Definition 2.6. *A query* $Q : \mathcal{T} \to \mathbb{B}$ *is* positively obfuscated *by* (Ω, P) *if for every* t *satisfying* $Q(t) = 1$ *there exists a trace* t' *such that* $Q(t') = 0$ *and* $P(t) = P(t')$*.*

A query $Q : \mathcal{T} \to \mathbb{B}$ *is* negatively obfuscated *by* (Ω, P) *if for every* t *satisfying* $Q(t) = 0$ *there exists a trace* t' *such that* $Q(t') = 1$ *and* $P(t) = P(t')$*.*

In other words, positive obfuscation means that the provenance never reveals that Q holds of the trace, but it may reveal that Q fails. This weaker notion is useful for asserting that sensitive data is protected: if the sensitive data is not present in the trace then it is harmless to reveal this, but if the sensitive data is present then the provenance should hide enough information to make its presence uncertain. Dually, negative obfuscation means that the provenance view does not reveal $\neg Q$.

Example 2.7. *Again suppose* $\mathcal{T} = \Omega = \{a, b\}^*$*. Define* $Q(t)$ *to be true if and only if the number of* a *symbols in* t *is odd and false otherwise. Define* $P(t)$ *to be* t *with all* a*s replaced by* b*s. (In other words,* $P(t) = b^{|t|}$*, a string of* b*s of the same length as* t*.)* P *positively obfuscates* Q *because if* $Q(t)$ *holds, then* t *is a nonempty string with an odd number of* a*s, and we can form* t' *such that* $P(t) = P(t')$ *by replacing one of the* a*s of* t *with a* b*. However,* P *does not negatively obfuscate* Q *because* $P(\epsilon) = \epsilon$ *and there is no other string* t' *with* $Q(t') = true$ *and* $P(t') = \epsilon$*.*

Proposition 2.8. *If* P *both positively discloses and negatively discloses* Q *via* q*, then* P *discloses* Q*. Similarly, if* P *both positively and negatively obfuscates* Q *then* P *obfuscates* Q*.*

Proof. For the first part, suppose P discloses Q positively and negatively via q. Let $t, t' \in \mathcal{T}$ be given where $P(t) = P(t')$. If $q(P(t))$ holds then $q(P(t'))$ holds so $Q(t) = 1 = Q(t')$. If $q(P(t)) = 0$ then $q(P(t')) = 0$ and so $Q(t) = 0 = Q(t')$.

The argument for obfuscation is straightforward. □

Remark 2.9. *It may seem surprising that positive and negative disclosure specify a provenance query* q *while full disclosure does not specify such a parameter. If full disclosure holds, then there is no need to mention the provenance query* q *that answers trace queries over* Ω*, since it is the characteristic function of* $\{P(t) \mid Q(t) = 1\}$*. However, if we leave out (or existentially quantify over) the provenance query* q *in positive or negative disclosure, then both definitions become trivial, since positive disclosure always holds for* $q(x) = 0$ *and negative disclosure always holds for* $q(x) = 1$*. Moreover, we want to be able to decompose proving full disclosure into proving positive and negative disclosure, but the argument given above requires that the positive and negative disclosure hold with respect to the same* q*.*

We now proceed to instantiate the framework with traces generated by a richer language, with corresponding notions of trace query and provenance view.

$$
\begin{array}{llcl}
\text{Types} & \tau & ::= & b \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \mu\alpha.\tau \mid \alpha \\
\text{Type contexts} & \Gamma & ::= & [x_1 : \tau_1, \ldots, x_n : \tau_n] \\
\text{Code pointers} & \kappa & ::= & f(x).e \\
\text{Matches} & m & ::= & \{\mathtt{inl}(x_1).e_1; \mathtt{inr}(x_2).e_2\} \\
\text{Values} & v & ::= & c \mid (v_1, v_2) \mid \mathtt{inl}(v) \mid \mathtt{inr}(v) \mid \langle \kappa, \gamma \rangle \mid \mathtt{roll}(v) \\
\text{Environments} & \gamma & ::= & [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \\
\text{Expressions} & e & ::= & c \mid x \mid \oplus(\overline{e}) \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \\
 & & \mid & (e_1, e_2) \mid \mathtt{fst}(e) \mid \mathtt{snd}(e) \\
 & & \mid & \mathtt{inl}(e) \mid \mathtt{inr}(e) \mid \mathtt{case}\ e\ \mathtt{of}\ m \\
 & & \mid & \mathtt{fun}\ \kappa \mid (e\ e') \\
 & & \mid & \mathtt{roll}(e) \mid \mathtt{unroll}(e) \\
\text{Traces} & T & ::= & c \mid x \mid \oplus(\overline{T}) \mid \mathtt{let}\ x = T_1\ \mathtt{in}\ T_2 \\
 & & \mid & (T_1, T_2) \mid \mathtt{fst}(T) \mid \mathtt{snd}(T) \\
 & & \mid & \mathtt{inl}(T) \mid \mathtt{inr}(T) \mid \mathtt{case}\ T \triangleright_{\mathtt{inl}} x.T_1 \mid \mathtt{case}\ T \triangleright_{\mathtt{inr}} x.T_2 \\
 & & \mid & \mathtt{fun}\ \kappa \mid (T_1\ T_2) \triangleright_{\kappa,\Gamma} f(x).T \\
 & & \mid & \mathtt{roll}(T) \mid \mathtt{unroll}(T)
\end{array}
$$

Figure 2: Abstract syntax of Core TML.

3 Core Language

We will develop a core language for provenance based on a standard, typed, call-by-value, pure language, called Transparent ML, or TML. For the purpose of this article, we focus on terminating runs of pure computations. We only consider terminating runs since otherwise there is no trace or end result to analyze; the question of how to deal with provenance in nonterminating or effectful programs is interesting, but left for future work. Allowing for effects or moving to a small-step semantics each seem likely to complicate the trace semantics (and subsequent analysis) considerably.

The syntax of TML types, expressions, and other syntactic classes is shown in Figure 2. The syntax of expressions and values is standard, following common textbook treatments of languages with binary pairs, binary sums, recursive types, and recursive functions [41]; constructs such as boolean conditionals, records, datatypes, or mutually recursive functions can be added without difficulty following the same pattern. We parameterize the syntax and semantics over primitive operations $\oplus$ that take inputs of base type only; for example, equality on integers, arithmetic and boolean operations. In $f(x).e$, both f and x are variable names; f is the name of the recursively defined function while x is the name of the argument. Both f and x are bound in e in an expression of the form $f(x).e$; generally, we adhere to the convention that in an expression of the form $x.e$, variable x is bound in e.

We abbreviate functional terms of the form $f(x).e$ using the letter κ, when convenient; similarly, we often abbreviate the expression $\{\mathtt{inl}(x_1).e_1; \mathtt{inr}(x_2).e_2\}$ as m. We sometimes refer to κ or m as a *code pointer* or *match pointer* respectively; in a fixed program, there are a fixed finite number of such terms and so we can share them instead of explicitly copying them when used in traces.

The syntax of traces is also defined in Figure 2. Trace expressions T have many syntactic forms in common with expressions; they differ primarily in the case and application trace forms, which include additional information showing how an application or case expression was evaluated. Traces can be viewed as witnessing terms for the operational derivation of an expression, and so their meaning is explained below along with that of the operational semantics rules. We will refer to trace expressions T as TML-traces when necessary to distinguish them from abstract traces $\mathcal{T}$ introduced in the previous section.

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

$$\frac{}{\gamma, c \Downarrow c, c} \qquad \frac{}{\gamma, x \Downarrow \gamma(x), x} \qquad \frac{\gamma, \overline{e} \Downarrow \overline{v}, \overline{T}}{\gamma, \oplus(\overline{e}) \Downarrow \hat{\oplus}(\overline{v}), \oplus(\overline{T})} \qquad \frac{\gamma, e_1 \Downarrow v_1, T_1 \qquad \gamma[x \mapsto v_1], e_2 \Downarrow v_2, T_2}{\gamma, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2, \texttt{let}\ x = T_1\ \texttt{in}\ T_2}$$

$$\frac{\gamma, e_1 \Downarrow v_1, T_1 \qquad \gamma, e_2 \Downarrow v_2, T_2}{\gamma, (e_1, e_2) \Downarrow (v_1, v_2), (T_1, T_2)} \qquad \frac{\gamma, e \Downarrow (v_1, v_2), T}{\gamma, \texttt{fst}(e) \Downarrow v_1, \texttt{fst}(T)} \qquad \frac{\gamma, e \Downarrow (v_1, v_2), T}{\gamma, \texttt{snd}(e) \Downarrow v_2, \texttt{snd}(T)}$$

$$\frac{\gamma, e \Downarrow v, T}{\gamma, \texttt{inl}(e) \Downarrow \texttt{inl}(v), \texttt{inl}(T)} \qquad \frac{(\texttt{inl}(x_1).e_1 \in m) \qquad \gamma, e \Downarrow \texttt{inl}(v), T \qquad \gamma[x_1 \mapsto v], e_1 \Downarrow v_1, T_1}{\gamma, \texttt{case}\ e\ \texttt{of}\ m \Downarrow v_1, \texttt{case}\ T \triangleright_{\texttt{inl}} x_1.T_1}$$

$$\frac{\gamma, e \Downarrow v, T}{\gamma, \texttt{inr}(e) \Downarrow \texttt{inr}(v), \texttt{inr}(T)} \qquad \frac{(\texttt{inr}(x_2).e_2 \in m) \qquad \gamma, e \Downarrow \texttt{inr}(v), T \qquad \gamma[x_2 \mapsto v], e_2 \Downarrow v_2, T_2}{\gamma, \texttt{case}\ e\ \texttt{of}\ m \Downarrow v_2, \texttt{case}\ T \triangleright_{\texttt{inr}} x_2.T_2}$$

$$\frac{\gamma, e \Downarrow v, T}{\gamma, \texttt{roll}(e) \Downarrow \texttt{roll}(v), \texttt{roll}(T)} \qquad \frac{\gamma, e \Downarrow \texttt{roll}(v), T}{\gamma, \texttt{unroll}(e) \Downarrow v, \texttt{unroll}(T)}$$

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

Figure 3: Dynamic semantics of Core TML: rules for expression evaluation.

3.1 Dynamic Semantics

We augment a standard large-step operational semantics for TML by adding a parameter T, which records a trace of the evaluation of the expression. The judgment $\gamma, e \Downarrow v, T$, defined in Figure 3, says that in environment γ, expression e evaluates to value v with trace T.

If we ignore the trace parameter in this judgment, then the rules are essentially the standard ones for a call-by-value, pure functional language with pairs, sums, and recursive types and functions [41]. In particular, pairs are constructed by pairing and can be taken apart using the `fst` and `snd` operations. Values of sum type are constructed using the left and right injection operations `inl`, `inr` and can be analyzed using the case expression, which examines a value of type $\tau_1 + \tau_2$ and calls the appropriate branch with the injected value bound to a variable. Values of recursive type $\mu\alpha.\tau$ are constructed using `roll` and destructed using `unroll`; these operations indicate the explicit isomorphisms in the *isorecursive* treatment of recursive types. Finally, functions are (as usual) constructed using the function expression $\texttt{fun}\ f(x).e$ and applied using function application $e_1\ e_2$.

Now if we consider the trace parameter, note that each rule has its own trace form, which builds the trace up from sub-traces obtained by the hypotheses of the rule. Traces can contain bound variables, reflecting the binding structure of the original expression. To illustrate, for let expressions, traces are similar to expressions:

$$\texttt{let}\ x = T_1\ \texttt{in}\ T_2$$

where we bind the variable in T_2.

The case and application evaluation traces record additional information about control flow. In either case, the first argument is evaluated to determine what expression to evaluate to obtain the final result. For case expressions, traces are of the form:

$$\texttt{case}\ T \triangleright_{\texttt{inl}} x_1.T_1 \quad \texttt{case}\ T \triangleright_{\texttt{inr}} x_2.T_2$$

where we record the trace of the case scrutinee T and the taken branch (T_1 or T_2), and we re-bind the variable (x_1 or x_2) in the trace of the taken branch. The subscript indicates which branch was taken. Similarly, for an application expression:

$$(T_1\ T_2) \triangleright_\kappa f(x).T$$

we record the traces of the function subexpression T_1, the argument subexpression T_2, and the trace T of the evaluation of the body of the function. The subscript $\kappa = \langle f(x).e, \Gamma \rangle$ is a code pointer indicating the function and the typing environment of the call. The Γ annotation is needed only to typecheck traces, so we usually elide it. Again, since the body trace can mention the function and argument names as free variables, we re-bind these variables.

$\boxed{\gamma, T \curvearrowright v}$

$$\frac{}{\gamma, c \curvearrowright c} \qquad \frac{}{\gamma, x \curvearrowright \gamma(x)} \qquad \frac{\gamma, \overline{T} \curvearrowright \overline{v}}{\gamma, \oplus(\overline{T}) \curvearrowright \hat{\oplus}(\overline{v})} \qquad \frac{\gamma, T_1 \curvearrowright v_1 \qquad \gamma[x \mapsto v_1], T_2 \curvearrowright v_2}{\gamma, \mathtt{let}\ x = T_1\ \mathtt{in}\ T_2 \curvearrowright v_2}$$

$$\frac{\gamma, T_1 \curvearrowright v_1 \qquad \gamma, T_2 \curvearrowright v_2}{\gamma, (T_1, T_2) \curvearrowright (v_1, v_2)} \qquad \frac{\gamma, T \curvearrowright (v_1, v_2)}{\gamma, \mathtt{fst}(T) \curvearrowright v_1} \qquad \frac{\gamma, T \curvearrowright (v_1, v_2)}{\gamma, \mathtt{snd}(T) \curvearrowright v_2}$$

$$\frac{\gamma, T \curvearrowright v}{\gamma, \mathtt{inl}(T) \curvearrowright \mathtt{inl}(v)} \qquad \frac{\gamma, T \curvearrowright \mathtt{inl}(v) \qquad \gamma[x_1 \mapsto v], T_1 \curvearrowright v_1}{\gamma, \mathtt{case}\ T \rhd_{\mathtt{inl}} x_1.T_1 \curvearrowright v_1}$$

$$\frac{\gamma, T \curvearrowright v}{\gamma, \mathtt{inr}(T) \curvearrowright \mathtt{inr}(v)} \qquad \frac{\gamma, T \curvearrowright \mathtt{inr}(v) \qquad \gamma[x_2 \mapsto v], T_2 \curvearrowright v_2}{\gamma, \mathtt{case}\ T \rhd_{\mathtt{inr}} x_2.T_2 \curvearrowright v_2}$$

$$\frac{\gamma, T \curvearrowright v}{\gamma, \mathtt{roll}(T) \curvearrowright \mathtt{roll}(v)} \qquad \frac{\gamma, T \curvearrowright \mathtt{roll}(v)}{\gamma, \mathtt{unroll}(T) \curvearrowright v}$$

$$\frac{}{\gamma, \mathtt{fun}\,\kappa \curvearrowright \langle \kappa, \gamma \rangle} \qquad \frac{\gamma, T_1 \curvearrowright \langle \kappa, \gamma' \rangle \qquad \gamma, T_2 \curvearrowright v_2 \qquad \gamma'[f \mapsto \langle \kappa, \gamma' \rangle, x \mapsto v_2], T \curvearrowright v}{\gamma, (T_1\ T_2) \rhd_\kappa f(x).T \curvearrowright v}$$

Figure 4: Dynamic semantics of Core TML: rules for trace replay.

We want to emphasize at this point that we do not necessarily expect that implementations routinely construct fully detailed traces along the above lines. Rather, the trace semantics is proposed here as a candidate for the most detailed form of provenance we will consider. Recording and compressing or filtering relevant information from traces in an efficient way is beyond the scope of this paper. However, some preliminary experiments in this direction have been performed in a recent paper on slicing for higher-order functional programs, based on a similar trace model [40].

Example 3.1. *Consider the factorial program expressed in Core TML:*

```
let f = fun f(x). if x = 0 then 1 else x*(f(x-1))
in f 4
```

The trace of this program has the form

```
let f = fun f(x). e
in f 4    |> f(x).(e |>_else x * (
    f(x-1) |> f(x).(e |>_else x * (
     f(x-1) |> f(x).(e |>_else x * (
      f(x-1) |> f(x).(e |>_else x * (
       f(x-1) |> f(x).(e |>_then 1))))))))) 
```

where `e = if x = 0 then 1 else x*(f(x-1))`. *The trace reflects that f calls itself four additional times when evaluating f 4 and the* `else`*-branch is taken four times, and finally the* `then`*-branch is taken. Here, we use subscripts* `then` *and* `else` *to indicate the branch taken instead of* `inl` *and* `inr`*.*

Remark 3.2. *The syntax of traces and expressions, and their corresponding evaluation rules, exhibit some redundancy. The syntax and semantics of expressions and traces could be fused so that both fall out as subsystems of one joint syntax / semantics. We adopt an explicit treatment for clarity, despite the resulting redundancy.*

The operational semantics rules in Figures 3 and 4 illustrate a recipe that appears straightforward to follow in order to extend the system to a more realistic (pure) language; it is less clear how to extend the trace semantics to handle effects, nontermination, or other features. It may be interesting to try to capture the recipe as a formal construction over operational semantic specifications.

Trace Replay. We equip traces with a semantics that relates them to expressions. We write $\gamma, T \curvearrowright v$ for the *replay* relation that reruns a trace on an environment (possibly different from the one originally used to construct T). Figure 4 shows the rules for replaying traces. The rules for most trace forms are the same as the standard rules for evaluating the corresponding expression forms. Essentially, these rules require that the same control flow branches are taken as in the original run. If the input environment is different enough that the same branches cannot be taken, then replay fails.

Remark 3.3. *This behavior should be contrasted with traces used in self-adjusting computation [4, 2]. Such traces must always recompute the updated result; however, they typically operate at a coarser granularity by tracking reads and writes to memory locations. Moreover, the traces are essentially graphs built in memory using references and closures, so it is not straightforward to traverse such traces to obtain fine-grained information about what happened at run-time, as we shall do in Section 4.*

Like evaluation, replay is deterministic, in that if a trace can be replayed on an environment then the resulting value is unique:

Theorem 3.4. *If* $\gamma, T \curvearrowright v_1$ *and* $\gamma, T \curvearrowright v_2$ *then* $v_1 = v_2$.

Proof. Proof is by (straightforward) structural induction on the first derivation and inversion on the second. ☐

3.2 Basic Properties of Traces

In this section, we identify key properties of traces, including type safety, and the consistency and fidelity properties that characterize how traces record the evaluation of an expression.

Determinacy and Type Safety. We employ a standard type system for expressions. Figure 5 shows the (standard) typing rules for expressions and Figure 6 shows the rules for values and environments. Type checking requires a variable context Γ that maps variables to types. We write $\Gamma \vdash e : \tau$ to indicate that e has type τ in context Γ. Similarly, we write $\Gamma \vdash_{\mathtt{T}} T : \tau$ to indicate that T is a well-formed trace of type τ in context Γ. Figure 7 shows the typing rules for traces. The unusual rules are those for case and application traces, whose form differs from the corresponding expression forms. One important point is that in the rule for application traces, the trace of the function body needs to typecheck in the same context Γ' as the body of e. This is why we allow the annotation Γ' indicating the environment of the called function in application traces.

As noted above, expressions (and hence also traces) can be well-formed at more than one type, but this does not matter since we are not concerned with typechecking algorithms here.

Theorem 3.5. *If* $\Gamma \vdash e : \tau$ *and* $\vdash \gamma : \Gamma$ *and* $\gamma, e \Downarrow v, T$ *then* $\vdash v : \tau$ *and* $\Gamma \vdash_{\mathtt{T}} T : \tau$.

Proof. Proof is by induction on the structure of the evaluation derivation, using inversion on the typing derivation. The only nonstandard cases are for the well-formedness of the trace, but these cases are straightforward. ☐

Replay is also type-safe in the obvious sense:

Theorem 3.6. *If* $\Gamma \vdash_{\mathtt{T}} T : \tau$ *and* $\vdash \gamma : \Gamma$ *and* $\gamma, T \curvearrowright v$ *then* $\vdash v : \tau$.

Proof. Proof is by induction on the structure of the replay derivation; most cases are similar to analogous cases for Theorem 3.5. ☐

Consistency and Fidelity. We say that a trace T is *consistent* with an environment γ if there exists v such that $\gamma, T \curvearrowright v$. A trace can easily be inconsistent with an environment, either because it is untyped nonsense and can never run (e.g. $\mathtt{fst}(42)$), or, more interestingly, because replaying leads to situations that disagree with the control flow of the trace (e.g. while replaying $\mathtt{case}\ T \triangleright_{\mathtt{inl}} x_1.T_1$, the replay of T yields $\mathtt{inr}(v)$).

Evaluation produces consistent traces, and replaying a trace on the same input yields the same value:

Theorem 3.7 (Consistency). *If* $\gamma, e \Downarrow v, T$ *then* $\gamma, T \curvearrowright v$.

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

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash \overline{e} : \overline{\tau} \qquad \oplus : \overline{\tau} \to \tau' \in \Sigma}{\Gamma \vdash \oplus(\overline{e}) : \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}\, x = e_1 \,\mathtt{in}\, e_2 : \tau_2}$$

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

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

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

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

Figure 5: Well-typed expressions of TML.

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

$$\frac{c : \tau \in \Sigma}{\vdash c : \tau} \qquad \frac{\vdash v_1 : \tau_1 \qquad \vdash v_2 : \tau_2}{\vdash (v_1, v_2) : \tau_1 \times \tau_2} \qquad \frac{\vdash v : \tau_1}{\vdash \mathtt{inl}(v) : \tau_1 + \tau_2} \qquad \frac{\vdash v : \tau_2}{\vdash \mathtt{inr}(v) : \tau_1 + \tau_2}$$

$$\frac{\vdash \gamma : \Gamma \qquad \Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2}{\vdash \langle\langle f(x).e, \Gamma\rangle, \gamma\rangle : \tau_1 \to \tau_2} \qquad \frac{\vdash v : \tau[\mu\alpha.\tau/\alpha]}{\vdash \mathtt{roll}(v) : \mu\alpha.\tau} \qquad \frac{\vdash v_1 : \tau_1 \qquad \cdots \qquad \vdash v_n : \tau_n}{\vdash [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] : [x_1 : \tau_1, \ldots, x_n : \tau_n]}$$

Figure 6: Value and environment typing.

Proof. Proof is by (straightforward) induction on the structure of derivations. □

The converse does not hold: a trace can be consistent without ever being produced by running a program. In particular, consistency does not check that the traces corresponding to bodies of function calls match the code pointers recorded in the trace. It is possible to refine the definition of replay so that the function bodies are checked against the traces, providing a stronger notion of consistency. However, this would complicate the replay semantics. In the rest of this article we usually consider traces obtained by running the tracing semantics. When this is the case, the derivation of $\gamma, e \Downarrow v, T$ is itself a witness to this, so there is no need to introduce an additional judgment that captures this invariant.

Furthermore, the trace produced by evaluation is faithful to the original expression, in the sense that whenever the trace can be successfully replayed on a different input, the result (and its trace) is the same as what we would obtain by rerunning e from scratch, and the resulting trace is the same as well. We call this property *fidelity*.

Theorem 3.8 (Fidelity). *If $\gamma, e \Downarrow v, T$ and $\gamma', T \curvearrowright v'$ then $\gamma', e \Downarrow v', T$.*

Proof. Straightforward proof by induction on the structure of derivations. The interesting cases are for case and application expressions; in each case, the induction hypothesis ensures that the intermediate sum or function value encountered when recomputing e in γ' matches that in the original derivation, so that the subtraces contingent on this value can be reused. □

Intuitively, fidelity corresponds to a repeatability or reproducibility property: it does not just guarantee that we get the same results when the trace is replayed on the same input, it also guarantees that the trace tells us what would happen if we rerun on inputs that are similar enough to the original input that replay can succeed. Thus, traces correspond to a form of *explanation*, analogous to forms of explanation explored in causal models and workflow provenance [14, 30, 31]. While we do not make more of this connection here, fidelity is also related to the correctness properties for various forms of slicing, including disclosure slicing (as discussed in Section 5).

$\boxed{\Gamma \vdash_{\mathtt{T}} T : \tau}$

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash_{\mathtt{T}} c : \tau} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash_{\mathtt{T}} x : \tau} \qquad \frac{\Gamma \vdash_{\mathtt{T}} \overline{T} : \overline{\tau} \qquad \oplus : \overline{\tau} \to \tau' \in \Sigma}{\Gamma \vdash_{\mathtt{T}} \oplus(\overline{T}) : \tau'} \qquad \frac{\Gamma \vdash_{\mathtt{T}} T_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash_{\mathtt{T}} T_2 : \tau_2}{\Gamma \vdash_{\mathtt{T}} \mathtt{let}\ x = T_1\ \mathtt{in}\ T_2 : \tau_2}$$

$$\frac{\Gamma \vdash_{\mathtt{T}} T_1 : \tau_1 \qquad \Gamma \vdash_{\mathtt{T}} T_2 : \tau_2}{\Gamma \vdash_{\mathtt{T}} (T_1, T_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash_{\mathtt{T}} T : \tau_1 \times \tau_2}{\Gamma \vdash_{\mathtt{T}} \mathtt{fst}(T) : \tau_1} \qquad \frac{\Gamma \vdash_{\mathtt{T}} T : \tau_1 \times \tau_2}{\Gamma \vdash_{\mathtt{T}} \mathtt{snd}(T) : \tau_2}$$

$$\frac{\Gamma \vdash_{\mathtt{T}} T : \tau_1}{\Gamma \vdash_{\mathtt{T}} \mathtt{inl}(T) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash_{\mathtt{T}} T : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau \qquad \Gamma, x_1 : \tau_1 \vdash_{\mathtt{T}} T_1 : \tau}{\Gamma \vdash_{\mathtt{T}} \mathtt{case}\ T\ \mathtt{of}\ \{\mathtt{inl}(x_1).e_1; \mathtt{inr}(x_2).e_2\} \triangleright_{\mathtt{inl}} x_1.T_1 : \tau}$$

$$\frac{\Gamma \vdash_{\mathtt{T}} T : \tau_2}{\Gamma \vdash_{\mathtt{T}} \mathtt{inr}(T) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash_{\mathtt{T}} T : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash_{\mathtt{T}} T_2 : \tau}{\Gamma \vdash_{\mathtt{T}} \mathtt{case}\ T\ \mathtt{of}\ \{\mathtt{inl}(x_1).e_1; \mathtt{inr}(x_2).e_2\} \triangleright_{\mathtt{inr}} x_2.T_2 : \tau}$$

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

$$\frac{\Gamma \vdash_{\mathtt{T}} T_1 : \tau_1 \to \tau_2 \qquad \Gamma', f{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash e : \tau_2 \qquad \Gamma \vdash_{\mathtt{T}} T_2 : \tau_1 \qquad \Gamma', f{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash_{\mathtt{T}} T : \tau_2}{\Gamma \vdash_{\mathtt{T}} (T_1\ T_2) \triangleright_{f(x).e,\Gamma'} f(x).T : \tau_2}$$

$$\frac{\Gamma \vdash_{\mathtt{T}} T : \mu\alpha.\tau}{\Gamma \vdash_{\mathtt{T}} \mathtt{unroll}(T) : \tau[\mu\alpha.\tau/\alpha]} \qquad \frac{\Gamma \vdash_{\mathtt{T}} T : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash_{\mathtt{T}} \mathtt{roll}(T) : \mu\alpha.\tau}$$

Figure 7: Well-typed traces of TML.

Remark 3.9. *As noted at the beginning of the section, we made two simplifying assumptions: we consider traces only for terminating runs, and we exclude side-effects from the language. These assumptions are reasonable for many application areas of provenance (for example, in scientific computation and databases), but it is naturally of interest to consider extending our approaches to trace nonterminating or effectful computations. These raise potential complications: for example, adapting the trace semantics to a small-step semantics seems nontrivial, and it is not as clear what the appropriate correctness properties are for traces involving effects (including nondeterminism or allocation). These are interesting areas for exploration in future work.*

4 Provenance Views and Extraction

In this section we consider different kinds of views and queries over provenance traces. To be specific, we consider a consistent triple (γ, T, v) where $\gamma, T \curvearrowright v$ to be the "traces" in the sense of the provenance security framework. Then queries over these triples correspond to sets of triples (generally definable using some compact syntax), and views correspond to functions from triples to some other data. We first consider a general class of views definable using *annotation propagation*, by giving a generic framework for extracting other kinds of provenance from execution traces. These forms of provenance induce provenance views in a natural way if we allow for initial annotations that uniquely identify each part of a value by a path.

4.1 Annotations, Paths, and Provenance Extraction

Many previous approaches to provenance can be viewed as performing a form of *annotation propagation*. The idea is to decorate the input with annotations (often, initially, unique identifiers) and propagate the annotations through the evaluation. For example, in where-provenance, annotations are optional tags that can be thought of as pointers showing where output data was copied from in the source [11, 10]. Other techniques, such as why-, how-, and dependency provenance, can also be defined in terms of annotation propagation [28, 27, 9, 18]. We gave similar definitions of different forms of provenance using a common framework for XQuery [13]; some of the properties proved are generalizations of properties shown there or in [10].

$$
\begin{array}{rcl}
|[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]| & = & [x_1 \mapsto |v_1|, \ldots, x_n \mapsto |v_n|] \\
|c^a| & = & c \\
|(\widehat{v}_1, \widehat{v}_2)^a| & = & (|\widehat{v}_1|, |\widehat{v}_2|) \\
|\mathtt{inl}(\widehat{v})^a| & = & \mathtt{inl}(|\widehat{v}|) \\
|\mathtt{inr}(\widehat{v})^a| & = & \mathtt{inr}(|\widehat{v}|) \\
|\langle \kappa, \widehat{\gamma} \rangle^a| & = & \langle \kappa, |\widehat{\gamma}| \rangle \\
|\mathtt{roll}(\widehat{v})^a| & = & \mathtt{roll}(|\widehat{v}|)
\end{array}
$$

Figure 8: Erasure operation.

$$
\begin{array}{rcl}
occ(c^a) & = & \{c^a\} \\
occ((\widehat{v}_1, \widehat{v}_2)^a) & = & \{(\widehat{v}_1, \widehat{v}_2)^a\} \cup occ(\widehat{v}_1) \cup occ(\widehat{v}_2) \\
occ((\mathtt{inl}(\widehat{v}))^a) & = & \{(\mathtt{inl}(\widehat{v}))^a\} \cup occ(\widehat{v}) \\
occ((\mathtt{inr}(\widehat{v}))^a) & = & \{(\mathtt{inr}(\widehat{v}))^a\} \cup occ(\widehat{v}) \\
occ((\mathtt{roll}(\widehat{v}))^a) & = & \{(\mathtt{roll}(\widehat{v}))^a\} \cup occ(\widehat{v}) \\
occ(\langle \kappa, \widehat{\gamma} \rangle^a) & = & \{\langle \kappa, \widehat{\gamma} \rangle^a\} \cup occ(\widehat{\gamma}) \\
occ(\widehat{\gamma}) & = & \displaystyle\bigcup_{x \in \mathrm{dom}(\widehat{\gamma})} occ(\widehat{\gamma}(x))
\end{array}
$$

Figure 9: Occurrences of annotated values.

Based on this observation, we define a provenance extraction framework in which values are decorated with annotations and extraction functions take traces and return annotated values that can be interpreted as useful provenance information. We first define annotated values and give a generic annotation-propagation operation. We apply this framework to specify several concrete annotation schemes and extraction functions.

Annotations. Let A be an arbitrary set of *annotations* a, which we usually assume includes a blank annotation $\bot$ and a countably infinite set of identifiers $\ell \in \mathrm{Loc}$, called *locations*. We define *A-annotated values* $\widehat{v}$ (or just *annotated values*, when A is clear) using the following grammar:

$$
\begin{array}{rcl}
\widehat{v} & ::= & w^a \\
\widehat{\gamma} & ::= & [x_1 \mapsto \widehat{v}_1, \ldots, x_n \mapsto \widehat{v}_n] \\
w & ::= & c \mid (\widehat{v}_1, \widehat{v}_2) \mid \mathtt{inl}(\widehat{v}) \mid \mathtt{inr}(\widehat{v}) \mid \langle \kappa, \widehat{\gamma} \rangle \mid \mathtt{roll}(\widehat{v})
\end{array}
$$

We write $\widehat{\gamma}$ for *annotated environments* mapping variables to annotated values. We define an erasure function $|\widehat{v}|$ that maps each annotated value to an ordinary value by erasing the annotations. Similarly, $|\widehat{\gamma}|$ is the ordinary environment obtained by erasing the annotations from the values of $\widehat{\gamma}$. This function is defined mutually recursively on annotated values and environments as shown in Figure 8. We also introduce a notation for the set of annotated values occurring in a value in Figure 9. Moreover, we write

$$
occ^{\not\bot}(\widehat{v}) = \{w^a \in occ(\widehat{v}) \mid a \neq \bot\}
$$

for the set of annotated values with annotation $a \neq \bot$.

Paths as annotations. For annotations to be useful when the full input is unavailable, we consider annotations where the locations ℓ are *paths* that uniquely address parts of the input environment. Paths π have syntax:

$$
\pi ::= \epsilon \mid x.\pi \mid 1.\pi \mid 2.\pi
$$

$$\begin{array}{rcl}
v[\epsilon] & = & v \\
(v_1, v_2)[1.\pi] & = & v_1[\pi] \\
(v_1, v_2)[2.\pi] & = & v_2[\pi] \\
(\mathtt{inl}(v))[1.\pi] & = & v[\pi] \\
(\mathtt{inr}(v))[1.\pi] & = & v[\pi] \\
(\mathtt{roll}(v))[1.\pi] & = & v[\pi] \\
(\langle \kappa, \gamma \rangle)[1.\pi] & = & \gamma[\pi] \\
\gamma[x.\pi] & = & \gamma(x)[\pi]
\end{array}$$

Figure 10: Path lookup operation.

$$\begin{array}{rcl}
\mathsf{path}_\pi(\gamma) & = & [x_1 \mapsto \mathsf{path}_{\pi.x_1}(\gamma(x_1)), \ldots, x_n \mapsto \mathsf{path}_{\pi.x_n}(\gamma(x_n))] \\
\mathsf{path}_\pi(c) & = & c^\pi \\
\mathsf{path}_\pi((v_1, v_2)) & = & (\mathsf{path}_{\pi.1}(v_1), \mathsf{path}_{\pi.2}(v_2))^\pi \\
\mathsf{path}_\pi(\mathtt{inl}(v)) & = & \mathtt{inl}(\mathsf{path}_{\pi.1}(v))^\pi \\
\mathsf{path}_\pi(\mathtt{inr}(v)) & = & \mathtt{inr}(\mathsf{path}_{\pi.1}(v))^\pi \\
\mathsf{path}_\pi(\langle \kappa, \gamma \rangle) & = & \langle \kappa, \mathsf{path}_{\pi.1}(\gamma) \rangle^\pi
\end{array}$$

Figure 11: Path annotation operation.

and we consider path concatenation $\pi.\pi'$ to be associative with unit ϵ, so that we may write $\pi.i$ to construct a pattern ending in i. Paths address parts of values or environments; we write $v[\pi]$ or $\gamma[\pi]$ for the part of v or γ addressed by π, defined in Figure 10.

We write $\mathsf{path}(\gamma)$ for the environment γ with each component annotated with the path to that component. More generally, we define $\mathsf{path}_\pi(\gamma)$ and $\mathsf{path}_\pi(v)$ as shown in Figure 11. Then $\mathsf{path}(v) = \mathsf{path}_\epsilon(v)$ and $\mathsf{path}(\gamma) = \mathsf{path}_\epsilon(\gamma)$. For example, $\mathsf{path}([x \mapsto (1,2), y \mapsto \mathtt{inl}(4)]) = [x \mapsto (1^{x.1}, 2^{x.2})^x, y \mapsto \mathtt{inl}(4^{y.1})^y]$.

Extraction framework. We will define a family of provenance extraction functions $\mathsf{F}(T, \widehat{\gamma})$ that take a trace T and an environment $\widehat{\gamma}$ and return an annotated value. Each such F can be specified by giving the following annotation-propagation functions:

$$\begin{array}{rcl}
\mathsf{F}_c, \mathsf{F}_\kappa & : & A \\
\mathsf{F}_1, \mathsf{F}_2, \mathsf{F}_L, \mathsf{F}_R, \mathsf{F}_{\mathsf{app}}, \mathsf{F}_{\mathtt{unroll}} & : & A \times A \to A \\
\mathsf{F}_\oplus & : & A^n \to A \qquad \text{(where } \oplus \text{ is } n\text{-ary)}
\end{array}$$

Each function shows how the annotations involved in the corresponding computational step propagate to the result. For example, $\mathsf{F}_1(a, b)$ gives the annotation on the result of a `fst`-projection, where a is the annotation on the pair and b is the annotation of the first element. Figure 12 shows how to propagate annotations through a trace given basic annotation-propagation functions.

Remark 4.1. *The extraction framework hard-wires the behavior of certain operations such as* `let`, `inl()`*,* `inr()`*,*`roll()`*, and pairing, using $\bot$ to handle all of them. On the other hand, even though these constructors are hard-wired so that the top-level annotation is always $\bot$, this does not imply that the first arguments supplied to the corresponding extraction functions $\mathsf{F}_1, \mathsf{F}_2$, etc. are always $\bot$; see Example 4.13 for an illustration of this point.*

It would also be possible to extend the framework to allow greater customization; however, this functionality is not needed by any of the forms of provenance in this article. We believe that the framework presented in this paper is

$$
\begin{array}{rcll}
\mathsf{F}(x,\widehat{\gamma}) & = & \widehat{\gamma}(x) & \\
\mathsf{F}(\mathtt{let}\ x = T_1\ \mathtt{in}\ T_2,\widehat{\gamma}) & = & \mathsf{F}(T_2,\widehat{\gamma}[x{\mapsto}\mathsf{F}(T_1,\widehat{\gamma})]) & \\
\mathsf{F}(c,\widehat{\gamma}) & = & c^{\mathsf{F}_c} & \\
\mathsf{F}(\oplus(T_1,\ldots,T_n),\widehat{\gamma}) & = & (\hat{\oplus}(c_1,\ldots,c_n))^{\mathsf{F}_\oplus(a_1,\ldots,a_n)} & \text{where} \quad c_i^{a_i} = \mathsf{F}(T_i,\widehat{\gamma}) \\
\mathsf{F}((T_1,T_2),\widehat{\gamma}) & = & (\mathsf{F}(T_1,\widehat{\gamma}),\mathsf{F}(T_2,\widehat{\gamma}))^{\perp} & \\
\mathsf{F}(\mathtt{fst}(T),\widehat{\gamma}) & = & v_1^{\mathsf{F}_1(a,b)} & \text{where} \quad (v_1^b,\widehat{v}_2)^a = \mathsf{F}(T,\widehat{\gamma}) \\
\mathsf{F}(\mathtt{snd}(T),\widehat{\gamma}) & = & v_2^{\mathsf{F}_2(a,b)} & \text{where} \quad (\widehat{v}_1,v_2^b)^a = \mathsf{F}(T,\widehat{\gamma}) \\
\mathsf{F}(\mathtt{inl}(T),\widehat{\gamma}) & = & \mathtt{inl}(\mathsf{F}(T,\widehat{\gamma}))^{\perp} & \\
\mathsf{F}(\mathtt{inr}(T),\widehat{\gamma}) & = & \mathtt{inr}(\mathsf{F}(T,\widehat{\gamma}))^{\perp} & \\
\mathsf{F}((\mathtt{case}\ T) \rhd_{\mathtt{inl}} x.T_1,\widehat{\gamma}) & = & v^{\mathsf{F}_L(a,b)} & \text{where} \quad \mathtt{inl}(\widehat{v})^a = \mathsf{F}(T,\widehat{\gamma}) \\
 & & & \text{and} \quad v^b = \mathsf{F}(T_1,\widehat{\gamma}[y{\mapsto}\widehat{v}]) \\
\mathsf{F}((\mathtt{case}\ T) \rhd_{\mathtt{inr}} y.T_2,\widehat{\gamma}) & = & v^{\mathsf{F}_R(a,b)} & \text{where} \quad \mathtt{inr}(\widehat{v})^a = \mathsf{F}(T,\widehat{\gamma}) \\
 & & & \text{and} \quad v^b = \mathsf{F}(T_2,\widehat{\gamma}[y{\mapsto}\widehat{v}]) \\
\mathsf{F}(\mathtt{fun}\ \kappa,\widehat{\gamma}) & = & \langle\kappa,\widehat{\gamma}\rangle^{\mathsf{F}_\kappa} & \\
\mathsf{F}((T_1\ T_2) \rhd_\kappa f(x).T,\widehat{\gamma}) & = & v^{\mathsf{F}_{\mathsf{app}}(a,b)} & \text{where} \quad \langle\kappa,\widehat{\gamma}'\rangle^a = \mathsf{F}(T_1,\widehat{\gamma}) \\
 & & & \text{and} \quad \widehat{v}_2 = \mathsf{F}(T_2,\widehat{\gamma}) \\
 & & & \text{and} \quad v^b = \mathsf{F}(T,\widehat{\gamma}'[f{\mapsto}\langle\kappa,\widehat{\gamma}'\rangle^a, x{\mapsto}\widehat{v}_2])) \\
\mathsf{F}(\mathtt{roll}(T),\widehat{\gamma}) & = & \mathtt{roll}(\mathsf{F}(T,\widehat{\gamma}))^{\perp} & \\
\mathsf{F}(\mathtt{unroll}(T),\widehat{\gamma}) & = & v^{\mathsf{F}_{\mathtt{unroll}}(a,b)} & \text{where} \quad \mathtt{roll}(v^b)^a = \mathsf{F}(T,\widehat{\gamma})
\end{array}
$$

Figure 12: Generic extraction.

general enough to be of use beyond the three provenance models we considered, but we do not know how one could prove that it is general enough for all purposes — or how one could prove that any alternative framework is general enough for all purposes. It is also possible that there are natural forms of provenance that do not fit (a reasonable generalization of) the framework.

Theorem 4.2. *Every generic provenance extraction function is compatible with replay: that is, for any $\widehat{\gamma}, T, v$, if $|\widehat{\gamma}|, T \curvearrowright v$ then $|\mathsf{F}(T,\widehat{\gamma})| = v$.*

Proof. Straightforward induction on replay derivations. □

Remark 4.3. *Consider the trivial annotation structure Triv with underlying annotation set $\{\perp\}$. In this setting, the erasure function $|-|$ is bijective; its inverse just decorates each part of a value with $\perp$. Consider also a trivial instance of the generic provenance framework for $\{\perp\}$ such that $\mathsf{Triv}_\kappa = \perp$ and $\mathsf{Triv}_c = \perp$ and all of the annotation-propagation functions are constant functions returning $\perp$, that is, $\mathsf{Triv}_1(x,y) = \perp$, etc. This instance of the provenance framework is essentially the same as the trace replay semantics defined by the judgment $\gamma, T \curvearrowright v$. Thus, the generic extraction framework can be viewed as a denotational presentation of the replay semantics of traces, generalized to allow for annotated values.*

Where-provenance. Where-provenance can be defined via an annotation-propagating semantics where annotations are either labels ℓ or the blank annotation $\perp$. Intuitively, for where-provenance, an explicit label ℓ annotating a part of the input indicates that that part "comes from" a part of the input with the same label; an annotation $\perp$ provides no information about where the output part "comes from" in the input (if anywhere). We define the where-provenance semantics $\mathsf{W}(T,\widehat{\gamma})$ using the following annotation-propagation functions:

$$
\begin{array}{rcl}
\mathsf{W}_c, \mathsf{W}_\kappa & = & \perp \\
\mathsf{W}_1, \mathsf{W}_2, \mathsf{W}_L, \mathsf{W}_R, \mathsf{W}_{\mathsf{app}}, \mathsf{W}_{\mathtt{unroll}} & = & \lambda(x,y).y \\
\mathsf{W}_\oplus & = & \lambda(a_1,\ldots,a_n).\perp
\end{array}
$$

Essentially, these functions preserve the annotations of data that are copied, and annotate computed or constructed data with $\perp$. This semantics is similar to that in Buneman et al. [10] and previous treatments of where-provenance in

$$
\begin{array}{rcll}
\mathsf{W}(c,\widehat{\gamma}) & = & c^{\perp} & \\
\mathsf{W}(\oplus(T_1,\ldots,T_n),\widehat{\gamma}) & = & (\hat{\oplus}(c_1,\ldots,c_n))^{\perp} & \text{where} \quad c_i^{a_i} = \mathsf{W}(T_i,\widehat{\gamma}) \\
\mathsf{W}(\mathtt{fst}(T),\widehat{\gamma}) & = & \widehat{v}_1 & \text{where} \quad (\widehat{v}_1,\widehat{v}_2)^a = \mathsf{W}(T,\widehat{\gamma}) \\
\mathsf{W}(\mathtt{snd}(T),\widehat{\gamma}) & = & \widehat{v}_2 & \text{where} \quad (\widehat{v}_1,\widehat{v}_2)^a = \mathsf{W}(T,\widehat{\gamma}) \\
\mathsf{W}((\mathtt{case}\ T) \triangleright_{\mathtt{inl}} x.T_1,\widehat{\gamma}) & = & \mathsf{W}(T_1,\widehat{\gamma}[x \mapsto \widehat{v}]) & \text{where} \quad \mathtt{inl}(\widehat{v})^a = \mathsf{W}(T,\widehat{\gamma}) \\
\mathsf{W}((\mathtt{case}\ T) \triangleright_{\mathtt{inr}} y.T_2,\widehat{\gamma}) & = & \mathsf{W}(T_2,\widehat{\gamma}[y \mapsto \widehat{v}]) & \text{where} \quad \mathtt{inr}(\widehat{v})^a = \mathsf{W}(T,\widehat{\gamma}) \\
\mathsf{W}(\mathtt{fun}\ \kappa,\widehat{\gamma}) & = & \langle \kappa,\widehat{\gamma}\rangle^{\perp} & \\
\mathsf{W}((T_1\ T_2) \triangleright_{\kappa} f(x).T,\widehat{\gamma}) & = & \mathsf{W}(T,\widehat{\gamma}'[f \mapsto \langle \kappa,\widehat{\gamma}'\rangle^a, x \mapsto \widehat{v}]) & \text{where} \quad \langle \kappa,\widehat{\gamma}'\rangle^a = \mathsf{W}(T_1,\widehat{\gamma}) \\
 & & & \text{and} \quad \widehat{v} = \mathsf{W}(T_2,\widehat{\gamma}) \\
\mathsf{W}(\mathtt{unroll}(T),\widehat{\gamma}) & = & \widehat{v} & \text{where} \quad \mathtt{roll}(\widehat{v})^a = \mathsf{W}(T,\widehat{\gamma})
\end{array}
$$

Figure 13: Where-provenance extraction (selected cases).

databases, adapted to TML. Figure 13 shows the generic semantics specialized to where-provenance. Note that for a function like "factorial", the where-provenance of the output is always $\perp$.

Example 4.4. *Recall the example program from Section 1.1:*

```
let y = 2@L in
let f x = if x = y then y else x+1 in
map f [1@L1,2@L2,3@L3]
```

The result of the where-provenance extraction semantics applied to this program is $[2, 2^L, 4]$*, as shown graphically in Figure 1, showing that the second result element is copied from* y *and giving no information about the other two.*

To state the key property of where-provenance, we use the function occ introduced earlier in this section. The key property of where-provenance is that if an annotated value w^a appears in $\mathsf{W}(T,\widehat{\gamma})$ with annotation $a \neq \perp$, then w^a is an exact copy (including any nested annotations) of a part of $\widehat{\gamma}$.

Theorem 4.5. *Suppose* $|\widehat{\gamma}|, e \Downarrow v, T$*. Then* $occ^{\not\perp}(\mathsf{W}(T,\widehat{\gamma})) \subseteq occ^{\not\perp}(\widehat{\gamma})$*.*

Proof. See Appendix A.1. ☐

Remark 4.6. *Buneman et al. [10] consider a where-provenance semantics for database query and update languages (with nested collection types and pairs, but no recursion or datatypes), which we adapt here to a conventional functional language (with recursion and datatypes, but no collection types). The basic idea, propagating annotations from the input to output when data are copied, is the same. They did not propose a tracing model of their calculus, but instead defined where-provenance via a syntactic translation that inserts annotation propagation code. The correctness property we discuss here corresponds to their* copying *property [10, Prop. 5.5]. They studied additional query normalization and semantic expressiveness properties that we do not address here.*

Expression provenance. To model expression provenance, we consider *expression annotations* t consisting of labels ℓ, blanks $\perp$, constants c, or primitive function applications $\oplus(t_1,\ldots,t_n)$.

$$t ::= \ell \mid c \mid \oplus(t_1,\ldots,t_n) \mid \perp$$

Intuitively, a label ℓ indicates that a part of the output is copied from a part of the input with the same label; a constant c indicates an output part that is built by evaluating constant in the program; a term $\oplus(t_1,\ldots,t_n)$ indicates a part of the output that is computed by evaluating $\oplus$ on values obtained from $t_1,\ldots,t_n$, and $\perp$ provides no information about how a part of the output was computed from the input. We define expression-provenance extraction $\mathsf{E}(T,\widehat{\gamma})$ in much the same way as W, with the following differences:

$$\mathsf{E}_c = c \qquad \mathsf{E}_{\oplus}(t_1,\ldots,t_n) = \oplus(t_1,\ldots,t_n)$$

Figure 14 shows the generic semantics specialized to expression-provenance.

$$
\begin{array}{rcll}
\mathsf{E}(c,\widehat{\gamma}) & = & c^c & \\
\mathsf{E}(\oplus(T_1,\ldots,T_n),\widehat{\gamma}) & = & (\hat{\oplus}(c_1,\ldots,c_n))^{\oplus(t_1,\ldots,t_n)} & \text{where} \quad c_i^{t_i} = \mathsf{E}(T_i,\widehat{\gamma}) \\
\mathsf{E}(\mathtt{fst}(T),\widehat{\gamma}) & = & \widehat{v}_1 & \text{where} \quad (\widehat{v}_1,\widehat{v}_2)^t = \mathsf{E}(T,\widehat{\gamma}) \\
\mathsf{E}(\mathtt{snd}(T),\widehat{\gamma}) & = & \widehat{v}_2 & \text{where} \quad (\widehat{v}_1,\widehat{v}_2)^t = \mathsf{E}(T,\widehat{\gamma}) \\
\mathsf{E}((\mathtt{case}\ T) \triangleright_{\mathtt{inl}} x.T_1,\widehat{\gamma}) & = & \mathsf{E}(T_1,\widehat{\gamma}[x\mapsto\widehat{v}]) & \text{where} \quad \mathsf{E}(T,\widehat{\gamma}) = \mathtt{inl}(\widehat{v})^t \\
\mathsf{E}((\mathtt{case}\ T) \triangleright_{\mathtt{inr}} y.T_2,\widehat{\gamma}) & = & \mathsf{E}(T_2,\widehat{\gamma}[y\mapsto\widehat{v}]) & \text{where} \quad \mathsf{E}(T,\widehat{\gamma}) = \mathtt{inr}(\widehat{v})^t \\
\mathsf{E}(\mathtt{fun}\ \kappa,\widehat{\gamma}) & = & \langle\kappa,\widehat{\gamma}\rangle^{\perp} & \\
\mathsf{E}((T_1\ T_2) \triangleright_{\kappa} f(x).T,\widehat{\gamma}) & = & \mathsf{E}(T,\widehat{\gamma}'[f\mapsto\langle\kappa,\widehat{\gamma}'\rangle^t, x\mapsto\widehat{v}) & \text{where} \quad \langle\kappa,\widehat{\gamma}'\rangle^t = \mathsf{E}(T_1,\widehat{\gamma})) \\
 & & & \text{and} \quad \widehat{v} = \mathsf{E}(T_2,\widehat{\gamma}) \\
\mathsf{E}(\mathtt{unroll}(T),\widehat{\gamma}) & = & \widehat{v} & \text{where} \quad \mathtt{roll}(\widehat{v})^t = \mathsf{E}(T,\widehat{\gamma})
\end{array}
$$

Figure 14: Expression provenance extraction (selected cases).

Example 4.7. *Continuing with the* map *example from Section 1.1, the result of the expressione-provenance extraction semantics applied to this program is* $[2^{L_1+1}, 2^{L}, 4^{L_3+1}]$*, as shown graphically in Figure 1. This shows that the second result element is copied from* y *and the other two arguments are computed by incrementing the first and last elements of the input, respectively. Observe that this is strictly more informative than the where-provenance.*

The correctness property for expression provenance states that the expression annotation correctly recomputes the value it annotates. To formalize this, we use the auxiliary definitions $occ()$, $occ^{\not\perp}()$ introduced for where-provenance. Let $h : \mathrm{Loc} \to \mathrm{Val}$ be a function from locations to values, and let $h(t)$ be the value obtained by evaluating annotation term t with values from h substituted for locations in t. We say that h is *consistent* with $\widehat{v}$ if whenever $w^t \in occ^{\not\perp}(\widehat{v})$, we have $h(t) = |w|$. Similarly, h is consistent with $\widehat{\gamma}$ if whenever $w^t \in occ^{\not\perp}(\widehat{\gamma})$, we have $h(t) = |w|$. We note that for any distinctly-annotated value, for example $\mathsf{path}(\gamma)$, there is always a consistent mapping h, obtained by mapping ℓ to $|w|$ whenever $w^\ell \in occ(\widehat{\gamma})$.

Example 4.8. *Consider* $\gamma = [x \mapsto (1,2), y \mapsto \mathtt{inl}(3)]$ *and* $\widehat{\gamma} = \mathsf{path}(\gamma) = [x \mapsto (1^{x.1}, 2^{x.2})^x, y \mapsto \mathtt{inl}(3^{y}.1)]$*. Then the consistent mapping* h *is defined as follows:*

$$
\begin{array}{rclrclrcl}
h(x) & = & (1,2) & h(x.1) & = & 1 & h(x.2) & = & 2 \\
h(y) & = & \mathtt{inl}(3) & h(y.1) & = & 3 & & &
\end{array}
$$

Observe in particular that this illustrates that $|\widehat{\gamma}|$ *is typically not a consistent mapping for* $\widehat{\gamma}$ *(since in this example,* $|\widehat{\gamma}| = \gamma \neq h$*).*

Theorem 4.9. *Suppose* $|\widehat{\gamma}|, e \Downarrow v', T$*. Then if* h *is consistent with* $\widehat{\gamma}$*, then* h *is also consistent with* $\mathsf{E}(T,\widehat{\gamma})$*.*

Proof. Similar to the proof of Theorem 4.5. See Appendix A.2. □

Dependency provenance. To extract dependency provenance (adapting the definition from [17]) we will use annotations ϕ that are sets of source locations $\{\ell_1,\ldots,\ell_n\}$, and we take the default annotation $\perp$ to be the empty set $\emptyset$. Initial annotations consist of disjoint singleton sets $\{\ell\}$. We define $(\widehat{v})^{+a}$ to mean adding annotations a to the top-level of $\widehat{v} = w^a$; that is, $(w^a)^{+b} = w^{a\cup b}$. We define $\mathsf{D}(T,\widehat{\gamma})$ using the following propagation functions:

$$
\begin{array}{rcl}
\mathsf{D}_c, \mathsf{D}_\kappa & = & \emptyset \\
\mathsf{D}_1, \mathsf{D}_2, \mathsf{D}_L, \mathsf{D}_R, \mathsf{D}_{\mathsf{app}}, \mathsf{D}_{\mathtt{unroll}} & = & \lambda(x,y).x \cup y \\
\mathsf{D}_\oplus & = & \lambda(a_1,\ldots,a_n).a_1 \cup \cdots \cup a_n
\end{array}
$$

This semantics is based on the dynamic provenance tracking semantics given by Cheney et al. [17], generalized to TML. Figure 15 shows the generic semantics specialized to dependency-provenance.

Example 4.10. *Continuing with the* map *example from Section 1.1, the result of the dependency-provenance extraction semantics applied to this program is* $[2^{L_1,L}, 2^{L_1,L}, 4^{L_3,L}]$*, as shown graphically in Figure 1. This shows that all three arguments depend on both L and on the respective element of the input list. This information is not computable from the where-provenance or expression-provenance, or vice versa.*

$$
\begin{array}{rcll}
\mathsf{D}(c,\widehat{\gamma}) & = & c^{\emptyset} & \\
\mathsf{D}(\oplus(T_1,\ldots,T_n),\widehat{\gamma}) & = & (\hat{\oplus}(c_1,\ldots,c_n))^{\bigcup_i \phi_i} & \text{where} \quad c_i^{\phi_i} = \mathsf{D}(T_i,\widehat{\gamma}) \\
\mathsf{D}(\mathtt{fst}(T),\widehat{\gamma}) & = & (\widehat{v}_1)^{+\phi} & \text{where} \quad (\widehat{v}_1,\widehat{v}_2)^{\phi} = \mathsf{D}(T,\widehat{\gamma}) \\
\mathsf{D}(\mathtt{snd}(T),\widehat{\gamma}) & = & (\widehat{v}_2)^{+\phi} & \text{where} \quad (\widehat{v}_1,\widehat{v}_2)^{\phi} = \mathsf{D}(T,\widehat{\gamma}) \\
\mathsf{D}((\mathtt{case}\ T) \triangleright_{\mathtt{inl}} x.T_1,\widehat{\gamma}) & = & \mathsf{D}(T_1,\widehat{\gamma}[x{\mapsto}\widehat{v}])^{+\phi} & \text{where} \quad \mathtt{inl}(\widehat{v})^{\phi} = \mathsf{D}(T,\widehat{\gamma}) \\
\mathsf{D}((\mathtt{case}\ T) \triangleright_{\mathtt{inr}} y.T_2,\widehat{\gamma}) & = & \mathsf{D}(T_2,\widehat{\gamma}[y{\mapsto}\widehat{v}])^{+\phi} & \text{where} \quad \mathtt{inr}(\widehat{v})^{\phi} = \mathsf{D}(T,\widehat{\gamma}) \\
\mathsf{D}(\mathtt{fun}\,\kappa,\widehat{\gamma}) & = & \langle\kappa,\widehat{\gamma}\rangle^{\emptyset} & \\
\mathsf{D}((T_1\ T_2) \triangleright_{\kappa} f(x).T,\widehat{\gamma}) & = & \mathsf{D}(T,\widehat{\gamma}'[f{\mapsto}\langle\kappa,\widehat{\gamma}'\rangle^{\phi}, x{\mapsto}\widehat{v}])^{+\phi} & \text{where} \quad \langle\kappa,\widehat{\gamma}'\rangle^{\phi} = \mathsf{D}(T_1,\widehat{\gamma}) \\
& & & \text{and} \quad \widehat{v} = \mathsf{D}(T_2,\widehat{\gamma}) \\
\mathsf{D}(\mathtt{unroll}(T),\widehat{\gamma}) & = & (\widehat{v})^{+\phi} & \text{where} \quad \mathtt{roll}(\widehat{v})^{\phi} = \mathsf{D}(T,\widehat{\gamma})
\end{array}
$$

Figure 15: Dependency provenance extraction.

$\boxed{\widehat{v} \approx_\ell \widehat{v}'}$

$$
\frac{w \approx_\ell w'}{w^{\phi} \approx_\ell (w')^{\phi}} \qquad \frac{\ell \in \phi \cap \phi'}{w^{\phi} \approx_\ell (w')^{\phi'}} \qquad \frac{}{c \approx_\ell c}
$$

$$
\frac{\widehat{v}_1 \approx_\ell \widehat{v}_1' \qquad \widehat{v}_2 \approx_\ell \widehat{v}_2'}{(\widehat{v}_1,\widehat{v}_2) \approx_\ell (\widehat{v}_1',\widehat{v}_2')} \qquad \frac{\widehat{v} \approx_\ell \widehat{v}'}{\mathtt{inl}(\widehat{v}) \approx_\ell \mathtt{inl}(\widehat{v}')} \qquad \frac{\widehat{v} \approx_\ell \widehat{v}'}{\mathtt{inr}(\widehat{v}) \approx_\ell \mathtt{inr}(\widehat{v}')} \qquad \frac{\widehat{v} \approx_\ell \widehat{v}'}{\mathtt{roll}(\widehat{v}) \approx_\ell \mathtt{roll}(\widehat{v}')} \qquad \frac{\gamma \approx_\ell \gamma'}{\langle\kappa,\gamma\rangle \approx_\ell \langle\kappa,\gamma'\rangle}
$$

$$
\gamma \approx_\ell \gamma' \iff \forall x \in \mathrm{dom}(\gamma) \cup \mathrm{dom}(\gamma').\ \gamma(x) \approx_\ell \gamma'(x)
$$

Figure 16: Equal-except-at relation

This definition satisfies the *dependency-correctness* property introduced in [17]. As explained in Cheney et al. [17], dependency-correctness is intuitively motivated by analogy to dependency-tracking and information flow analyses, following the dependency core calculus of Abadi et al. [1]. Analogously to noninterference in information flow security, we define an auxiliary relation $\approx_\ell$, where $v \approx_\ell v'$ intuitively says that two annotated values are equal except (possibly) at parts labeled by ℓ, defined as shown in Figure 16. The $\approx_l$ relation is reflexive, symmetric and transitive, and in particular $c \approx_\ell c$ for any ℓ, since $c = c$, whereas $c^{\ell} \approx_\ell d^{\ell}$ holds even though $c \neq d$, because both are labeled by ℓ.

As discussed in [17], for distinctly-annotated values, $v \approx_\ell v'$ holds if and only if v and v' are of the form $C[v_0]$ and $C[v_0']$, where $C[]$ is a context capturing the common parts of v and v' (above ℓ), and v_0 and v_0' are subvalues showing where v and v' differ (below ℓ). However, during propagation of dependency annotations, values do not remain distinctly-annotated, and the $\approx_\ell$ relation is an appropriate generalization of this property.

Then we can show:

Theorem 4.11. *Suppose $|\widehat{\gamma}|, e \Downarrow v, T$ and $\widehat{\gamma}' \approx_\ell \widehat{\gamma}$ and $|\widehat{\gamma}'|, e \Downarrow v', T'$. Then we have $\mathsf{D}(T,\widehat{\gamma}) \approx_\ell \mathsf{D}(T',\widehat{\gamma}')$.*

Proof. See Appendix A.3. □

This says that the label of a value in the input propagates to all parts of the output where changing the value can have an impact on the result.

Example 4.12. *Revisiting the previous example, dependency-correctness has several implications for output $[2^{L_1,L}, 2^{L_2,L}, 4^{L_3,L}]$. Taking $\ell = L$, dependency-correctness tells us that if the value of y were changed, all three parts of the output list might change. If $\ell = L_1$, then dependency-correctness implies that the output will be of the form $[v, 2, 4]$ for some value v (which by type-safety must be an integer also). Similarly, dependency-correctness implies that if L_2 or L_3 change then the output will be of the form $[2, v, 4]$ or $[2, 2, v]$ respectively.*

Dependency-correctness does not provide a guarantee concerning the effects of multiple, independent changes at different locations (although an approximation of this information is available by using a single location that includes

both changes). Also, in all cases, the structure of the output list cannot change, because we cannot change the length of the input list by changing y or changing the values of elements of the list.

Example 4.13. *Consider a trace $T = \mathtt{fst}(x)$ evaluated in environment $\gamma = [x \mapsto (1^a, 2^b)^c]$, whose result is $\mathsf{D}(T,\gamma) = 1^{D_1(\{a\},\{b\})} = 1^{a,b}$. This can naturally happen if the pair is a part of the (annotated) input, rather than being constructed by the program. This example illustrates that although the annotations of pairs, sum injections, and other constructors are hard-wired to be $\bot$, this does not mean that the binary functions $\mathsf{F}_1, \mathsf{F}_2$ etc. are always called with $\bot$ as the first argument.*

Remark 4.14. *Cheney et al. [17] considered a query language with nested collection types (similar to that used by Buneman et al. [10]). That language included an equality operation at all types, and the dependency provenance semantics for equality made use of another operation $\|\widehat{v}\|$ that collects all of the annotations in $\widehat{v}$. Here, we only consider equality at base types, and so it suffices to consider only local annotations during propagation.*

5 Disclosure and obfuscation analysis

In the previous sections we have defined a trace model for Core TML and defined certain classes of trace queries, provenance views, and introduced technical machinery such as paths and partial values. In this section we put these components to work by investigating the disclosure and obfuscation problems for Core TML traces and provenance views. We confine attention to queries that test properties of the input or output. Investigating queries that capture properties of the trace is more difficult, since traces involve variable binding, whereas for values we have restricted attention to queries formulated in terms of partial values.

5.1 Patterns, partial traces, and trace queries

In section 2, we reviewed and refined a general provenance framework with definitions of disclosure and obfuscation, formulated in terms of abstract sets of traces. We now introduce additional concepts needed to formulate the TML model of provenance as an instance of the abstract provenance framework in section 2, so that we can analyze the security properties of TML-traces. Specifically, we will consider a consistent triple (γ, T, v) as an abstract trace, we will define some provenance queries over such traces, and we will consider some approaches to defining provenance views of the traces. The queries and views rely on notions of patterns and partial traces; specifically, we will consider queries based on testing whether a partial value is present in the input or output, and we will consider views based on deleting information from the trace, input or output.

We introduce patterns for values, environments and traces. The syntax of patterns (pattern environments) is similar to that of values (respectively environments), extended with special *holes*:

$$
\begin{array}{rcl}
p & ::= & c \mid (p_1, p_2) \mid \mathtt{inl}(p) \mid \mathtt{inr}(p) \mid \mathtt{roll}(p) \mid \langle \kappa, \rho \rangle \mid \Diamond \mid \Box \\
\rho & ::= & [x_1 \mapsto p_1, \ldots, x_n \mapsto p_n]
\end{array}
$$

Patterns actually denote binary relations on values. The hole symbol $\Box$ denotes the total relation, while the exact-match symbol $\Diamond$ denotes the identity relation. The $\Diamond$ pattern is a technical device used later in this section in backward disclosure slicing; we sometimes refer to *$\Diamond$-free patterns* that do not contain $\Diamond$.

We say that v matches v' modulo p (written $v \eqsim_p v'$) if v and v' match the structure of p, and are equal at corresponding positions denoted by $\Diamond$. Moreover, we write $p \sqcup p'$ for the least upper bound (join) of two patterns and define $p \sqsubseteq p'$ to hold if $p' = p \sqcup p'$. Rules defining $\eqsim_p$ and $\sqcup$ are given in Figures 17 and 18. In the equations defining $p \sqcup \Diamond$, we use notation $p[\Diamond/\Box]$ to denote the result of replacing all occurrences of $\Box$ in p with $\Diamond$.

When $p \sqsubseteq v$, we write $v|_p$ for the pattern obtained by replacing all of the $\Diamond$-holes in p with the corresponding

$$\boxed{v \eqsim_p v'}$$

$$\frac{}{v \eqsim_\Box v'} \qquad \frac{}{v \eqsim_\Diamond v} \qquad \frac{}{c \eqsim_c c} \qquad \frac{v_1 \eqsim_{p_1} v_1' \quad v_2 \eqsim_{p_2} v_2'}{(v_1, v_2) \eqsim_{(p_1,p_2)} (v_1', v_2')} \qquad \frac{v \eqsim_p v'}{\mathtt{inl}(v) \eqsim_{\mathtt{inl}(p)} \mathtt{inl}(v')} \qquad \frac{v \eqsim_p v'}{\mathtt{inr}(v) \eqsim_{\mathtt{inr}(p)} \mathtt{inr}(v')}$$

$$\frac{v \eqsim_p v'}{\mathtt{roll}(v) \eqsim_{\mathtt{roll}(p)} \mathtt{roll}(v')} \qquad \frac{\gamma \eqsim_\rho \gamma'}{\langle \kappa, \gamma \rangle \eqsim_{\langle \kappa, \rho \rangle} \langle \kappa, \gamma' \rangle}$$

$$\gamma \eqsim_\rho \gamma' \iff \forall x \in \mathrm{dom}(\rho).\ \gamma(x) \eqsim_{\rho(x)} \gamma'(x)$$

Figure 17: Equality modulo patterns.

values in v, defined as follows:

$$\begin{aligned}
v|_\Box &= \Box \\
v|_\Diamond &= v \\
c|_c &= c \\
(v_1, v_2)|_{(p_1,p_2)} &= (v_1|_{p_1}, v_2|_{p_2}) \\
\mathtt{inl}(v)|_{\mathtt{inl}(p)} &= \mathtt{inl}(v|_p) \\
\mathtt{inr}(v)|_{\mathtt{inr}(p)} &= \mathtt{inr}(v|_p) \\
\mathtt{roll}(v)|_{\mathtt{roll}(p)} &= \mathtt{roll}(v|_p) \\
\langle \kappa, \gamma \rangle|_{\langle \kappa, \rho \rangle} &= \langle \kappa, \gamma|_\rho \rangle \\
[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]|_{[x_1 \mapsto p_1, \ldots, x_n \mapsto p_n]} &= [x_1 := v_1|_{p_1}, \ldots, x_n := v_n|_{p_n}]
\end{aligned}$$

For example, $(1, 2)|_{(\Diamond,\Box)} = (1, \Box)$.

Lemma 5.1. *For any v and $p \sqsubseteq v$, we have $v \eqsim_p v|_p$.*

Lemma 5.2. *The set $\{p \mid p \sqsubseteq v\}$ of partial values matching a given value v is an upper semilattice with least element $\Box$, greatest element v and the least upper bound operation $\sqcup$ as defined in Figure 18.*

Lemma 5.3. *For each p, $\eqsim_p$ is a partial equivalence relation. Moreover, whenever $p \sqcup p'$ is defined, we have $(\eqsim_{p \sqcup p'}) = (\eqsim_p) \cap (\eqsim_{p'})$, and whenever $p \sqsubseteq p'$ we have $(\eqsim_p) \supseteq (\eqsim_{p'})$.*

Proof. To show $\eqsim_p$ is a partial equivalence relation, we must show that it is symmetric and transitive (but not necessarily reflexive). Symmetry follows by straightforward induction on derivations. Transitivity is by induction on p. The second part follows by induction on the (partial recursive) definition of $\sqcup$. The third part follows from the second by the fact that $p \sqsubseteq p'$ holds if and only if $p \sqcup p' = p'$. Full details of the proof are in Appendix A.4. □

Lemma 5.4. *If $v \eqsim_p v'$ then $p \sqsubseteq v$ and $p \sqsubseteq v'$. Conversely, if $p \sqsubseteq v$ then $v \eqsim_p v$.*

Proof. By induction on derivations. By symmetry it suffices to show by induction that $v \eqsim_p v'$ implies $p \sqsubseteq v$. The second part is straightforward. □

We also consider partial traces, usually written S, which are trace expressions where some subexpressions have been replaced with $\Box$:

$$S \quad ::= \quad \cdots \mid \Box$$

As with patterns, we write $S \sqsubseteq T$ to indicate that T matches S, that is, S can be made equal to T by filling in some holes.

As mentioned at the beginning of this section, for the purpose of disclosure and obfuscation analysis, we will consider the "traces" to be triples (γ, T, v) where T is consistent with γ and v, that is, $\gamma, T \curvearrowright v$. We refer to such a triple as a *consistent triple*. We consider trace or provenance queries built out of partial values and partial traces.

$$\begin{array}{rcl}
\Box \sqcup p = p \sqcup \Box &=& p \\
\Diamond \sqcup p = p \sqcup \Diamond &=& p[\Diamond/\Box] \\
(p_1, p_2) \sqcup (p_1', p_2') &=& (p_1 \sqcup p_1', p_2 \sqcup p_2') \\
c \sqcup c &=& c \\
\mathtt{inl}(p) \sqcup \mathtt{inl}(p') &=& \mathtt{inl}(p \sqcup p') \\
\mathtt{inr}(p) \sqcup \mathtt{inr}(p') &=& \mathtt{inr}(p \sqcup p') \\
\mathtt{roll}(p) \sqcup \mathtt{roll}(p') &=& \mathtt{roll}(p \sqcup p') \\
\langle \kappa, \rho \rangle \sqcup \langle \kappa, \rho' \rangle &=& \langle \kappa, \rho \sqcup \rho' \rangle \\
(\rho \sqcup \rho')(x) &=& \begin{cases} \rho(x) \sqcup \rho'(x) & x \in \mathrm{dom}(\rho) \cup \mathrm{dom}(\rho') \\ \rho(x) & x \in \mathrm{dom}(\rho) \backslash \mathrm{dom}(\rho') \\ \rho'(x) & x \in \mathrm{dom}(\rho') \backslash \mathrm{dom}(\rho) \end{cases}
\end{array}$$

Figure 18: Least upper bounds of patterns and environments.

Definition 5.5. *1. Let $\phi(\gamma)$ be a predicate on input environments. An* input query $\mathsf{IN}\gamma.\phi(\gamma)$ *is defined as* $\{(\gamma, T, v) \mid \gamma, T \curvearrowright v \text{ and } \phi(\gamma)\}$. *(Here,* IN *binds x in $\phi(x)$.) As a special case, we write IN_ρ for $\mathsf{IN}\gamma.(\rho \sqsubseteq \gamma)$.*

2. Let $\phi(v)$ be a predicate on output values. An output query $\mathsf{OUT}v.\phi(v)$ *is defined as* $\{(\gamma, T, v) \mid \gamma, T \curvearrowright v \text{ and } \phi(v)\}$. *(Here,* OUT *binds x in $\phi(x)$.) As a special case, we write OUT_p for $\mathsf{OUT}v.(p \sqsubseteq v)$.*

Remark 5.6. *The $\mathsf{IN}\gamma.\phi(\gamma)$ and $\mathsf{OUT}v.\phi(v)$ notations are chosen to resemble quantifiers; they should be read as "In the input of the trace, ϕ holds" or "In the output of the trace, ϕ holds". One can also think of them as higher-order functions, for example $\mathsf{IN} : (\mathrm{Env} \to \mathbb{B}) \to \mathcal{T} \to \mathbb{B}$ and $\mathsf{OUT} : (\mathrm{Val} \to \mathbb{B}) \to \mathcal{T} \to \mathbb{B}$, and regard $\mathsf{IN}\gamma.\phi(\gamma)$ and $\mathsf{OUT}v.\phi(v)$ respectively as syntactic sugar for $\mathsf{IN}(\Lambda\gamma.\phi(\gamma))$ and $\mathsf{OUT}(\Lambda v.\phi(v))$.*

To analyze forms of provenance based on annotation (as considered in Section 4) we will also consider consistent annotated triples $(\widehat{\gamma}, T, \widehat{v})$ where $\widehat{v} = \mathsf{F}(T, \widehat{\gamma})$. We will later also consider corresponding queries derived from different forms of provenance, based on annotated triples.

5.2 Disclosure

We first consider properties disclosed by various forms of provenance considered above. Both where-provenance and expression provenance disclose useful information about the input. Dependency provenance does not disclose input information in an easy-to-analyze way, but is useful for obfuscation, as discussed in Section 5.3.

For where-provenance, we consider input queries

$$\mathsf{IN}^{\mathsf{W}}_{v_0,\pi} = \mathsf{IN}\gamma.\ (\gamma[\pi] = v_0))$$

and output queries

$$\mathsf{OUT}^{\mathsf{W}}_{v_0,\pi} = \mathsf{OUT}\hat{v}.\ (w^\pi \in occ(\widehat{v}) \wedge |w| = v_0)$$

where π is a path and v_0 is a value. Such a query tests whether γ or v contains a value v_0 with the provided annotation.

Theorem 5.7. *The where-provenance view $(\gamma, T, v) \mapsto \mathsf{W}(T, \mathsf{path}(\gamma))$ positively discloses $\mathsf{IN}^{\mathsf{W}}_{v_0,\pi}$ via $\mathsf{OUT}^{\mathsf{W}}_{v_0,\pi}$.*

Proof. If $\mathsf{OUT}^{\mathsf{W}}_{v_0,\pi}$ holds of $\mathsf{W}(T, \mathsf{path}(\gamma))$ then by Theorem 4.5 we know that γ contains a copy of v_0 annotated by π, hence $\mathsf{IN}^{\mathsf{W}}_{v_0,\pi}$ holds of $\mathsf{path}(\gamma)$. □

For expression-provenance, suppose t is an expression over paths (that is, locations ℓ in t are paths π). We define $\gamma(t)$ to be the result of evaluating t in γ with all paths π replaced by their values $\gamma[\pi]$ in γ. This is defined as follows:

$$\begin{array}{rcl}
\gamma(x.\pi) &=& \gamma(x)[\pi] \\
\gamma(c) &=& c \\
\gamma(\oplus(t_1, \ldots, t_n)) &=& \hat{\oplus}(\gamma(t_1), \ldots, \gamma(t_n)) \\
\gamma(\bot) &=& \bot
\end{array}$$

We consider queries $\mathsf{IN}^{\mathsf{E}}_{v_0,t} = \mathsf{IN}\gamma.\ (\gamma(t) = v_0)$, where t is an expression provenance annotation and v_0 is a value. Such a query tests whether evaluating an expression t over γ yields the specified value. For example, $\mathsf{IN}\gamma.\ x.1 + y.2 = 4$ holds for $\gamma = [x = (1,2), y = (2,3)]$, because $\gamma(x.1) + \gamma(y.2) = 1 + 3 = 4$. We also consider output queries $\mathsf{OUT}^{\mathsf{E}}_{v_0,t} = \mathsf{OUT}\widehat{v}.\ (w^t \in occ(\widehat{v}) \wedge |w| = v_0)$, that simply test whether an annotated copy of v_0 appears in the output with annotation t.

Theorem 5.8. *The expression-provenance view $(\gamma, T, v) \mapsto \mathsf{E}(T, \mathsf{path}(\gamma))$ positively discloses $\mathsf{IN}^{\mathsf{E}}_{v_0,t}$ via $\mathsf{OUT}^{\mathsf{E}}_{v_0,t}$.*

Proof. Similarly to where-provenance, using Theorem 4.9 we show that if $\mathsf{OUT}^{\mathsf{E}}_{v_0,t}$ holds on $\mathsf{E}(T, \gamma)$ then $\gamma(t) = v_0$, which implies $\mathsf{IN}^{\mathsf{E}}_{v_0,t}$. □

For example, if the annotated output is 42^{x+y}, then we know that the annotated value is equal to the sum of $\gamma(x)$ and $\gamma(y)$, but we do not know anything more about the values of x and y beyond the equation $x + y = 42$. However, if the output is $(42^{x+y}, 17^{x-y})$ then we know that x and y are the (unique) solution to the linear equations

$$\begin{aligned} x + y &= 42 \\ x - y &= 17\,, \end{aligned}$$

that is, $x = 29.5, y = 12.5$.

Expression provenance and where-provenance are also related in the following sense:

Theorem 5.9. *Where-provenance is computable from expression-provenance.*

Proof. Where-provenance annotations can be extracted from expression-provenance annotations by mapping locations ℓ to themselves and all other expressions to $\bot$. □

Hence, any query disclosed by where-provenance is disclosed by expression-provenance, and any query obfuscated by expression-provenance is also obfuscated by where-provenance.

We now consider a form of *trace slicing* that takes a partial output value and removes information from the input and trace that is not needed to disclose part of the output. We show that such *disclosure slices* also disclose generic provenance views (Theorem 5.20). Thus, disclosure slices form a quite general form of provenance in their own right.

Definition 5.10. *Let $\gamma, T \curvearrowright v$, and suppose $S \sqsubseteq T$ and $\rho \sqsubseteq \gamma$. We say (ρ, S) is a* disclosure slice *with respect to partial value p if for all $\gamma' \sqsupseteq \rho$ and $T' \sqsupseteq S$ such that if $\gamma', T' \curvearrowright v'$, we have $p \sqsubseteq v$ iff $p \sqsubseteq v'$.*

The intuition is that a disclosure slice should contain enough information that any replay of a (completed) trace on a (completed) input environment (both extending the respective components of the slice) yields a result that matches p; in other words, the slice is a (possibly smaller) "witness" to the construction of p from the input. Note that by this definition, minimal disclosure slices exist (since there are finitely many slices) but need not be unique. For example, both $\Box \vee \texttt{true}$ and $\texttt{true} \vee \Box$ are disclosure slices showing that $\texttt{true} \vee \texttt{true}$ evaluates to $\texttt{true}$, but $\Box \vee \Box$ is not a disclosure slice.

Figure 19 shows rules defining a disclosure slicing judgment $p, T \xrightarrow{\mathsf{disc}} S, \rho$. Basically, the idea is to push a partial value backwards through a trace to obtain a partial input environment and trace slice. The partial input environment is needed to handle local variables in traces. In the rule for `let`, we first slice through the body of the let, then identify the partial value showing the needed parts of the let-bound value, and use that to slice backwards through the first subtrace.

Example 5.11. *To illustrate the behavior of* `let` *and bound variables, consider:*

$$\dfrac{\dfrac{\dfrac{\dfrac{(\Box,1), x \xrightarrow{\mathsf{disc}} x, [x \mapsto (\Box,1)]}{1, \texttt{snd}(x) \xrightarrow{\mathsf{disc}} \texttt{snd}(x), [x \mapsto (\Box,1)]} \quad \Box, \texttt{fst}(x) \xrightarrow{\mathsf{disc}} \Box, []}{(1,\Box), (\texttt{snd}(x), \texttt{fst}(x)) \xrightarrow{\mathsf{disc}} (\texttt{snd}(x), \Box), [x \mapsto (\Box,1)]} \quad \dfrac{\Box, y \xrightarrow{\mathsf{disc}} \Box, [] \quad 1, z \xrightarrow{\mathsf{disc}} z, [z \mapsto 1]}{(\Box,1), (y,z) \xrightarrow{\mathsf{disc}} (\Box, z), [z \mapsto 1]}}{(1,\Box), \texttt{let}\ x = (y,z)\ \texttt{in}\ (\texttt{snd}(x), \texttt{fst}(x)) \xrightarrow{\mathsf{disc}} \texttt{let}\ x = (\Box, z)\ \texttt{in}\ (\texttt{snd}(x), \Box), [z \mapsto 1]}}{}$$

$$\boxed{p, T \xrightarrow{\mathsf{disc}} S, \rho}$$

$$\frac{}{\Box, T \xrightarrow{\mathsf{disc}} \Box, []} \qquad \frac{}{p, x \xrightarrow{\mathsf{disc}} x, [x \mapsto p]} \qquad \frac{}{c, c \xrightarrow{\mathsf{disc}} c, []} \qquad \frac{}{\langle \kappa, \rho \rangle, \mathtt{fun}\ \kappa \xrightarrow{\mathsf{disc}} \mathtt{fun}\ \kappa, \rho}$$

$$\frac{p_2, T_2 \xrightarrow{\mathsf{disc}} S_2, \rho_2[x \mapsto p_1] \qquad p_1, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1}{p_2, \mathtt{let}\ x = T_1\ \mathtt{in}\ T_2 \xrightarrow{\mathsf{disc}} \mathtt{let}\ x = S_1\ \mathtt{in}\ S_2, \rho_1 \sqcup \rho_2} \qquad \frac{\Diamond, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1 \quad \cdots \quad \Diamond, T_n \xrightarrow{\mathsf{disc}} S_n, \rho_n}{p, \oplus(T_1, \ldots, T_n) \xrightarrow{\mathsf{disc}} \oplus(S_1, \ldots, S_n), \rho_1 \sqcup \cdots \sqcup \rho_n}$$

$$\frac{p_1, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1 \qquad p_2, T_2 \xrightarrow{\mathsf{disc}} S_2, \rho_2}{(p_1, p_2), (T_1, T_2) \xrightarrow{\mathsf{disc}} (S_1, S_2), \rho_1 \sqcup \rho_2} \qquad \frac{(p, \Box), T \xrightarrow{\mathsf{disc}} S, \rho}{p, \mathtt{fst}(T) \xrightarrow{\mathsf{disc}} \mathtt{fst}(S), \rho} \qquad \frac{(\Box, p), T \xrightarrow{\mathsf{disc}} S, \rho}{p, \mathtt{snd}(T) \xrightarrow{\mathsf{disc}} \mathtt{snd}(S), \rho}$$

$$\frac{p, T \xrightarrow{\mathsf{disc}} S, \rho}{\mathtt{inl}(p), \mathtt{inl}(T) \xrightarrow{\mathsf{disc}} \mathtt{inl}(S), \rho} \qquad \frac{p, T \xrightarrow{\mathsf{disc}} S, \rho}{\mathtt{inr}(p), \mathtt{inr}(T) \xrightarrow{\mathsf{disc}} \mathtt{inr}(S), \rho} \qquad \frac{p, T \xrightarrow{\mathsf{disc}} S, \rho}{\mathtt{roll}(p), \mathtt{roll}(T) \xrightarrow{\mathsf{disc}} \mathtt{roll}(S), \rho}$$

$$\frac{\mathtt{roll}(p), T \xrightarrow{\mathsf{disc}} S, \rho}{p, \mathtt{unroll}(T) \xrightarrow{\mathsf{disc}} \mathtt{unroll}(S), \rho} \qquad \frac{p_1, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1[x_1 \mapsto p] \qquad \mathtt{inl}(p), T \xrightarrow{\mathsf{disc}} S, \rho}{p_1, \mathtt{case}\ T \triangleright_{\mathtt{inl}} x_1.T_1 \xrightarrow{\mathsf{disc}} \mathtt{case}\ S \triangleright_{\mathtt{inl}} x_1.S_1, \rho \sqcup \rho_1}$$

$$\frac{p_2, T_2 \xrightarrow{\mathsf{disc}} S_2, \rho_2[x_2 \mapsto p] \qquad \mathtt{inr}(p), T \xrightarrow{\mathsf{disc}} S, \rho}{p_2, \mathtt{case}\ T \triangleright_{\mathtt{inr}} x_2.T_2 \xrightarrow{\mathsf{disc}} \mathtt{case}\ S \triangleright_{\mathtt{inr}} x_2.S_2, \rho \sqcup \rho_2}$$

$$\frac{p, T \xrightarrow{\mathsf{disc}} S, \rho[f \mapsto p_1, x \mapsto p_2] \qquad p_1 \sqcup \langle \kappa, \rho \rangle, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1 \qquad p_2, T_2 \xrightarrow{\mathsf{disc}} S_2, \rho_2}{p, (T_1\ T_2) \triangleright_\kappa f(x).T \xrightarrow{\mathsf{disc}} (S_1\ S_2) \triangleright_\kappa f(x).S, \rho_1 \sqcup \rho_2}$$

$$\frac{fvs(\kappa) = \{x_1, \ldots, x_n\}}{\Diamond, \mathtt{fun}\ \kappa \xrightarrow{\mathsf{disc}} \mathtt{fun}\ \kappa, [x_1 \mapsto \Diamond, \ldots, x_n \mapsto \Diamond]} \qquad \frac{\Diamond, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1 \qquad \Diamond, T_2 \xrightarrow{\mathsf{disc}} S_2, \rho_2}{\Diamond, (T_1, T_2) \xrightarrow{\mathsf{disc}} (S_1, S_2), \rho_1 \sqcup \rho_2} \qquad \frac{\Diamond, T \xrightarrow{\mathsf{disc}} S, \rho}{\Diamond, \mathtt{inl}(T) \xrightarrow{\mathsf{disc}} \mathtt{inl}(S), \rho}$$

$$\frac{\Diamond, T \xrightarrow{\mathsf{disc}} S, \rho}{\Diamond, \mathtt{inr}(T) \xrightarrow{\mathsf{disc}} \mathtt{inr}(S), \rho} \qquad \frac{\Diamond, T \xrightarrow{\mathsf{disc}} S, \rho}{\Diamond, \mathtt{roll}(T) \xrightarrow{\mathsf{disc}} \mathtt{roll}(S), \rho}$$

Figure 19: Disclosure slicing.

Slicing for conditionals (case expressions) follows a similar pattern to let-binding. For example, consider a trace $\mathtt{case}\ T \triangleright_{\mathtt{inl}} x_1.T_1$ indicating that case expression m was executed with argument computed by T, evaluating to some value $\mathtt{inl}(v')$ and the trace of the body of the $\mathtt{inl}$-branch was T_1. If we wish to slice this with respect to p, then we first slice the trace of the taken branch T_1 with respect to T, yielding trace slice S_1 and pattern environment $\rho_1[x_1 \mapsto p]$. Since we know that the result of T must been of the form $\mathtt{inl}(v')$, we slice T with respect to $\mathtt{inl}(p)$, yielding slice S and pattern environment ρ. The final result is slice $\mathtt{case}\ S \triangleright_{\mathtt{inl}} x_1.S_1$ with pattern environment $\rho \sqcup \rho_1$.

Slicing for application traces is similar to slicing for let and case constructs, but more complex due to the need to propagate partial values backwards through closure environments. Specifically, a trace of the form $(T_1\ T_2) \triangleright_\kappa f(x).T$ is sliced with respect to output pattern p as follows. First, T is sliced with respect to p, yielding slice S and partial environment $\rho[f \mapsto p_1, x \mapsto v_2]$. Here, ρ is the part of the closure needed to rerun the body of the function call. We then slice T_1, the subtrace that computed the called function, with respect to $p_1 \sqcup \langle \kappa, \rho \rangle$, since we know from the trace that κ was the called function and we know that the parts of the environment denoted ρ were needed in the call, and we also know that p_1 was needed to apply recursive calls of f. (In particular, if f was not called recursively in T, then $p_1 = \Box$.) This yields a slice S_1 and pattern environment ρ_1. We also slice T_2 with respect to p_2, obtaining a slice S_2 and pattern environment ρ_2 that show what part of the trace and input environment were needed to compute the function argument. The final result is slice $(S_1\ S_2) \triangleright_\kappa f(x).S$ with pattern environment $\rho_1 \sqcup \rho_2$; any dependence on the environment in which the function closure was constructed is propagated to ρ_1 via the slicing subderivation for T_1.

Note also that the special $\Diamond$ patterns are used to slice backwards through primitive operations even when we do not know the values of the inputs or results. This necessitates additional rules that deal with the cases where p is $\Diamond$. Another possibility is to annotate the traces of primitive operations with these values, an approach taken on related

$$
\begin{array}{rcl}
\mathsf{Witness}(p, v) & = & \Box \quad \text{if } p \sqsubseteq v \\
\mathsf{Witness}(c', c) & = & c \\
\mathsf{Witness}((p_1, p_2), (v_1, v_2)) & = & \begin{cases} (\mathsf{Witness}(p_1, v_1), \Box) & \text{if } p_1 \not\sqsubseteq v_1 \\ (\Box, \mathsf{Witness}(p_2, v_2)) & \text{if } p_1 \sqsubseteq v_1 \text{ and } p_2 \not\sqsubseteq v_2 \end{cases} \\
\mathsf{Witness}(p, (v_1, v_2)) & = & (\Box, \Box) \quad \text{otherwise} \\
\mathsf{Witness}(\mathtt{inl}(p), \mathtt{inl}(v)) & = & \mathtt{inl}(\mathsf{Witness}(p, v)) \\
\mathsf{Witness}(p, \mathtt{inl}(v)) & = & \mathtt{inl}(\Box) \quad \text{otherwise} \\
\mathsf{Witness}(\mathtt{inr}(p), \mathtt{inr}(v)) & = & \mathtt{inr}(\mathsf{Witness}(p, v)) \\
\mathsf{Witness}(p, \mathtt{inr}(v)) & = & \mathtt{inr}(\Box) \quad \text{otherwise} \\
\mathsf{Witness}(\mathtt{roll}(p), \mathtt{roll}(v)) & = & \mathtt{roll}(\mathsf{Witness}(p, v)) \\
\mathsf{Witness}(p, \mathtt{roll}(v)) & = & \mathtt{roll}(\Box) \quad \text{otherwise} \\
\mathsf{Witness}(\langle \kappa, \rho \rangle, \langle \kappa, \gamma \rangle) & = & \langle \kappa, [x_1 \mapsto \mathsf{Witness}(\rho(x_1), \gamma(x_1)), \ldots, x_n \mapsto \mathsf{Witness}(\rho(x_n), \gamma(x_n))] \rangle \\
& & \quad \text{where } \mathrm{dom}(\rho) = \mathrm{dom}(\gamma) = \{x_1, \ldots, x_n\} \\
\mathsf{Witness}(\Box, \langle \kappa, \gamma \rangle) & = & \langle \kappa, [] \rangle \quad \text{otherwise}
\end{array}
$$

Figure 20: Witness function

work on using traces for program slicing [40]; however, this approach does not work as well in this setting since our disclosure slicing criterion involves replaying the trace on changed inputs.

Lemma 5.12. *If $\gamma, T \curvearrowright v$ then for any $p \sqsubseteq v$ there exists $S \sqsubseteq T$ and $\rho \sqsubseteq \gamma$ such that $p, T \xrightarrow{\mathsf{disc}} S, \rho$.*

Proof. The first part follows by induction on the structure of the derivation of $\gamma, T \curvearrowright v$. If $p = \Box$ then the conclusion is immediate in any case. For each constructor case (pairs, `inl`, `inr`, `fun`, `roll`), if p is not $\Box$ then its toplevel constructor must match, so we can proceed by induction. The other cases, for primitive operations, pair projection, cases, function application, and unroll, are straightforward because there is no restriction on p (though we need to check that the invariant $p \sqsubseteq v$ holds for the induction hypotheses). □

We define a function $\mathsf{Disc}_p(\gamma, T, v)$ on consistent triples (γ, T, v) as follows.

$$
\mathsf{Disc}_p(\gamma, T, v) = \begin{cases} (\gamma|_\rho, S) & \text{if } p \sqsubseteq v \text{ and } p, T \xrightarrow{\mathsf{disc}} S, \rho \\ (\gamma|_\rho, S) & \text{if } p \not\sqsubseteq v \text{ and } \mathsf{Witness}(p, v), T \xrightarrow{\mathsf{disc}} S, \rho \end{cases}
$$

The idea is that when $p \sqsubseteq v$, we slice using the rules in Figure 19 and then transform ρ by filling in all $\Diamond$-holes with the corresponding values in γ. However, when $p \not\sqsubseteq v$, we do not use p to slice, but instead use $\mathsf{Witness}(p, v)$, a pattern that contains enough of v to show how v fails to match p. The Witness function is defined in Figure 20.

Example 5.13. *Recall the running example $map\ f\ xs$ where*

$$
\gamma = [f \mapsto \mathtt{fun}\ f(x).\mathtt{if}\ x = y\ \mathtt{then}\ y\ \mathtt{else}\ x + 1, xs = [1, 2, 3], y = 2]
$$

and yielding result $v = [2, 2, 4]$. We write $a :: l$ for the list construction operator, that is, $[2, 2, 4] = 2 :: 2 :: 4 :: []$. Let T be the trace obtained by running this example, i.e. $\gamma, map\ f\ xs \Downarrow [2, 2, 4], T$.

- *If $p = [\Box, \Box, \Box]$ then $\mathsf{Disc}_p(\gamma, T) = ([f \mapsto \Box, xs \mapsto [\Box, \Box, \Box], y \mapsto \Box], S)$ where S shows three recursive calls to map, each with a partial trace of f.*
- *If $p = []$ then $\mathsf{Disc}_p(\gamma, T) = ([xs \mapsto \Box :: \Box], S)$ where S shows one recursive call to $map\ f\ xs$ in which xs is inspected and found to be of the form $v :: vs$, the corresponding branch is taken and a nonempty list is constructed.*

- *If* $p = [2, \square, \square]$ *then* $\mathsf{Disc}_p(\gamma, T) = ([f \mapsto \texttt{fun}\ f(x).\texttt{if}\ x = y\ \texttt{then}\ y\ \texttt{else}\ x+1, y \mapsto 2], S)$ *where* S *lists all three calls to* $map\ f\ xs$*, two partial calls to* f *and one complete call to* f *on* 1.
- *If* $p = [\square, 3, \square]$ *then* $\mathsf{Disc}_p(\gamma, T) = ([f \mapsto \texttt{fun}\ f(x).\texttt{if}\ x = y\ \texttt{then}\ y\ \texttt{else}\ x+1, xsmapsto\square :: 2 :: \square, y \mapsto 2], S)$ *where* S *lists two calls to* map*, one partial call to* f *and one complete call to* f *on* 2.

Correctness of disclosure slicing. We now establish the key properties of disclosure slicing, culminating in the main result that Disc_p discloses the output query OUT_p (Theorem 5.19).

Lemma 5.14. *If* $p \not\sqsubseteq v$ *then* $p \not\sqsubseteq \mathsf{Witness}(p, v)$*; moreover, for any* $v' \sqsupseteq \mathsf{Witness}(p, v)$ *we have* $p \not\sqsubseteq v'$.

Proof. The first part follows by induction on the structure of v, with secondary case analysis on the possible forms of p. The second part is immediate. □

The witness function can be replaced by any other function that has this property (for example, we could alter Witness to find a minimum-size pattern witnessing $p \not\sqsubseteq v$.)

Recall the definition of $v \approx_p v'$ as shown in Figure 17. Using this relation, we can prove the correctness of the slicing relation as follows:

Lemma 5.15. *Assume* $\gamma, T \curvearrowright v$ *and* $p, T \xrightarrow{\mathsf{disc}} S, \rho$ *where* $p \sqsubseteq v$*. Then for all* $\gamma' \approx_\rho \gamma$ *and* $T' \sqsupseteq S$*, if* $\gamma', T' \curvearrowright v'$ *then* $v' \approx_p v$.

Proof. See Appendix A.5. □

Correctness follows as a consequence of the above property. To simplify the argument, we prove positive and negative disclosure simultaneously using an auxiliary query on the provenance view. Specifically, we define a function on sliced traces and environments called Replay that, intuitively, computes a plausible output for the slice. Given slice (ρ, S), we define the auxiliary function Replay as follows:

$$\mathsf{Replay}(\rho, S) = choose(\{v' \mid \exists \gamma' \sqsupseteq \rho, T' \sqsupseteq S.\ \gamma', T' \curvearrowright v'\})$$

In other words, $\mathsf{Replay}(\rho, S)$ chooses one of the possible values obtainable by replaying a complete trace extending S on a complete environment extending ρ, if such a value exists; otherwise, the result is arbitrary. Here, $choose : \mathcal{P}(\mathrm{Val}) \to \mathrm{Val}$ is a choice function such that if $X \subseteq \mathrm{Val}$ and $X \neq \emptyset$ then $choose(X) \in X$. If $X = \emptyset$, then $choose(X)$ is an arbitrary value, say 42.

Remark 5.16. *Observe that this is not a constructive definition. We can use the Axiom of Choice to define choose, or define a linear ordering on values to avoid appealing to the Axiom of Choice; however, it is not obvious whether* Replay *itself is computable. In any case,* Replay *is only needed as a technical device to help define the intermediate provenance query used to prove positive and negative disclosure; we never need to try to compute it directly.*

Lemma 5.17. Disc_p *negatively discloses* OUT_p *via* $\mathsf{OUT}_p \circ \mathsf{Replay}$.

Proof. We prove the contrapositive. Suppose $\mathsf{OUT}_p(\gamma, T, v)$ holds, that is, $p \sqsubseteq v$. Then let $(\rho, S) = \mathsf{Disc}_p(\gamma, T, v)$ be the computed slice, where $p, T \xrightarrow{\mathsf{disc}} S, \rho$, and suppose $v' = \mathsf{Replay}(\rho, S)$, where $T' \sqsupseteq S$ and $\gamma' \sqsupseteq \rho$ are the complete trace and environment used by Replay to compute $\gamma', T' \curvearrowright v'$. Thus, we have $\rho \sqsubseteq \gamma$ and $\rho \sqsubseteq \gamma'$, which together with the fact that ρ is $\Diamond$-free (by definition of Disc) implies $\gamma \approx_\rho \gamma'$. By Lemma 5.15 and Lemma 5.1 this implies $v \approx_p v'$ so $p \sqsubseteq v'$. □

Lemma 5.18. Disc_p *positively discloses* OUT_p *via* $\mathsf{OUT}_p \circ \mathsf{Replay}$.

Proof. We prove the contrapositive. Suppose that $\mathsf{OUT}_p(\gamma, T, v)$ fails, that is, $p \not\sqsubseteq v$. We need to show that $\mathsf{OUT}_p(\mathsf{Replay}(\mathsf{Disc}_p(\gamma, T, v)))$ also fails. Since $p \not\sqsubseteq v$, we know that $\mathsf{Disc}_p(\gamma, T, v) = (\rho, S)$ where $\mathsf{Witness}(p, v), T \xrightarrow{\mathsf{disc}} S, \rho$. Thus, $\mathsf{Replay}(\rho, S) = v'$ for some v' obtained by replaying $T' \sqsupseteq S$ and $\gamma' \sqsupseteq \rho$, that is, $\gamma', T' \curvearrowright v'$. Since $\rho \sqsubseteq \gamma$ and ρ is $\Diamond$-free we have that $\gamma \approx_\rho \gamma'$. So, by Lemma 5.15, we know that $v \approx_{\mathsf{Witness}(p,v)} v'$ holds, which implies $\mathsf{Witness}(p, v) \sqsubseteq v'$, and by Lemma 5.14 this implies $p \not\sqsubseteq v'$. This is what we need to show to conclude $\mathsf{OUT}_p(\mathsf{Replay}(\mathsf{Disc}_p(\gamma, T, v))) = 0$. □

Then by Proposition 2.8 and the previous two lemmas we have:

Theorem 5.19. Disc_p *discloses* OUT_p.

This is the main result about disclosure; we previously established some disclosure results for more restricted computational models [15], but this is the first such result for a general-purpose language. It means that the disclosure slicing algorithm can be used to identify a subset of the trace that is large enough to recompute a part of the output, provided the parts of the input specified by ρ remain fixed. As noted elsewhere, this may not be a minimal slice, but it can be much smaller than the original trace. In particular, let (x, e) be a program where e is an arbitrarily complex expression, evaluated in a context with x bound to 42. If all we care about is the first component of the result then the slice with respect to $(\Diamond, \Box)$ is $([x \mapsto 42], (x, \Box))$, which can be arbitrarily smaller than the full trace.

This does not mean that there is no room for improvement in the disclosure slicing algorithm, for example by taking advantage of program analyses that can identify dead code or subprograms whose values are constant: this information can be used to further shrink the trace. Further investigation is needed to experiment with the syntactic disclosure slicing algorithm on realistic examples and identify areas for improvement.

Disclosure from slices. Finally, we link disclosure for value patterns to disclosure for generic provenance views. Essentially, we show that for any F, the disclosure slice for p positively discloses the F-provenance annotations of values matching p. Informally, this means that disclosure slices provide a highly general form of provenance specialized to a part of the output: one can compute and reveal the disclosure slice and others can then compute any generic provenance view from the slice, without rerunning the original computation or consulting input data or subtraces that are dropped in the slice.

To state the desired property, we need to lift $\eqsim_{_}$ to apply to annotated values. The definition is similar to that for unannotated values, with additional rules:

$$\frac{v \eqsim_p w}{v^a \eqsim_p w^a} \qquad \frac{}{v^a \eqsim_\Box w^b} \qquad \frac{}{v^a \eqsim_\Diamond v^a}$$

Theorem 5.20. *Assume* $|\widehat{\gamma}|, T \curvearrowright v$ *and* $p \sqsubseteq v$. *Suppose* $p, T \xrightarrow{\mathsf{disc}} S, \rho$. *Suppose that* F *is a generic extraction function. Then the annotations associated with* p *in* $\mathsf{F}(T, \widehat{\gamma})$ *can be correctly extracted from* S *using only input parts needed by* ρ. *That is, suppose we have* $\widehat{\gamma} \eqsim_\rho \widehat{\gamma}'$ *and* $T' \sqsupseteq S$, *where* $|\widehat{\gamma}'|, T' \curvearrowright v'$. *Then we have* $\mathsf{F}(T, \widehat{\gamma}) \eqsim_p \mathsf{F}(T', \widehat{\gamma}')$.

Proof. Straightforward induction on the structure of derivations of $p, T \xrightarrow{\mathsf{disc}} S, \rho$. See Appendix A.6. $\Box$

Observe that some minimal slices discard information needed for provenance extraction. For example, given expression $\vee(x, \texttt{true})$, the minimal slice with respect to $\texttt{true}$ is $(\Box, \texttt{true})$. However, this slice makes it impossible for dependency or expression provenance extraction to produce the right answer, since in both cases the annotation on x is needed. Moreover, if we ignore code and match pointers, our slicing algorithm appears to be minimal with respect to provenance extraction (that is, removing any more from a trace would produce slices that do not satisfy Theorem 5.20). This supports our view that the trace slicing algorithm is a natural one for the purpose of generating provenance or explanations, despite its non-minimality with respect to the semantic replayability criterion.

An alternative approach to slicing based on a criterion for which minimal slices exist is explored in another recent paper [40]. Intuitively, the difference arises because disclosure slices are defined in terms of a fixed semantics, whereas Perera et al. [40] define a correct backward slice as one that contains enough information to recompute a given part of the output using an ad hoc replay semantics defined over expressions with holes. This makes a nice theory but means that we are required to include information in the slice that is not required in a disclosure slice.

Note that it is typical for a notion oftypical for a notion of witness to lack unique minimal solutions (e.g. why-provenance in databases is defined as the set of minimal witnesses to a query result [11]) and for minimal slices to be non-computable. For example, in the original work on slicing by Weiser [49] minimal slices are shown to exist but are not computable. Similarly, since TML is Turing-complete, it is easy to show that it is undecidable to determine whether a given partial trace is a (minimal) disclosure slice.

5.3 Obfuscation

We now consider obfuscating properties of the input. We first consider what can be obfuscated by the standard provenance views. Where-provenance, essentially, obfuscates anything that can never be copied to the output or affect the control flow of something that is copied to the output. Similarly, expression provenance obfuscates any part of the input that never participates in or influences expression annotations. In both cases, we can potentially learn about parts of the input that affected control flow, however. For example, $\texttt{if}\ x = 1\ \texttt{then}\ 1\ \texttt{else}\ y$ does not obfuscate the value of x in either model, provided y comes from the input, since we can inspect the annotation of the result to determine that $x = 1$ or $x \neq 1$.

This illustrates a possibly counterintuitive fact: obfuscation of the query that tests whether $x = 1$ fails if we can ever learn anything about the result of the query, even if we cannot determine the exact value of x. Thus, where-provenance and expression-provenance do not provide particularly strong obfuscation properties, since they do not take control-flow into account. Given that we want to ensure obfuscation, we consider conservative techniques that accept (or construct) only provenance views that successfully obfuscate, but may reject some views or construct views that are unnecessarily opaque.

There are several ways to erase information from traces (or other provenance views) to ensure obfuscation of input properties. One way is to re-use the static analysis of dependency provenance (in [17], for example) to identify and elide parts of the output that suffice to make it impossible to guess sensitive parts of the input. Alternatively we can use dynamic dependency provenance to increase precision, by propagating dependency tracking information from the input to the output.

This is similar to using static analysis or dynamic labels for information flow security; the difference is one of emphasis. In information flow security, we usually identify high- or low-security locations and try to certify that high-security data does not affect the computation of low-security data; here, instead, we identify a high-security property of the trace (e.g. that the input satisfies a certain formula) and try to determine what parts of the output do not depend on sensitive inputs, and hence can be safely included in the provenance view. However, these techniques do not provide guidance about what parts of the trace can be safely included in the provenance view.

Here, we develop an alternative approach based on directly analyzing and slicing traces. Consider a pattern $\rho \sqsubseteq \gamma$, in which the parts of γ that are considered confidential have been replaced by $\Box$. We construct an *obfuscation slice* by re-evaluating T on ρ as much as possible, to compute a sliced trace S and partial value p. We erase parts of T and of the original output value that depend on the erased parts of ρ. Thus, any part of the trace or output value that remains in the obfuscation slice is irrelevant to the sensitive part of the input, and cannot be used to guess it.

Figure 21 shows a syntactic algorithm for computing obfuscation slices as described above, defined via a judgment $\rho, T \xrightarrow{\mathsf{obf}} p, S$, which takes a partial input environment ρ and trace T as input and computes a partial output p and sliced trace S. Many rules are essentially generalizations of the rules for evaluation to allow for partial inputs, outputs and traces. The rules of interest, near the bottom of the figure, show how to handle attempts to compute that encounter holes in places where a value constructor is expected. When this happens, we essentially propagate the hole result and return a hole trace. This may be unnecessarily draconian for some cases, but is necessary for the case and application traces where the trace form gives clues about the control flow.

Example 5.21. *To illustrate the behavior of $\xrightarrow{\mathsf{obf}}$, consider again a simple program that swaps the elements of a pair:*

$$\dfrac{\dfrac{[z \mapsto 1], y \xrightarrow{\mathsf{obf}} \Box, \Box \qquad [z \mapsto 1], z \xrightarrow{\mathsf{obf}} 1, z}{[z \mapsto 1], (y, z) \xrightarrow{\mathsf{obf}} (\Box, 1), (\Box, z)} \qquad \dfrac{\dfrac{\gamma, x \xrightarrow{\mathsf{obf}} (\Box, 1), x}{\gamma, \texttt{snd}(x) \xrightarrow{\mathsf{obf}} 1, \texttt{snd}(x)} \qquad \dfrac{\gamma, x \xrightarrow{\mathsf{obf}} (\Box, 1), x}{\gamma, \texttt{fst}(x) \xrightarrow{\mathsf{obf}} \Box, \texttt{fst}(x)}}{\gamma, (\texttt{snd}(x), \texttt{fst}(x)) \xrightarrow{\mathsf{obf}} (1, \Box), (\texttt{snd}(x), \texttt{fst}(x))}}{[z \mapsto 1], \texttt{let}\ x = (y, z)\ \texttt{in}\ (\texttt{snd}(x), \texttt{fst}(x)) \xrightarrow{\mathsf{obf}} (1, \Box), \texttt{let}\ x = (\Box, z)\ \texttt{in}\ (\texttt{snd}(x), \texttt{fst}(x))}$$

where $\gamma = [z \mapsto 1, x \mapsto (\Box, 1)]$. *Notice that it is impossible to guess the value of y used in the original run from the slice* $\texttt{let}\ x = (\Box, z)\ \texttt{in}\ (\texttt{snd}(x), \texttt{fst}(x))$ *or partial result* $(1, \Box)$.

Correctness of obfuscation slicing. We now show the correctness of obfuscation slicing in the sense that the slicing algorithm supports positive obfuscation.

$$\boxed{\rho, T \xrightarrow{\mathsf{obf}} p, S}$$

$$\frac{}{\rho, x \xrightarrow{\mathsf{obf}} \rho(x), x} \qquad \frac{}{\rho, c \xrightarrow{\mathsf{obf}} c, c} \qquad \frac{}{\rho, \mathtt{fun}\ \kappa \xrightarrow{\mathsf{obf}} \langle \kappa, \rho \rangle, \mathtt{fun}\ \kappa} \qquad \frac{\rho, T_1 \xrightarrow{\mathsf{obf}} p_1, S_1 \qquad \rho[x \mapsto p_1], T_2 \xrightarrow{\mathsf{obf}} p_2, S_2}{\rho, \mathtt{let}\ x = T_1\ \mathtt{in}\ T_2 \xrightarrow{\mathsf{obf}} p_2, \mathtt{let}\ x = S_1\ \mathtt{in}\ S_2}$$

$$\frac{\rho, T_1 \xrightarrow{\mathsf{obf}} v_1, S_1 \quad \cdots \quad \rho, T_n \xrightarrow{\mathsf{obf}} v_n, S_n}{\rho, \oplus(T_1, \ldots, T_n) \xrightarrow{\mathsf{obf}} \oplus(v_1, \ldots, v_n), \oplus(S_1, \ldots, S_n)} \qquad \frac{\rho, T_1 \xrightarrow{\mathsf{obf}} p_1, S_1 \qquad \rho, T_2 \xrightarrow{\mathsf{obf}} p_2, S_2}{\rho, (T_1, T_2) \xrightarrow{\mathsf{obf}} (p_1, p_2), (S_1, S_2)}$$

$$\frac{\rho, T \xrightarrow{\mathsf{obf}} (p_1, p_2), S}{\rho, \mathtt{fst}(T) \xrightarrow{\mathsf{obf}} p_1, \mathtt{fst}(S)} \qquad \frac{\rho, T \xrightarrow{\mathsf{obf}} (p_1, p_2), S}{\rho, \mathtt{snd}(T) \xrightarrow{\mathsf{obf}} p_2, \mathtt{snd}(S)} \qquad \frac{\rho, T \xrightarrow{\mathsf{obf}} p, S}{\rho, \mathtt{inl}(T) \xrightarrow{\mathsf{obf}} \mathtt{inl}(p), \mathtt{inl}(S)}$$

$$\frac{\rho, T \xrightarrow{\mathsf{obf}} p, S}{\rho, \mathtt{inr}(T) \xrightarrow{\mathsf{obf}} \mathtt{inr}(p), \mathtt{inr}(S)} \qquad \frac{\rho, T \xrightarrow{\mathsf{obf}} \mathtt{inl}(p), S \qquad \rho[x_1 \mapsto p], T_1 \xrightarrow{\mathsf{obf}} p_1, S_1}{\rho, \mathtt{case}\ T \triangleright_{\mathtt{inl}} x_1.T_1 \xrightarrow{\mathsf{obf}} p_1, \mathtt{case}\ S \triangleright_{\mathtt{inl}} x_1.S_1}$$

$$\frac{\rho, T \xrightarrow{\mathsf{obf}} \mathtt{inr}(p), S \qquad \rho[x_2 \mapsto p], T_2 \xrightarrow{\mathsf{obf}} p_2, S_2}{\rho, \mathtt{case}\ T \triangleright_{\mathtt{inr}} x_2.T_2 \xrightarrow{\mathsf{obf}} p_2, \mathtt{case}\ S \triangleright_{\mathtt{inr}} x_2.S_2} \qquad \frac{\rho, T \xrightarrow{\mathsf{obf}} p, S}{\rho, \mathtt{roll}(T) \xrightarrow{\mathsf{obf}} \mathtt{roll}(p), \mathtt{roll}(S)}$$

$$\frac{\rho, T \xrightarrow{\mathsf{obf}} \mathtt{roll}(p), S}{\rho, \mathtt{unroll}(T) \xrightarrow{\mathsf{obf}} p, \mathtt{unroll}(S)} \qquad \frac{\rho, T_1 \xrightarrow{\mathsf{obf}} \langle \kappa, \rho_0 \rangle, S_1 \qquad \rho, T_2 \xrightarrow{\mathsf{obf}} p_2, S_2 \qquad \rho_0[f \mapsto \langle \kappa, \rho_0 \rangle, x \mapsto p_2], T \xrightarrow{\mathsf{obf}} p, S}{\rho, (T_1\ T_2) \triangleright_\kappa f(x).T \xrightarrow{\mathsf{obf}} p, (S_1\ S_2) \triangleright_\kappa f(x).S}$$

$$\frac{\rho, T_i \xrightarrow{\mathsf{obf}} \Box, S_i \qquad (\text{for some } i \in 1, \ldots, n)}{\rho, \oplus(T_1, \ldots, T_n) \xrightarrow{\mathsf{obf}} \Box, \Box} \qquad \frac{\rho, T \xrightarrow{\mathsf{obf}} \Box, S}{\rho, \mathtt{fst}(T) \xrightarrow{\mathsf{obf}} \Box, \Box} \qquad \frac{\rho, T \xrightarrow{\mathsf{obf}} \Box, S}{\rho, \mathtt{snd}(T) \xrightarrow{\mathsf{obf}} \Box, \Box} \qquad \frac{\rho, T \xrightarrow{\mathsf{obf}} \Box, S}{\rho, \mathtt{unroll}(T) \xrightarrow{\mathsf{obf}} \Box, \Box}$$

$$\frac{\rho, T \xrightarrow{\mathsf{obf}} \Box, S}{\rho, \mathtt{case}\ T \triangleright_{\mathtt{inl}} x_1.T_1 \xrightarrow{\mathsf{obf}} \Box, \Box} \qquad \frac{\rho, T \xrightarrow{\mathsf{obf}} \Box, S}{\rho, \mathtt{case}\ T \triangleright_{\mathtt{inr}} x_1.T_1 \xrightarrow{\mathsf{obf}} \Box, \Box} \qquad \frac{\rho, T_1 \xrightarrow{\mathsf{obf}} \Box, S_1}{\rho, (T_1\ T_2) \triangleright_\kappa f(x).T \xrightarrow{\mathsf{obf}} \Box, \Box}$$

Figure 21: Obfuscation slicing.

Lemma 5.22. *If $\gamma, T \curvearrowright v$ and $\rho \sqsubseteq \gamma$ is $\Diamond$-free then there exist unique $p \sqsubseteq v$ and $S \sqsubseteq T$ such that $\rho, T \xrightarrow{\mathsf{obf}} p, S$.*

Proof. First, we show that if $\gamma, T \curvearrowright v$ and $\rho \sqsubseteq \gamma$ is $\Diamond$-free then there exists $p \sqsubseteq v$ and $S \sqsubseteq T$ such that $\rho, T \xrightarrow{\mathsf{obf}} p, S$. Uniqueness is straightforward by induction over derivations of $\rho, T \xrightarrow{\mathsf{obf}} p, S$. $\Box$

Accordingly, we define a partial function $\mathsf{Obf}_\rho(\gamma, T, v)$ as (p, S) where $\rho, T \xrightarrow{\mathsf{obf}} p, S$ and $p \sqsubseteq v$. We can show that this function is total for well-formed, partial traces and $\Diamond$-free input environments.

Example 5.23. *Consider again the running $map\ f\ xs$ example, with γ, T, v as in Example 5.21.*

- *If $\rho = [f \mapsto \mathtt{fun}\ f(x).\mathtt{if}\ x = y\ \mathtt{then}\ y\ \mathtt{else}\ x + 1, xs = [1, 2, 3], y = \Box]$ then $\mathsf{Obf}_\rho(\gamma, T, v) = ([\Box, \Box, \Box], S)$ where S shows three recursive calls to $map\ f$ and three partial calls to f where the parts of the trace showing the results of the conditional tests $x = y$ in f are deleted.*
- *If $\rho = [f \mapsto \Box, xs = [1, 2, 3], y = 2]$ then $\mathsf{Obf}_\rho(\gamma, T, v) = ([\Box, \Box, \Box], S)$ where S shows three recursive calls to $map\ f$ where the traces showing the execution of f are deleted.*
- *If $\rho = [f \mapsto \mathtt{fun}\ f(x).\mathtt{if}\ x = y\ \mathtt{then}\ y\ \mathtt{else}\ x + 1, xs = \Box :: \Box], y = 2]$ then $\mathsf{Obf}_\rho(\gamma, T, v) = (\Box :: \Box, S)$ where S shows two recursive calls to $map\ f$ and one call to f, where information about the control flow branch taken after testing $x = y$ is omitted.*

Lemma 5.24. *If $\gamma, e \Downarrow v, T$ and $\rho \sqsubseteq \gamma$ and $\rho, T \xrightarrow{\mathsf{obf}} p, S$ then for all $\gamma' \sqsupseteq \rho$, if $\gamma', e \Downarrow v', T'$ then $\rho, T' \xrightarrow{\mathsf{obf}} p, S$ and $p \sqsubseteq v'$.*

Proof. See Appendix A.7. $\Box$

Finally, before considering the main correctness result for Obf_ρ, we note a technical issue: In our language, every base type happens to have at least two values, so we can always instantiate a hole at base type in at least two ways. Similarly, pairs, functions and so on involving base types can always be instantiated in several ways. However, in general we cannot assume that every type has more than one ground value. We say that a type is *nonsingular* if it has at least two different values, and in the following result we restrict attention to patterns containing holes of nonsingular types:

Theorem 5.25. *For traces generated by terminating expressions, and ρ with holes of nonsingular types, and $\rho' \sqsupset \rho$, we have Obf_ρ positively obfuscates $\mathsf{IN}_{\rho'}$.*

Proof. Suppose $\mathsf{IN}_{\rho'}$ holds of (γ, T, v) where $\rho' \sqsupset \rho$. Then $\rho \sqsubset \rho' \sqsubseteq \gamma$. Moreover, since the inclusion is strict, and since ρ contains holes of nonsingular type, ρ must contain holes that can be replaced with different values, so there exists another $\gamma' \sqsupseteq \rho$ that differs from ρ'. Since T was generated by a terminating expression, we know that $\gamma', e \Downarrow v', T'$ can be derived for some v', T'. By Lemma 5.24 we know that $\rho, T' \xrightarrow{\mathsf{obf}} p, S$, hence $\mathsf{Obf}_\rho(\gamma', T', v') = (p, S) = \mathsf{Obf}_\rho(\gamma, T, v)$, as required. $\square$

This is the main result about obfuscation. As with disclosure, previously some properties of obfuscation were established for limited computational models [15], but this is the first such result to be established for a general-purpose programming language. This result shows that the obfuscation slicing algorithm (a syntactic traversal of the trace that propagates "holes" forward) provides a safe approach to positive obfuscation. This means that given a pattern ρ' identifying a sensitive part of the input, for any triple (γ, T, v), the syntactic algorithm yields a subtrace (p, S) such that there exists (γ', T', v') whose obfuscation slice is also (p, S) but such that γ' does not contain ρ. Thus, we cannot deduce that ρ' is present in γ' from (p, S).

Negative obfuscation may also hold for the obfuscation slicing algorithm, but if so it appears more difficult to prove: we would have to show that if ρ does not match the input γ, then there is another γ' that does match ρ but produces the same obfuscation slice as ρ. Calculating such a γ' is not straightforward if the expression e can diverge, because even finding a different input on which e terminates is generally an undecidable problem. However, even under an assumption of termination, it is not clear how to compute obfuscation slices to ensure that all traces on inputs that avoid a certain pattern are indistinguishable from traces on inputs that do contain the pattern.

5.4 Discussion

The analysis in section 5.1 gives novel characterizations of what information is disclosed by where-provenance and expression provenance. Essentially, where-provenance discloses information about what parts of the input are copied to the output, while expression provenance additionally discloses information about how parts of the input can be combined to compute parts of the output. Both forms ignore the control flow of the program. The analysis in section 5.1 also shows (in a formal sense) that where-provenance and expression provenance are closely related: one can obtain where-provenance from expressions simply by erasure. Moreover, we can obtain a number of other intermediate provenance models, by extracting information compositionally from expression-provenance annotations.

The disclosure slicing algorithm is based on an interesting insight (which we are exploring in concurrent work on slicing [40]): at a technical level, the information we need for program comprehension via slicing (to understand how a program has evaluated its inputs to produce outputs) is quite similar to what we need for provenance. Our past work on dependency provenance identified connections between provenance and slicing [17] which we have explored in more recent work [40] that employs slicing techniques similar to disclosure slicing.

Obfuscation seems to be fundamentally more difficult to obtain than disclosure. From an intuitive point of view, this is not surprising; however, it is interesting to see where the complications arise at a technical level, and how these interact with conventional forms of provenance. For example, both where- and expression provenance effectively disclose certain information about the output given the input (or vice versa), while dependency provenance does not appear to disclose information in a particularly direct way. On the other hand, since it was inspired in part by information flow security techniques, dependency provenance does seem to obfuscate information about the input, but cannot directly tell us how much of the trace it is safe to provide while still obfuscating a part of the input.

Obfuscation slicing, which is based on a similar idea to dependency provenance, does allow us to provide part of the trace in the provenance view while obfuscating sensitive input. However, we were only able to obtain a positive

obfuscation result for slicing. We do not currently have either a proof of negative obfuscation or a counterexample to it for the obfuscation slicing algorithm. This means that whenever the query actually holds, we cannot be certain of this from the provenance view; however, when the input query fails it may be possible to tell this from the view. Negative obfuscation seems more difficult to prove than positive obfuscation, at least for the input queries we considered. This is unsurprising, since as also found in [15], the definition of obfuscation is more complex.

It is interesting to consider whether alternative definitions of disclosure or obfuscation could lead to more satisfying results. As explained at the end of Section 5.3, the root of the difficulty with negative obfuscation seems to be the difficulty of analyzing the program to find alternative inputs that enable the program to complete and lead to the same obfuscation slice.

One alternative could be to model the knowledge of the attacker about the possible traces more explicitly (e.g. assume the attacker knows the original program). This seems orthogonal to the problem of proving negative obfuscation: it should complicate both positive and negative problems. Another alternative could be to adopt a probabilistic or information-theoretic definition of obfuscation that makes it easier to provide both positive and negative obfuscation. These are possibilities for future work.

6 Related Work

There is a huge, and growing, literature on provenance [8, 18, 44, 38], but there is little work on formal models of provenance and no previous work on provenance in a general-purpose higher-order language. Since we already covered prior work on provenance security in the introduction, we confine our comparison to closely related work on formal techniques for provenance, and on related ideas in programming languages and language-based security.

Provenance. This work differs from previous work on provenance in databases in several important ways. First, we consider a general purpose, higher-order language, whereas previous work considers database query languages of limited expressiveness (e.g., monotone query languages), which include unordered collection types with monadic iteration operations but not sum types, recursive types or first-class functions. Second, we aim to record traces adequate to answer a wide range of provenance queries in this general setting, whereas previous work has focused on particular kinds of queries (e.g., where-provenance [11, 10], why-provenance [11], how-provenance [28, 27]).

Provenance has also been studied extensively for scientific workflow systems [8, 44, 22]. Many workflow provenance systems record additional information to support replaying the computation (analogous to our fidelity property) or provenance queries focusing on explaining parts of the result (analogous to our extraction and slicing techniques). Most work in this area describes the provenance tracking behavior of a system through examples and does not give a formal semantics that could be used to prove correctness properties; furthermore, there has been little work (and there is currently no consensus) on what the appropriate correctness properties are. An exception is Hidders et al. [33], which is the closest workflow provenance work to ours. They model workflows using a core database query language extended with nondeterministic, external function calls, and partially formalize a semantics of *runs*, or sets of triples (γ, e, v) labeling an operational derivation tree. They also discuss extracting *subruns* which seem similar to slices, and extracting provenance information from runs. However, their definitions of subrun and provenance extraction are complex, incomplete, and not accompanied by precise statements or proofs of correctness properties. Further progress on formalizing their approach has been made recently in a workshop paper by Acar et al. [5]; however, provenance extraction and trace slicing are not addressed in [5].

Other related topics. Our trace model is partly inspired by previous work on self-adjusting computation [4], where execution traces are used to efficiently recompute functional programs under arbitrary modifications to their inputs. Previous work on self-adjusting computation has not investigated trace slicing techniques or a relationship between traces and provenance. Unlike self-adjusting traces, our traces are intended as data that can be manipulated and queried by users, with recomputation just one of many competing requirements. Provenance-like ideas have also appeared in the context of *alignment* in bidirectional computation [7] and language-based techniques for *audit* [34, 48] More recently, Dimoulas et al. identified an intriguing connection between provenance and notions of correctness for blame assignment in contracts. They introduce semantic properties that, they suggest, may be related to provenance [25].

However, to our knowledge no formal relationships between provenance and self-adjusting computation, bidirectional computation, or blame have been developed.

Finally, our model of execution traces for TML is closely related to that used in a recent publication [40]; however, the technical contributions, slicing algorithms and the correctness criteria are different. In this paper, we consider trace slicing algorithms that provide disclosure or obfuscation properties, while in [40] we consider trace and program slicing techniques that satisfy a different consistency property, aimed at comparing different runs of a program for debugging or program understanding. At a semantic level, the most important difference is that in this article our definitions are in terms of a standard operational semantics over standard values, which partial and annotated values need to respect; in the work on program slicing we consider a variant operational semantics over partial values (which is similar in some respects to the obfuscation slicing algorithm). Perera et al. [40] make several additional contributions, including algorithms for extracting program slices from trace slices and for constructing *differential slices* that can highlight the exact location of a bug in the source program. Investigating the applicability of these ideas to provenance or provenance security is future work.

7 Conclusions

While the importance of understanding provenance and its security characteristics has been widely documented, to date there has been little work on formal modeling of either provenance or its security. In this article, we elaborate upon the ideas introduced in previous work [15], by instantiating the formal framework proposed there with a general-purpose functional programming language and a natural notion of execution traces. We showed how more conventional forms of provenance can be extracted from such traces via a generic provenance extraction mechanism. Furthermore, we studied the key notions of disclosure and obfuscation in this context. In the process we identified weaker *positive* and *negative* variants of disclosure and obfuscation, based on the observation that the original definitions seem too strong to be satisfied often in practice. Our main results include algorithms for *disclosure slicing*, which traverses a trace backwards to retain information needed to certify how an output was produced, and *obfuscation slicing*, which reruns a trace on partial input (excluding sensitive parts of the input), yielding a partial trace and partial output that excludes all information that could help a principal learn sensitive data.

To summarize, our main contribution is the development of a general model of provenance in the form of a core calculus that instruments runs of programs with detailed execution traces. We validated the design of this calculus by showing that traces generalize other known forms of provenance and by studying their disclosure and obfuscation properties. There are many possible avenues for future work, including:

- identifying richer languages for defining trace queries or provenance views
- developing and implementing practical algorithms for trace slicing, and relating these to program slicing [40]
- developing a more uniform approach to the different forms of replay, extraction, and slicing
- extending trace and provenance models to handle references, exceptions, input/output, concurrency, nondeterminism, communication, etc.

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 and University of Edinburgh are authorized to reproduce and distribute reprints for their purposes 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. Acar is partially supported by an EU ERC grant (2012-StG 308246—DeepSea) and an NSF grant (CCF-1320563).

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL*, pages 147–160, 1999.

[2] U. A. Acar. Self-adjusting computation (an overview). In *PEPM*, pages 1–6, 2009.

[3] U. A. Acar, A. Ahmed, J. Cheney, and R. Perera. A core calculus for provenance. In P. Degano and J. D. Guttman, editors, *POST*, volume 7215 of *LNCS*, pages 410–429. Springer-Verlag, 2012.

[4] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, 2006.

[5] 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`.

[6] B. T. Blaustein, A. Chapman, L. Seligman, M. D. Allen, and A. Rosenthal. Surrogate parenthood: Protected and informative graphs. *PVLDB*, 4(8):518–527, 2011.

[7] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In *POPL*, pages 407–419, New York, NY, USA, 2008. ACM.

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

[9] P. Buneman, J. Cheney, W.-C. Tan, and S. Vansummeren. Curated databases. In *PODS*, pages 1–12, 2008.

[10] 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.

[11] 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.

[12] S. Carey and G. Rogow. UAL shares fall as old story surfaces online. Wall Street Journal, September 2008. `http://online.wsj.com/article/SB122088673738010213.html`.

[13] J. Cheney. Provenance, XML and the Scientific Web. In *PLAN-X*, 2009. Informal proceedings at: `http://db.ucsd.edu/planx2009`.

[14] J. Cheney. Causality and the semantics of provenance. In *Proceedings of the 2010 Workshop on Developments in Computational Models*, 2010.

[15] J. Cheney. A formal framework for provenance security. In *CSF*, pages 281–293. IEEE, 2011.

[16] J. Cheney, U. A. Acar, and A. Ahmed. Provenance traces. *CoRR*, abs/0812.0564, 2008.

[17] J. Cheney, A. Ahmed, and U. A. Acar. Provenance as dependency analysis. *Mathematical Structures in Computer Science*, 21(6):1301–1337, 2011.

[18] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[19] J. Cheney, S. Chong, N. Foster, M. Seltzer, and S. Vansummeren. Provenance: A future history. In *OOPSLA Companion (Onward! 2009)*, pages 957–964, 2009.

[20] 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/`.

[21] 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.

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

[23] S. B. Davidson, S. Khanna, T. Milo, D. Panigrahi, and S. Roy. Provenance views for module privacy. In *PODS*, pages 175–186, 2011.

[24] S. C. Dey, D. Zinn, and B. Ludäscher. ProPub: Towards a declarative approach for publishing customized, policy-aware provenance. In *SSDBM*, pages 225–243, 2011.

[25] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *POPL*, pages 215–226, New York, NY, USA, 2011. ACM.

[26] C. Dwork. A firm foundation for private data analysis. *Commun. ACM*, 54:86–95, January 2011.

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

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

[29] N. Guts, C. Fournet, and F. Z. Nardelli. Reliable evidence: auditability by typing. In *ESORICS*, pages 168–183, 2009.

[30] J. Halpern and J. Pearl. Causes and explanations: A structural-model approach–part I: Causes. *British J. Philos. Sci.*, 56:843–887, 2005.

[31] J. Halpern and J. Pearl. Causes and explanations: A structural-model approach–part II: Explanations. *British J. Philos. Sci.*, 56:889–911, 2005.

[32] R. Hasan, R. Sion, and M. Winslett. Preventing history forgery with secure provenance. *Trans. Storage*, 5:12:1–12:43, December 2009.

[33] 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.

[34] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: a programming language for authorization and audit. In *ICFP*, pages 27–38, New York, NY, USA, 2008.

[35] Z. Liu, S. B. Davidson, and Y. Chen. Generating sound workflow views for correct provenance analysis. *ACM Trans. Database Syst.*, 36(1):6, 2011.

[36] J. Lyle and A. Martin. Trusted computing and provenance: better together. In *Proceedings of the 2nd conference on Theory and practice of provenance (TAPP 2010)*, Berkeley, CA, USA, 2010. USENIX Association.

[37] A. Martin, J. Lyle, and C. Namilkuo. Provenance as a security control. In *TaPP*. USENIX, 2012. Online proceedings: `http://www.usenix.org/system/files/conference/-tapp12/tapp12-final17.pdf`.

[38] L. Moreau. The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2(2–3), 2010.

[39] R. Perera. *Interactive functional programming*. PhD thesis, University of Birmingham, 2013.

[40] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. In *ICFP*, pages 365–376. ACM, 2012.

[41] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[42] A. C. Revkin. Hacked e-mail is new fodder for climate dispute. New York Times, November 20 2009.

[43] S. Schneider. Formal analysis of a non-repudiation protocol. In *Proceedings of the 11th IEEE workshop on Computer Security Foundations*, pages 54–65, Washington, DC, USA, 1998. IEEE Computer Society.

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

[45] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, pages 266–278, 2011.

[46] 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.

[47] 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`.

[48] J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-based audit. In *CSF*, pages 177–191, 2008.

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

[50] J. Zhang, A. Chapman, and K. Lefevre. Do you know where your data's been? — tamper-evident database provenance. In *Proceedings of the 6th VLDB Workshop on Secure Data Management (SDM 2010)*, pages 17–32, Berlin, Heidelberg, 2009. Springer-Verlag.

A Proofs

A.1 Proof of Theorem 4.5

Proof of Theorem 4.5. We prove by induction on the structure of derivations that if $|\widehat{\gamma}|, e \Downarrow v, T$, then $occ^{\not\perp}(\mathsf{W}(T, \widehat{\gamma})) \subseteq occ^{\not\perp}(\widehat{\gamma})$.

- Base cases involving constants and variables are trivial.
- Cases involving constructors (pairs, `inl`, `inr`, `roll`, closures) and primitive operations ($\oplus$) are straightforward since the newly-constructed value is annotated with $\perp$.
- Cases involving pair projections (`fst`, `snd`) and `unroll` are straightforward, since the returned value is a subvalue of the value returned by a subexpression.
- For a derivation of the form:

$$\frac{\mathtt{inl}(x_1).e_1 \in m \qquad |\widehat{\gamma}|, e_1 \Downarrow \mathtt{inl}(v), T \qquad |\widehat{\gamma}|[x_1 \mapsto v], e_1 \Downarrow v_1, T_1}{|\widehat{\gamma}|, \mathtt{case}\ e\ \mathtt{of}\ m \Downarrow v_1, \mathtt{case}\ T \triangleright_{\mathtt{inl}} x_1.T_1}$$

  By induction, $occ^{\not\perp}(\mathsf{W}(T, \widehat{\gamma})) \subseteq occ^{\not\perp}(\widehat{\gamma})$. Moreover, $\mathsf{W}(T, \widehat{\gamma}) = \mathtt{inl}(\widehat{v})^a$ for some $\widehat{v}$ with $|\widehat{v}| = v$. So, $occ^{\not\perp}(\widehat{v}) \subseteq \mathsf{W}(T, \widehat{\gamma}) \subseteq occ^{\not\perp}(\widehat{\gamma})$. Also by induction, $occ^{\not\perp}(\mathsf{W}(T_1, \widehat{\gamma}[x_1 \mapsto \widehat{v}]) \subseteq occ^{\not\perp}(\widehat{\gamma}[x_1 \mapsto \widehat{v}])$. Hence,

$$occ^{\not\perp}(\mathsf{W}(\mathtt{case}\ T \triangleright_{\mathtt{inl}} x_1.T_1, \widehat{\gamma}[x_1 \mapsto \widehat{v}]) = occ^{\not\perp}(\mathsf{W}(T_1, \widehat{\gamma}[x_1 \mapsto \widehat{v}]) \subseteq occ^{\not\perp}(\widehat{\gamma}[x_1 \mapsto \widehat{v}]) = occ^{\not\perp}(\widehat{\gamma}) \cup occ^{\not\perp}(\widehat{v}) \subseteq occ^{\not\perp}(\widehat{\gamma})$$

- The case for `case` where the right branch is taken is symmetric.
- For function application, if the derivation is of the form:

$$\frac{|\widehat{\gamma}|, e_1 \Downarrow \langle f(x).e, \gamma_0\rangle, T_1 \qquad |\widehat{\gamma}|, e_2 \Downarrow v_2, T_2 \qquad \gamma_0[f \mapsto \langle f(x).e, \gamma_0\rangle, x \mapsto v_2], e \Downarrow v, T}{|\widehat{\gamma}|, (e_1\ e_2) \Downarrow v, (T_1\ T_2) \triangleright_{f(x).e} f(x).T}$$

  then by validity we know that $\mathsf{W}(T_1, \widehat{\gamma}) = \langle f(x).e, \widehat{\gamma}_0\rangle^a$ with $|\widehat{\gamma}_0| = \widehat{\gamma}$ and $\mathsf{W}(T_2, \widehat{\gamma}) = \widehat{v}_2$ with $|\widehat{v}_2| = v_2$. By induction, we also know:

$$occ^{\not\perp}(\langle f(x).e, \widehat{\gamma}_0\rangle^a) \subseteq occ^{\not\perp}(\widehat{\gamma}) \qquad occ^{\not\perp}(\widehat{v}_2) \subseteq occ^{\not\perp}(\widehat{\gamma})$$

  Hence,

$$\gamma_0[f \mapsto \langle f(x).e, \gamma_0\rangle, x \mapsto v_2] = |\widehat{\gamma}_0[f \mapsto \langle f(x).e, \widehat{\gamma}_0\rangle^a, x \mapsto \widehat{v}_2]|$$

  So the induction hypothesis applies to the third subderivation, yielding:

$$\mathsf{W}(T, \widehat{\gamma}_0[f \mapsto \langle f(x).e, \widehat{\gamma}_0\rangle^a, x \mapsto \widehat{v}_2]) \subseteq occ^{\not\perp}(\widehat{\gamma}_0[f \mapsto \langle f(x).e, \widehat{\gamma}_0\rangle^a, x \mapsto \widehat{v}_2])$$

  Thus, by the definition of W for application-traces, we have:

$$\begin{array}{rcl} \mathsf{W}((T_1\ T_2) \triangleright_{f(x).e} f(x).T, \widehat{\gamma}) & = & \mathsf{W}(T, \widehat{\gamma}_0[f \mapsto \langle f(x).e, \widehat{\gamma}_0\rangle^a, x \mapsto \widehat{v}_2]) \\ & \subseteq & occ^{\not\perp}(\widehat{\gamma}_0[f \mapsto \langle f(x).e, \widehat{\gamma}_0\rangle^a, x \mapsto \widehat{v}_2]) \\ & = & occ^{\not\perp}(\widehat{\gamma}_0) \cup occ^{\not\perp}(\langle f(x).e, \widehat{\gamma}_0\rangle^a) \cup occ^{\not\perp}(\widehat{v}_2) \\ & \subseteq & occ^{\not\perp}(\widehat{\gamma}) \end{array}$$

□

A.2 Proof of Theorem 4.9

Proof of Theorem 4.9. We prove by induction on the structure of derivations that if $|\widehat{\gamma}|, e \Downarrow v, T$, then for any h consistent with $\widehat{\gamma}$, we have that h is also consistent with $\mathsf{E}(T, \widehat{\gamma})$.

- Base cases involving constants and variables are trivial.
- Cases involving constructors (pairs, `inl`, `inr`, `roll`, closures) are straightforward since the newly-constructed value is annotated with $\bot$.
- For primitive operations, consider a primitive function evaluation:

$$\frac{|\widehat{\gamma}|, e_1 \Downarrow v_1, T_1 \qquad \cdots \qquad |\widehat{\gamma}|, e_n \Downarrow v_n, T_n}{|\widehat{\gamma}|, \oplus(e_1, \ldots, e_n) \Downarrow \hat{\oplus}(v_1, \ldots, v_n), \oplus(T_1, \ldots, T_n)}$$

  Suppose h is consistent with $\widehat{\gamma}$. By induction, h is consistent with $\mathsf{E}(T_i, \widehat{\gamma})$ for each i. This means that each of the results $v_i^{t_i}$ satisfies $h(t_i) = |\mathsf{E}(T_i, \widehat{\gamma})| = v_i$. Thus, $\hat{\oplus}(v_1, \ldots, v_n) = \hat{\oplus}(h(t_1), \ldots, h(t_n)) = h(\oplus(t_1, \ldots, t_n))$, which implies that h is consistent with $\mathsf{E}(T, \widehat{\gamma}) = (\hat{\oplus}(v_1, \ldots, v_n))^{\oplus(t_1, \ldots, t_n)}$, as desired.

- The remaining cases follow similar reasoning to that for where-provenance.
- For a derivation of the form:

$$\frac{\texttt{inl}(x_1).e_1 \in m \qquad |\widehat{\gamma}|, e_1 \Downarrow \texttt{inl}(v), T \qquad |\widehat{\gamma}|[x_1 \mapsto v], e_1 \Downarrow v_1, T_1}{|\widehat{\gamma}|, \texttt{case}\ e\ \texttt{of}\ m \Downarrow v_1, \texttt{case}\ T \triangleright_{\texttt{inl}} x_1.T_1}$$

  Suppose h is consistent with $\widehat{\gamma}$. By induction, h is consistent with $\mathsf{E}(T, \widehat{\gamma})$. Moreover, $\mathsf{E}(T, \widehat{\gamma}) = \texttt{inl}(\widehat{v})^a$ for some $\widehat{v}$ with $|\widehat{v}| = v$. Thus, h is consistent with $\widehat{v}$ and $\widehat{\gamma}[x_1 \mapsto \widehat{v}]$, so by induction, h is consistent with $\mathsf{E}(T_1, \widehat{\gamma}[x_1 \mapsto \widehat{v}])$. Since $\mathsf{E}(\texttt{case}\ T \triangleright_{\texttt{inl}} x_1.T_1, \widehat{\gamma})) = \mathsf{E}(T_1, \widehat{\gamma}[x_1 \mapsto \widehat{v}])$, it follows that h is consistent with $\mathsf{E}(\texttt{case}\ T \triangleright_{\texttt{inl}} x_1.T_1, \widehat{\gamma}))$.

- The case for `case` where the right branch is taken is symmetric.
- For function application, if the derivation is of the form:

$$\frac{|\widehat{\gamma}|, e_1 \Downarrow \langle f(x).e, \gamma_0\rangle, T_1 \qquad |\widehat{\gamma}|, e_2 \Downarrow v_2, T_2 \qquad \gamma_0[f \mapsto \langle f(x).e, \gamma_0\rangle, x \mapsto v_2], e \Downarrow v, T}{|\widehat{\gamma}|, (e_1\ e_2) \Downarrow v, (T_1\ T_2) \triangleright_{f(x).e} f(x).T}$$

  then by validity we know that $\mathsf{E}(T_1, \widehat{\gamma}) = \langle f(x).e, \widehat{\gamma}_0\rangle^a$ with $|\widehat{\gamma}_0| = \widehat{\gamma}$ and $\mathsf{E}(T_2, \widehat{\gamma}) = \widehat{v}_2$ with $|\widehat{v}_2| = v_2$. By induction, we also know h is consistent with $\langle f(x).e, \widehat{\gamma}_0\rangle^a$ and $\widehat{v}_2$, so h is consistent with $\widehat{\gamma}_0[f \mapsto \langle f(x).e, \widehat{\gamma}_0\rangle^a, x \mapsto \widehat{v}_2]$. By induction on the third subderivation, we have that h is consistent with $\mathsf{E}(T, \widehat{\gamma}_0[f \mapsto \langle f(x).e, \widehat{\gamma}_0\rangle^a, x \mapsto \widehat{v}_2])$. To conclude, since

$$\mathsf{E}((T_1\ T_2) \triangleright_{f(x).e} f(x).T, \widehat{\gamma}) = \mathsf{E}(T, \widehat{\gamma}_0[f \mapsto \langle f(x).e, \widehat{\gamma}_0\rangle^a, x \mapsto \widehat{v}_2])\ ,$$

  we know that h is consistent with $\mathsf{E}((T_1\ T_2) \triangleright_{f(x).e} f(x).T, \widehat{\gamma})$.

□

A.3 Proof of Theorem 4.11

Lemma A.1.

1. *If* $\widehat{v}_1 \approx_\ell \widehat{v}_2$ *then* $\widehat{v}_1^{+a} \approx_\ell \widehat{v}_2^{+a}$.
2. *If* $\ell \in a \cap b$ *then* $\widehat{v}_1^{+a} \approx_\ell \widehat{v}_2^{+b}$.

Proof. Similar to a property proved in [17]; the only new cases are for closures, and are straightforward. □

Proof of Theorem 4.11. Proof proceeds by induction on the structure of the derivation of $|\widehat{\gamma}|, e \Downarrow v, T$. There are many straightforward cases, similar to those proved in [17]. We show the proof cases for case and application traces; the other cases use similar techniques.

- If the derivation is of the form:

$$\frac{(\texttt{inl}(x_1).e_1 \in m) \qquad |\widehat{\gamma}|, e \Downarrow \texttt{inl}(v), T \qquad |\widehat{\gamma}|[x_1 \mapsto v], e_1 \Downarrow v_1, T_1}{|\widehat{\gamma}|, \texttt{case}\ e\ \texttt{of}\ m \Downarrow v_1, \texttt{case}\ T \triangleright_{\texttt{inl}} x_1.T_1}$$

  then by inversion of $|\widehat{\gamma}'|, \texttt{case}\ T \triangleright_{\texttt{inl}} x_1.T_1 \curvearrowright v'$ the derivation must be of the form:

$$\frac{|\widehat{\gamma}'|, T \curvearrowright \texttt{inl}(v) \qquad |\widehat{\gamma}'|[x_1 \mapsto v], T_1 \curvearrowright v_1}{|\widehat{\gamma}'|, \texttt{case}\ T \triangleright_{\texttt{inl}} x_1.T_1 \curvearrowright v_1}$$

  Then by validity of D and induction we know that $\texttt{inl}(\widehat{v})^a = \mathsf{D}(T, \widehat{\gamma}) \approx_\ell \mathsf{D}(T', \widehat{\gamma}) = \texttt{inl}(\widehat{v}')^b$, where $|\widehat{v}| = v$ and $|\widehat{v}'| = v'$. Then $|\widehat{\gamma}[x \mapsto \widehat{v}]| = |\widehat{\gamma}|[x \mapsto v]$ and $|\widehat{\gamma}'[x \mapsto \widehat{v}']| = |\widehat{\gamma}'|[x \mapsto v']$. Now, if $\ell \in a \cap b$ then we are done: it follows immediately that

$$\mathsf{D}(\texttt{case}\ T \triangleright_{\texttt{inl}} x_1.T_1, \widehat{\gamma}) = (\widehat{v}_1)^{+a} \approx_\ell (\widehat{v}_1')^{+b} = \mathsf{D}(\texttt{case}\ T' \triangleright_{\texttt{inl}} x_1.T_1', \widehat{\gamma}')$$

  Otherwise, we must have $\widehat{v} \approx_\ell \widehat{v}'$ and $a = b$; hence, $\widehat{\gamma}[x \mapsto \widehat{v}] \approx_\ell \widehat{\gamma}'[x \mapsto \widehat{v}']$. So, by induction, $\widehat{v}_1 = \mathsf{D}(T_1, \widehat{\gamma}[x \mapsto \widehat{v}]) \approx_\ell \mathsf{D}(T_1', \widehat{\gamma}'[x \mapsto \widehat{v}']) = \widehat{v}_1'$. Then it follows immediately that

$$\mathsf{D}(\texttt{case}\ T \triangleright_{\texttt{inl}} x_1.T_1, \widehat{\gamma}) = (\widehat{v}_1)^{+a} \approx_\ell (\widehat{v}_1')^{+a} = \mathsf{D}(\texttt{case}\ T' \triangleright_{\texttt{inl}} x_1.T_1', \widehat{\gamma}')$$

- Application: Suppose the derivation is of the form:

$$\frac{|\widehat{\gamma}|, e_1 \Downarrow \langle \kappa, \gamma_0 \rangle, T_1 \qquad (\kappa = f(x).e) \qquad |\widehat{\gamma}|, e_2 \Downarrow v_2, T_2 \qquad \gamma_0[f \mapsto \langle \kappa, \gamma_0 \rangle, x \mapsto v_2], e \Downarrow v, T}{|\widehat{\gamma}|, (e_1\ e_2) \Downarrow v, (T_1\ T_2) \triangleright_\kappa f(x).T}$$

  By validity we know that $\mathsf{D}(T_1, \widehat{\gamma}) = \langle \kappa, \widehat{\gamma}_0 \rangle^a$ for some $a, \widehat{\gamma}_0$ with $|\widehat{\gamma}_0| = \gamma_0$. Similarly, $\mathsf{D}(T_2, \widehat{\gamma}) = \widehat{v}_2$ for some $\widehat{v}_2$ with $|\widehat{v}_2| = v_2$. Finally, $\mathsf{D}(T, \widehat{\gamma}_0[f \mapsto \langle \kappa, \widehat{\gamma}_0 \rangle^a, x \mapsto \widehat{v}_2]) = \widehat{v}$ for some $\widehat{v}$ with $|\widehat{v}| = v$.

  Then the replay derivation is of the form:

$$\frac{|\widehat{\gamma}'|, T_1 \curvearrowright \langle \kappa, \gamma_0' \rangle \qquad |\widehat{\gamma}'|, T_2 \curvearrowright v_2' \qquad \gamma_0'[f \mapsto \langle \kappa, \gamma_0' \rangle, x \mapsto v_2'], T \curvearrowright v'}{|\widehat{\gamma}'|, (T_1\ T_2) \triangleright_\kappa f(x).T \curvearrowright v'}$$

  First, note that by fidelity and validity $\mathsf{D}(T_1', \widehat{\gamma}') = \langle \kappa, \widehat{\gamma}_0' \rangle^b$ for some $b, \widehat{\gamma}_0'$ with $|\widehat{\gamma}_0'| = \gamma_0'$. Similarly, $\mathsf{D}(T_2', \widehat{\gamma}') = \widehat{v}_2'$ for some $\widehat{v}_2'$ with $|\widehat{v}_2'| = v_2'$. Finally, $\mathsf{D}(T', \widehat{\gamma}_0'[f \mapsto \langle \kappa, \widehat{\gamma}_0' \rangle^b, x \mapsto \widehat{v}_2']) = \widehat{v}'$ for some $\widehat{v}'$ with $|\widehat{v}'| = v'$.

  Then, by induction, we know that

$$\langle \kappa, \widehat{\gamma}_0 \rangle^a = \mathsf{D}(T_1, \widehat{\gamma}) \approx_\ell \mathsf{D}(T_1', \widehat{\gamma}') = \langle \kappa, \widehat{\gamma}_0' \rangle^b \qquad \widehat{v}_2 = \mathsf{D}(T_2, \widehat{\gamma}) \approx_\ell \mathsf{D}(T_2', \widehat{\gamma}') = \widehat{v}_2'$$

  Now, there are two cases. If $\ell \in a \cap b$, then we are done since we can derive:

$$\mathsf{D}((T_1\ T_2) \triangleright_\kappa f(x).T, \widehat{\gamma}) = \widehat{v}^a \approx_\ell (\widehat{v}')^b = \mathsf{D}((T_1'\ T_2') \triangleright_\kappa f(x).T', \widehat{\gamma}')$$

  Otherwise, we know that $a = b$ and $\widehat{\gamma}_0 \approx_\ell \widehat{\gamma}_0'$, hence also:

$$\frac{\widehat{\gamma}_0 \approx_\ell \widehat{\gamma}_0' \qquad \langle \kappa, \widehat{\gamma}_0 \rangle^a \approx_\ell \langle \kappa, \widehat{\gamma}_0' \rangle^b \qquad \widehat{v}_2 \approx_\ell \widehat{v}_2'}{\widehat{\gamma}_0[f \mapsto \langle \kappa, \widehat{\gamma}_0 \rangle^a, x \mapsto \widehat{v}_2] \approx_\ell \widehat{\gamma}_0'[f \mapsto \langle \kappa, \widehat{\gamma}_0' \rangle^a, x \mapsto \widehat{v}_2']}$$

  Then by induction on the remaining subderivations, we have $\widehat{v} \approx_\ell \widehat{v}'$, from which we can infer $\widehat{v}^{+a} \approx_\ell (\widehat{v}')^{+a}$.

□

A.4 Proof of Lemma 5.1

Lemma A.2. *For any p, we have $(\eqsim_{p[\Diamond/\Box]}) = (\eqsim_{\Diamond}) \cap (\eqsim_p)$.*

Proof. Induction on p.

- If $p = \Box$, then the result is immediate since $(\eqsim_{\Diamond}) \cap (\eqsim_{\Box}) = (\eqsim_{\Diamond})$.
- If $p = \Diamond$, then the result is immediate.
- If $p = (p_1, p_2)$, then $p[\Diamond/\Box] = (p_1[\Diamond/\Box], p_2[\Diamond/\Box])$, so we reason as follows:

$$\begin{array}{rcl} (v_1, v_2) \eqsim_{(p_1[\Diamond/\Box], p_2[\Diamond/\Box])} (v_1', v_2') & \iff & v_1 \eqsim_{p_1[\Diamond/\Box]} v_1' \text{ and } v_2 \eqsim_{p_2[\Diamond/\Box]} v_2' \\ & \iff & v_1 \eqsim_{\Diamond} v_1' \text{ and } v_1 \eqsim_{p_1} v_1' \text{ and } v_2 \eqsim_{\Diamond} v_2' \text{ and } v_2 \eqsim_{p_2} v_2' \\ & \iff & (v_1, v_2) \eqsim_{(\Diamond,\Diamond)} (v_1', v_2') \text{ and } (v_1, v_2) \eqsim_{(p_1,p_2)} (v_1', v_2') \\ & \iff & (v_1, v_2) \eqsim_{\Diamond} (v_1', v_2') \text{ and } (v_1, v_2) \eqsim_{(p_1,p_2)} (v_1', v_2') \end{array}$$

- The cases for $p = C(p')$ or $p = \langle \kappa, \rho \rangle$ are similar to the case for pairing.

$\Box$

Proof of Lemma 5.1. Symmetry of $(\eqsim_p)$ follows by straightforward induction on derivations.

We show transitivity by induction on p.

- If $p = \Box$, then transitivity is obvious as $(\eqsim_p)$ is the total relation.
- If $p = \Diamond$, then transitivity is obvious as $(\eqsim_{\Diamond})$ is the identity relation.
- If $p = c$ then suppose $v_1 \eqsim_c v_2$ and $v_2 \eqsim_c v_3$. Then v_1, v_2, v_3 are all constants. By inversion, we must have $v_1 = v_2 = v_3 = c$ so we have $v_1 = c \eqsim_c c = v_3$.
- If $p = C(p')$, then suppose $v_1 \eqsim_{C(p')} v_2$ and $v_2 \eqsim_{C(p')} v_3$. By inversion, we must have

$$\frac{v_1' \eqsim_{p'} v_2'}{C(v_1') \eqsim_{C(p')} C(v_2')} \qquad \frac{v_2' \eqsim_{p'} v_3'}{C(v_2') \eqsim_{C(p')} C(v_3')}$$

  where $v_i = C(v_i')$ for each i. In this case, by induction we have $v_1' \eqsim_{p'} v_3'$ so we can conclude $C(v_1') \eqsim_{C(p')} C(v_3')$ as desired.
- If $p = (p_1, p_2)$ then transitivity follows immediately by induction.
- If $p = \langle \kappa, \rho \rangle$, then the reasoning is similar to that for $p = C(p')$.

We show by induction on pairs (p, p') such that $p \sqcup p'$ exists, that $(\eqsim_{p \sqcup p'}) = (\eqsim_p) \cap (\eqsim_{p'})$.

- If one of the patterns (say, p) is $\Box$ then $\Box \sqcup p' = p'$, so $(\eqsim_{\Box \sqcup p'}) = (\eqsim_{p'}) = (\eqsim_{\Box}) \cap (\eqsim_{p'})$ since $\eqsim_{\Box}$ is total.
- If one of the patterns (say, p) is $\Diamond$ then $\Diamond \sqcup p' = p'[\Diamond/\Box]$, so $(\eqsim_{\Diamond \sqcup p'}) = (\eqsim_{p'[\Diamond/\Box]}) = (\eqsim_{\Diamond}) \cap (\eqsim_{p'})$, where the second equation is Lemma A.2.
- If $(p_1, p_2) \sqcup (p_1', p_2') = (p_1 \sqcup p_1', p_2 \sqcup p_2')$ then

$$\begin{array}{rcl} (v_1, v_2) \eqsim_{(p_1 \sqcup p_1', p_2 \sqcup p_2')} (v_1', v_2') & \iff & v_1 \eqsim_{p_1 \sqcup p_1'} v_1' \text{ and } v_2 \eqsim_{p_2 \sqcup p_2'} v_2' \\ & \iff & v_1 \eqsim_{p_1} v_1' \text{ and } v_1 \eqsim_{p_1'} v_1' \text{ and } v_2 \eqsim_{p_2} v_2' \text{ and } v_2 \eqsim_{p_2'} v_2' \\ & \iff & (v_1, v_2) \eqsim_{(p_1,p_2)} (v_1', v_2') \text{ and } (v_1, v_2) \eqsim_{(p_1',p_2')} (v_1', v_2') \end{array}$$

- For the cases $C(p) \sqcup C(p')$ and $\langle \kappa, \rho \rangle \sqcup \langle \kappa, \rho' \rangle$ the reasoning is similar to the case for pairing.

$\Box$

A.5 Proof of Lemma 5.15

Proof of Lemma 5.15. The proof is by induction on the derivation of $p, T \xrightarrow{\textsf{disc}} S, \rho$ and inversion on $\gamma, T \curvearrowright v$. We use Lemma 5.1 freely without comment to infer, for example, $\gamma \eqsim_{\rho_i} \gamma'$ from $\gamma \eqsim_{\rho_1 \sqcup \rho_2} \gamma'$.

Empty pattern. If the derivation is of the form:

$$\frac{}{\Box, T \xrightarrow{\textsf{disc}} \Box, \Box}$$

then clearly, for any $\gamma' \eqsim_{\Box} \gamma$ and $T' \sqsupseteq \Box$, if $\widehat{\gamma}', T' \curvearrowright v'$ then $v' \eqsim_{\Box} v$.

Variable. If the derivations are of the form:

$$\frac{}{p, x \xrightarrow{\textsf{disc}} x, [x \mapsto p]} \quad \text{and} \quad \frac{}{\gamma, x \curvearrowright \gamma(x)}$$

Then $T' \sqsupseteq x$ implies $T' = x$, so

$$\frac{}{\gamma', x \Downarrow \gamma'(x)}$$

Since by assumption $\gamma' \eqsim_{[x \mapsto p]} \gamma$, we conclude that $\gamma'(x) \eqsim_p \gamma(x)$, as desired.

Let. If the derivations are of the form:

$$\frac{p_2, T_2 \xrightarrow{\textsf{disc}} S_2, \rho_2[x \mapsto p_1] \qquad p_1, T_1 \xrightarrow{\textsf{disc}} S_1, \rho_1}{p_2, \texttt{let}\ x = T_1\ \texttt{in}\ T_2 \xrightarrow{\textsf{disc}} \texttt{let}\ x = S_1\ \texttt{in}\ S_2, \rho_1 \sqcup \rho_2} \quad \text{and} \quad \frac{\gamma, T_1 \curvearrowright v_1 \qquad \gamma[x \mapsto v_1], T_2 \curvearrowright v_2}{\gamma, \texttt{let}\ x = T_1\ \texttt{in}\ T_2 \curvearrowright v_2}$$

Then by inversion we must have $T' = \texttt{let}\ x = T_1'\ \texttt{in}\ T_2'$ where $T_i' \sqsupseteq S_i$, and:

$$\frac{\gamma', T_1' \curvearrowright v_1' \qquad \gamma'[x \mapsto v_1'], T_2' \curvearrowright v_2'}{\gamma', \texttt{let}\ x = T_1'\ \texttt{in}\ T_2' \curvearrowright v_2'}$$

We also know that $\gamma \eqsim_{\rho_1} \gamma'$ and $\gamma \eqsim_{\rho_2} \gamma'$. Then, by induction, we have $v_1 \eqsim_{p_1} v_1'$, hence we know that $\gamma[x \mapsto v_1] \eqsim_{\rho_1[x \mapsto p_1]} \gamma'[x \mapsto v_1']$. So, the induction hypothesis applies to $p_2, T_2 \xrightarrow{\textsf{disc}} S_2, \rho_2[x \mapsto p_1]$, and we can conclude that $v_2 \eqsim_{p_2} v_2'$.

Constant trace. If the trace has the form c, then we have:

$$\frac{}{p, c \xrightarrow{\textsf{disc}} c, \Box} \quad \text{and} \quad \frac{}{\gamma, c \curvearrowright c}$$

Then clearly $\gamma', c \curvearrowright c$ and $c \eqsim_p c$.

Primitives. If the trace has the form $\oplus(\overline{T}) = \oplus(T_1, \ldots, T_n)$, then we have:

$$\frac{\Diamond, T_1 \xrightarrow{\textsf{disc}} S_1, \rho_1 \qquad \ldots \qquad \Diamond, T_n \xrightarrow{\textsf{disc}} S_n, \rho_n}{p, \oplus(T_1, \ldots, T_n) \xrightarrow{\textsf{disc}} \oplus(S_1, \ldots, S_n), \rho_1 \sqcup \cdots \sqcup \rho_n} \quad \text{and} \quad \frac{\gamma, T_1 \curvearrowright v_1 \qquad \ldots \qquad \gamma, T_n \curvearrowright v_n}{\gamma, \oplus(T_1, \ldots, T_n) \curvearrowright \hat{\oplus}(v_1, \ldots, v_n)} .$$

Suppose $T' \sqsupseteq \oplus(S_1, \ldots, S_n)$. This implies $T' = \oplus(T'_1, \ldots, T'_n)$ where $T'_i \sqsupseteq S_i$ for each i. Moreover, by inversion we must have:

$$\frac{\gamma, T'_1 \curvearrowright v'_1 \qquad \ldots \qquad \gamma, T'_n \curvearrowright v'_n}{\gamma, \oplus(T'_1, \ldots, T'_n) \curvearrowright \hat{\oplus}(v'_1, \ldots, v'_n).}$$

By induction, we have for all $1 \leq i \leq n$, that $v'_i \eqsim_{\diamond} v_i$, that is, $v'_i = v_i$. Hence, we can conclude that $\hat{\oplus}(v'_1, \ldots, v'_n) \eqsim_p \hat{\oplus}(v_1, \ldots, v_n)$ since both sides are equal.

Pairs/pair patterns. If the derivation is of the form:

$$\frac{p_1, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1 \qquad p_2, T_2 \xrightarrow{\mathsf{disc}} S_2, \rho_2}{(p_1, p_2), (T_1, T_2) \xrightarrow{\mathsf{disc}} (S_1, S_2), \rho_1 \sqcup \rho_2} \text{ and } \frac{\gamma, T_1 \curvearrowright v_1 \qquad \gamma, T_2 \curvearrowright v_2}{\gamma, (T_1, T_2) \Downarrow (v_1, v_2)}.$$

Then, as in the previous case we know $T' = (T'_1, T'_2)$ where $T'_i \sqsupseteq S_i$ and

$$\frac{\gamma', T'_1 \curvearrowright v'_1 \qquad \gamma', T'_2 \curvearrowright v'_2}{\gamma', (T'_1, T'_2) \curvearrowright (v'_1, v'_2)}.$$

By induction, we have $v_1 \eqsim_{p_1} v'_1$ and $v_2 \eqsim_{p_2} v'_2$, from which it follows that $(v_1, v_2) \eqsim_{(p_1,p_2)} (v'_1, v'_2)$.

First. If the derivation is of the form:

$$\frac{(p, \Box), T \xrightarrow{\mathsf{disc}} S, \rho}{p, \mathtt{fst}(T) \xrightarrow{\mathsf{disc}} \mathtt{fst}(S), \rho} \text{ and } \frac{\gamma, T \curvearrowright (v_1, v_2)}{\gamma, \mathtt{fst}(T) \curvearrowright v_1}$$

then, by inversion, we know that $T' = \mathtt{fst}(T'_0)$ for some $T'_0 \sqsupseteq S$, such that

$$\frac{\gamma', T'_0 \Downarrow (v'_1, v'_2), T''}{\gamma', \mathtt{fst}(T'_0) \Downarrow v'_1, \mathtt{fst}(T'')}$$

By induction, we have $(v'_1, v'_2) \eqsim_{(p,\Box)} (v_1, v_2)$, which implies that $v'_1 \eqsim_p v_1$.

Second. Symmetric to `fst` case.

Inl. Suppose the derivations are of the form:

$$\frac{p, T \xrightarrow{\mathsf{disc}} S, \rho}{\mathtt{inl}(p), \mathtt{inl}(T) \xrightarrow{\mathsf{disc}} \mathtt{inl}(S), \rho} \text{ and } \frac{\gamma, T \curvearrowright v}{\gamma, \mathtt{inl}(T) \curvearrowright \mathtt{inl}(v)}$$

Then we must have $T' = \mathtt{inl}(T'_0)$ where $T'_0 \sqsupseteq S$, and:

$$\frac{\gamma', T'_0 \curvearrowright v'}{\gamma', \mathtt{inl}(T'_0) \curvearrowright \mathtt{inl}(v')}$$

So, by induction, we know that $v \eqsim_p v'$ and so $\mathtt{inl}(v) \eqsim_{\mathtt{inl}(p)} \mathtt{inl}(v')$.

Inr. Symmetric to `inl` case.

Case/L. If the derivations are of the form:

$$\frac{p_1, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1[x_1 \mapsto p] \qquad \mathtt{inl}(p), T \xrightarrow{\mathsf{disc}} S, \rho}{p_1, \mathtt{case}\ T \triangleright_{\mathtt{inl}} x_1.T_1 \xrightarrow{\mathsf{disc}} \mathtt{case}\ S \triangleright_{\mathtt{inl}} x_1.S_1, \rho \sqcup \rho_1}$$

and

$$\frac{\gamma, T \curvearrowright \mathtt{inl}(v) \qquad \gamma[x_1 \mapsto v], T_1 \curvearrowright v_1}{\gamma, \mathtt{case}\ T \triangleright_{\mathtt{inl}} x_1.T_1 \curvearrowright v_1}$$

Then $T' = \mathtt{case}\ T_0' \triangleright_{\mathtt{inl}} x_1.T_1'$ where $T_0' \sqsupseteq S$ and $T_1' \sqsupseteq S_1$. The only way for the replay judgment $\gamma, T' \curvearrowright v'$ to be derived is:

$$\frac{\gamma', T_0' \curvearrowright \mathtt{inl}(v_0') \qquad \gamma[x_1 \mapsto v_0'], T_1' \curvearrowright v'}{\gamma, \mathtt{case}\ T_0' \triangleright_{\mathtt{inl}} x_1.T_1' \curvearrowright v'}$$

so by induction we can conclude $\mathtt{inl}(v) \eqsim_{\mathtt{inl}(p)} \mathtt{inl}(v_0')$, which in turn implies $v \eqsim_p v_0'$. Thus, $\gamma[x \mapsto v] \eqsim_{\rho_1[x \mapsto p]} \gamma'[x \mapsto v_0']$, from which it follows by induction that $v_1 \eqsim_{p_1} v'$.

Case/R. Symmetric to the previous case.

Function abstraction. If the derivations have the form:

$$\frac{}{\langle \kappa, \rho \rangle, \mathtt{fun}\ \kappa \xrightarrow{\mathsf{disc}} \mathtt{fun}\ \kappa, \rho} \quad \text{and} \quad \frac{}{\gamma, \mathtt{fun}\ \kappa \curvearrowright \langle \kappa, \gamma \rangle}$$

then T' must be of the form $\mathtt{fun}\ \kappa$, with derivation:

$$\frac{}{\gamma', \mathtt{fun}\ \kappa \curvearrowright \langle \kappa, \gamma' \rangle}$$

Hence, we can conclude $\langle \kappa, \gamma \rangle \eqsim_{\langle \kappa, \rho \rangle} \langle \kappa, \gamma' \rangle$ immediately from $\gamma \eqsim_\rho \gamma'$.

Application. If the derivations are of the form:

$$\frac{p, T \xrightarrow{\mathsf{disc}} S, \rho[f \mapsto p_1, x \mapsto p_2] \qquad p_1 \sqcup \langle \kappa, \rho \rangle, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1 \qquad p_2, T_2 \xrightarrow{\mathsf{disc}} S_2, \rho_2}{p, (T_1\ T_2) \triangleright_\kappa f(x).T \xrightarrow{\mathsf{disc}} (S_1\ S_2) \triangleright_\kappa f(x).S, \rho_1 \sqcup \rho_2}$$

and

$$\frac{\gamma, T_1 \curvearrowright \langle \kappa, \gamma_0 \rangle \qquad \gamma, T_2 \curvearrowright v_2 \qquad \gamma_0[f \mapsto \langle \kappa, \gamma_0 \rangle, x \mapsto v_2], T \curvearrowright v}{\gamma, (T_1\ T_2) \triangleright_\kappa f(x).T \curvearrowright v}$$

Then we know that $T' = (T_1'\ T_2') \triangleright_\kappa f(x).T_0'$ where $T_1' \sqsupseteq S_1$ and $T_2' \sqsupseteq S_2$ and $T_0' \sqsupseteq S$. The replay derivation of $\gamma', T' \curvearrowright v'$ must be of the form:

$$\frac{\gamma', T_1' \curvearrowright \langle \kappa, \gamma_0' \rangle \qquad \gamma', T_2' \curvearrowright v_2' \qquad \gamma_0'[f \mapsto \langle \kappa, \gamma_0' \rangle, x \mapsto v_2'], T_0' \curvearrowright v'}{\gamma', (T_1'\ T_2') \triangleright_\kappa f(x).T_0' \curvearrowright v'}$$

First, by induction on the first subderivation we know that $\langle \kappa, \gamma_0 \rangle \eqsim_{p_1 \sqcup \langle \kappa, \rho \rangle} \langle \kappa, \gamma_0' \rangle$. Here, recall that p_1 is a value pattern for the function argument obtained from slicing the body. By inversion, we have

$$\langle \kappa, \gamma_0 \rangle \eqsim_{p_1} \langle \kappa, \gamma_0' \rangle \qquad \gamma_0 \eqsim_\rho \gamma_0'$$

By induction, we also have that $v_2 \eqsim_{p_2} v_2'$. Hence, putting the above observations together, we have:

$$\gamma_0[f \mapsto \langle \kappa, \gamma_0 \rangle, x \mapsto v_2] \eqsim_{\rho[f \mapsto p_1, x \mapsto p_2]} \gamma_0'[f \mapsto \langle \kappa, \gamma_0' \rangle, x \mapsto v_2']$$

Thus, the induction hypothesis applies again and we can conclude that $v \eqsim_p v'$.

Roll and unroll. These cases are straightforward, similar to those for pairs and projection.

Pairs/wildcard. If the trace has the form (T_1, T_2), then we have:

$$\frac{\Diamond, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1 \qquad \Diamond, T_2 \xrightarrow{\mathsf{disc}} S_2, \rho_2}{\Diamond, (T_1, T_2) \xrightarrow{\mathsf{disc}} (S_1, S_2), \rho_1 \sqcup \rho_2} \text{ and } \frac{\gamma, T_1 \curvearrowright v_1 \qquad \gamma, T_2 \curvearrowright v_2}{\gamma, (T_1, T_2) \curvearrowright (v_1, v_2)}.$$

Then $T' \sqsupseteq (S_1, S_2)$ so T' must be of the form (T_1', T_2') with $T_1 \sqsupseteq S_1$ and $T_2 \sqsupseteq S_2$, and we must have

$$\frac{\gamma', T_1 \curvearrowright v_1' \qquad \gamma', T_2 \curvearrowright v_2'}{\gamma', (T_1', T_2') \curvearrowright (v_1', v_2')}.$$

By induction, we have $v_1 \eqsim_\Diamond v_1'$ and $v_2 \eqsim_\Diamond v_2'$, which implies $(v_1, v_2) \eqsim_\Diamond (v_1', v_2')$, as required.

Other wildcard cases Other cases involving wildcards are similar to the above. □

A.6 Proof of Theorem 5.20

Proof of Theorem 5.20. The proof is by induction on the structure of the derivation of $p, T \xrightarrow{\mathsf{disc}} S, \rho$. We show that if $|\widehat{\gamma}|, T \curvearrowright |\mathsf{F}(T, \widehat{\gamma})|$ then for any $T', \widehat{\gamma}'$, if $\widehat{\gamma} \eqsim_\rho \widehat{\gamma}'$ and $S \sqsubseteq T'$ and $|\widehat{\gamma}'|, T' \curvearrowright |\mathsf{F}(T', \widehat{\gamma}')|$ then $\mathsf{F}(T, \widehat{\gamma}) \eqsim_p \mathsf{F}(T', \widehat{\gamma}')$.

- If the slicing derivation is of the form:

  $$\frac{}{\Box, T \xrightarrow{\mathsf{disc}} \Box, \Box}$$

  then we are done: the conclusion is trivial since $\mathsf{F}(T, \widehat{\gamma}) \eqsim_\Box \mathsf{F}(T', \widehat{\gamma}')$.

- If the slicing derivation is of the form:

  $$\frac{}{p, x \xrightarrow{\mathsf{disc}} x, [x \mapsto p]}$$

  then we reason as follows:

  $$\mathsf{F}(x, \widehat{\gamma}) = \widehat{\gamma}(x) \eqsim_p \widehat{\gamma}'(x) = \mathsf{F}(x, \widehat{\gamma}')$$

  where $\widehat{\gamma}(x) \eqsim_p \widehat{\gamma}'(x)$ follows from the assumption that $\widehat{\gamma} \eqsim_{[x \mapsto p]} \widehat{\gamma}'$.

- If the slicing derivation is of the form:

  $$\frac{p_2, T_2 \xrightarrow{\mathsf{disc}} S_2, \rho_2[x \mapsto p_1] \qquad p_1, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1}{p_2, \mathtt{let}\ x = T_1\ \mathtt{in}\ T_2 \xrightarrow{\mathsf{disc}} \mathtt{let}\ x = S_1\ \mathtt{in}\ S_2, \rho_1 \sqcup \rho_2}$$

  then let $T_1' \sqsupseteq S_1, T_2' \sqsupseteq S_2$ and $\widehat{\gamma}' \eqsim_\rho \widehat{\gamma}$ be given. By induction and validity we have

  $$\widehat{v}_1 = \mathsf{F}(T_1, \widehat{\gamma}) \eqsim_{p_1} \mathsf{F}(T_1', \widehat{\gamma}') = \widehat{v}_1'$$

  thus, we also know that $\widehat{\gamma}[x \mapsto \widehat{v}_1] \eqsim_{\rho_1[x \mapsto p_1]} \widehat{\gamma}'[x \mapsto \widehat{v}_1']$. So, by induction, we also have:

  $$\widehat{v}_2 = \mathsf{F}(T_2, \widehat{\gamma}[x \mapsto \widehat{v}_1]) \eqsim_{p_2} \mathsf{F}(T_2', \widehat{\gamma}'[x \mapsto \widehat{v}_1']) = \widehat{v}_2'$$

  Thus,

  $$\mathsf{F}(\mathtt{let}\ x = T_1\ \mathtt{in}\ T_2, \widehat{\gamma}) = \widehat{v}_2 \eqsim_{p_2} \widehat{v}_2' = \mathsf{F}(\mathtt{let}\ x = T_1'\ \mathtt{in}\ T_2', \widehat{\gamma}')$$

- If the slicing derivation is of the form:

$$\frac{}{p, c \xrightarrow{\mathsf{disc}} c, \Box}$$

then again we are done as $\mathsf{F}(c, \widehat{\gamma}) = c^{\mathsf{F}_c} = \mathsf{F}(c, \widehat{\gamma}')$.

- If the slicing derivation is of the form:

$$\frac{\Diamond, \overline{T} \xrightarrow{\mathsf{disc}} \overline{S}, \rho}{p, \oplus(\overline{T}) \xrightarrow{\mathsf{disc}} \oplus(\overline{S}), \bigsqcup \overline{\rho}}$$

then let $\widehat{\gamma}', \overline{T'}$ be given with $\widehat{\gamma}' \eqsim_{\bigsqcup \overline{\rho}} \widehat{\gamma}$ and $\overline{T'} \sqsupseteq \overline{S}$. By induction, we know that $\mathsf{F}(T_i, \widehat{\gamma}) \eqsim_{\Diamond} \mathsf{F}(T_i', \widehat{\gamma}')$ (that is, $\mathsf{F}(T_i, \widehat{\gamma}) = \mathsf{F}(T_i', \widehat{\gamma}')$) holds for each i. Let $\widehat{v}_i = \mathsf{F}(T_i, \widehat{\gamma}) = \mathsf{F}(T_i', \widehat{\gamma}')$ for each i. Then we can reason as follows:

$$\mathsf{F}(\oplus(\overline{T}), \widehat{\gamma})) = \hat{\oplus}(|\widehat{v}_1|, \ldots, |\widehat{v}_n|)^{\mathsf{F}_{\oplus}(a_1, \ldots, a_n)} \eqsim_p \hat{\oplus}(|\widehat{v}_1|, \ldots, |\widehat{v}_n|)^{\mathsf{F}_{\oplus}(a_1, \ldots, a_n)} = \mathsf{F}(\oplus(\overline{T'}), \widehat{\gamma}')$$

- If the slicing derivation is of the form:

$$\frac{p_1, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1 \qquad p_2, T_2 \xrightarrow{\mathsf{disc}} S_2, \rho_2}{(p_1, p_2), (T_1, T_2) \xrightarrow{\mathsf{disc}} (S_1, S_2), \rho_1 \sqcup \rho_2}$$

then let γ', T_1', T_2' be given with $T_1' \sqsupseteq S_1, T_2' \sqsupseteq S_2$ and $\widehat{\gamma}' \eqsim_{\rho_1 \sqcup \rho_2} \widehat{\gamma}$. By induction we know that $\mathsf{F}(T_1, \widehat{\gamma}) \eqsim_{p_1} \mathsf{F}(T_1', \widehat{\gamma}')$ and $\mathsf{F}(T_2, \widehat{\gamma}) \eqsim_{p_2} \mathsf{F}(T_2', \widehat{\gamma}')$ so we can conclude:

$$\begin{array}{rcl} \mathsf{F}((T_1, T_2), \widehat{\gamma}) & = & (\mathsf{F}(T_1, \widehat{\gamma}), \mathsf{F}(T_2, \widehat{\gamma}))^{\perp} \\ & \eqsim_{(p_1, p_2)} & (\mathsf{F}(T_1', \widehat{\gamma}'), \mathsf{F}(T_2', \widehat{\gamma}'))^{\perp} \\ & = & \mathsf{F}((T_1', T_2'), \widehat{\gamma}') \end{array}$$

- If the slicing derivation is of the form:

$$\frac{(p, \Box), T \xrightarrow{\mathsf{disc}} S, \rho}{p, \mathtt{fst}(T) \xrightarrow{\mathsf{disc}} \mathtt{fst}(S), \rho}$$

then fix $T' \sqsupseteq S$ and $\widehat{\gamma}' \eqsim_{\rho} \widehat{\gamma}$. If $p = \Box$, then the conclusion is immediate. By induction we know that $\mathsf{F}(T, \rho) \eqsim_{(p, \Box)} \mathsf{F}(T', \rho)$. Then by inversion on the replay derivation and validity we know that

$$\mathsf{F}(T, \rho) = (\widehat{v}_1, \widehat{v}_2)^a \eqsim_{(p, \Box)} (\widehat{v}_1', \widehat{v}_2')^{a'} = \mathsf{F}(T', \widehat{\gamma}')$$

Since $p \neq \Box$, this implies $a = a'$ and $v_1^b = \widehat{v}_1 \eqsim_p \widehat{v}_1' = (v_1')^{b'}$, hence, $v_1 \eqsim_p v_1'$ and $b = b'$. We can conclude by reasoning as follows:

$$\mathsf{F}(\mathtt{fst}(T), \widehat{\gamma}) = v_1^{\mathsf{F}_1(a,b)} \eqsim_p (v_1')^{\mathsf{F}_1(a', b')} = \mathsf{F}(\mathtt{fst}(T'), \rho)$$

- If the slicing derivation is of the form:

$$\frac{(\Box, p), T \xrightarrow{\mathsf{disc}} S, \rho}{p, \mathtt{snd}(T) \xrightarrow{\mathsf{disc}} \mathtt{snd}(S), \rho}$$

then the reasoning is symmetric to the previous case.

- If the slicing derivation is of the form:

$$\frac{p, T \xrightarrow{\textsf{disc}} S, \rho}{\texttt{inl}(p), \texttt{inl}(T) \xrightarrow{\textsf{disc}} \texttt{inl}(S), \rho}$$

then fix $T' \sqsupseteq S$ and $\widehat{\gamma}' \eqsim_\rho \widehat{\gamma}$. By induction we know that $\mathsf{F}(T, \widehat{\gamma}) \eqsim_p \mathsf{F}(T', \widehat{\gamma}')$, so it follows directly that

$$\mathsf{F}(\texttt{inl}(T), \widehat{\gamma}) = \texttt{inl}(\mathsf{F}(T, \widehat{\gamma}))^\perp \eqsim_{\texttt{inl}(p)} \texttt{inl}(\mathsf{F}(T', \widehat{\gamma}'))^\perp = \mathsf{F}(\texttt{inl}(T'), \widehat{\gamma}')$$

- If the slicing derivation is of the form:

$$\frac{p, T \xrightarrow{\textsf{disc}} S, \rho}{\texttt{inr}(p), \texttt{inr}(T) \xrightarrow{\textsf{disc}} \texttt{inr}(S), \rho}$$

then the reasoning is symmetric to the previous case.

- If the slicing derivation is of the form:

$$\frac{p_1, T_1 \xrightarrow{\textsf{disc}} S_1, \rho_1[x_1 \mapsto p] \qquad \texttt{inl}(p), T \xrightarrow{\textsf{disc}} S, \rho}{p_1, \texttt{case}\ T \triangleright_{\texttt{inl}} x_1.T_1 \xrightarrow{\textsf{disc}} \texttt{case}\ S \triangleright_{\texttt{inl}} x_1.S_1, \rho \sqcup \rho_1}$$

then fix $T'_0 \sqsupseteq S$, $T'_1 \sqsupseteq S_1$ and $\widehat{\gamma}' \eqsim_{\rho \sqcup \rho_1} \widehat{\gamma}$. If $p = \Box$ then the conclusion is immediate. By induction we know that $\mathsf{F}(T, \widehat{\gamma}) \eqsim_{\texttt{inl}(p)} \mathsf{F}(T'_0, \widehat{\gamma})$, and by validity this means that $\mathsf{F}(T, \widehat{\gamma}) = (\texttt{inl}(\widehat{v}))^a$ and $\mathsf{F}(T'_0, \widehat{\gamma}') = (\texttt{inl}(\widehat{v}'))^a$ where $\widehat{v} \eqsim_p \widehat{v}'$. Thus, $\widehat{\gamma}[x_1 \mapsto \widehat{v}] \eqsim_{\rho_1[x_1 \mapsto p]} \widehat{\gamma}'[x_1 \mapsto \widehat{v}']$, so by induction we also have

$$v_1^b = \mathsf{F}(T_1, \widehat{\gamma}[x_1 \mapsto \widehat{v}]) \eqsim_{p_1} \mathsf{F}(T'_1, \widehat{\gamma}'[x_1 \mapsto \widehat{v}']) = (v'_1)^{b'} .$$

Moreover, since $p \neq \Box$ we know $b = b'$ and $v_1 \eqsim_{p_1} v'_1$, so:

$$\mathsf{F}(\texttt{case}\ T \triangleright_{\texttt{inl}} x_1.T_1, \widehat{\gamma}) = v_1^{\mathsf{F}_L(a,b)} \eqsim_{p_1} (v'_1)^{\mathsf{F}_L(a,b')} = \mathsf{F}(\texttt{case}\ T'_0 \triangleright_{\texttt{inl}} x_1.T'_1, \widehat{\gamma}')$$

- If the slicing derivation is of the form:

$$\frac{p_2, T_2 \xrightarrow{\textsf{disc}} S_2, \rho_2[x_2 \mapsto p] \qquad \texttt{inr}(p), T \xrightarrow{\textsf{disc}} S, \rho}{p_2, \texttt{case}\ T \triangleright_{\texttt{inr}} x_2.T_2 \xrightarrow{\textsf{disc}} \texttt{case}\ S \triangleright_{\texttt{inr}} x_2.S_2, \rho \sqcup \rho_2}$$

then the reasoning is symmetric to the previous case.

- If the slicing derivation is of the form:

$$\frac{}{\langle \kappa, \rho \rangle, \texttt{fun}\ \kappa \xrightarrow{\textsf{disc}} \texttt{fun}\ \kappa, \rho}$$

then let $T' \sqsupseteq \texttt{fun}\ \kappa$ and $\widehat{\gamma}' \eqsim_\rho \widehat{\gamma}$ be given; note that $T' = \texttt{fun}\ \kappa$. We can conclude immediately that

$$\mathsf{F}(\texttt{fun}\ \kappa, \widehat{\gamma}) = \langle \kappa, \widehat{\gamma} \rangle^\perp \eqsim_\rho \langle \kappa, \widehat{\gamma}' \rangle^\perp = \mathsf{F}(\texttt{fun}\ \kappa, \widehat{\gamma}')$$

- If the slicing derivation is of the form:

$$\frac{p, T \xrightarrow{\textsf{disc}} S, \rho[f \mapsto p_1, x \mapsto p_2] \qquad p_1 \sqcup \langle \kappa, \rho \rangle, T_1 \xrightarrow{\textsf{disc}} S_1, \rho_1 \qquad p_2, T_2 \xrightarrow{\textsf{disc}} S_2, \rho_2}{p, (T_1\ T_2) \triangleright_\kappa f(x).T \xrightarrow{\textsf{disc}} (S_1\ S_2) \triangleright_\kappa f(x).S, \rho_1 \sqcup \rho_2}$$

Let T_1', T_2', T_0' and $\widehat{\gamma}'$ be given with $T_1' \sqsupseteq S_1$, $T_2' \sqsupseteq S_2$ and $T_0' \sqsupseteq S$, and $\widehat{\gamma}' \eqsim_{\rho_1 \sqcup \rho_2} \widehat{\gamma}$. If $p = \Box$, then the conclusion is immediate. Otherwise, by induction, validity, and inversion of $\eqsim_{_}$ derivations, we know that:

$$\begin{array}{rcl} \langle \kappa, \widehat{\gamma}_0 \rangle^a = \mathsf{F}(T_1, \widehat{\gamma}) & \eqsim_{p_1 \sqcup \langle \kappa, \rho \rangle} & \mathsf{F}(T_1', \widehat{\gamma}') = \langle \kappa, \widehat{\gamma}_0' \rangle^a \\ \widehat{v}_2 = \mathsf{F}(T_2, \widehat{\gamma}) & \eqsim_{p_2} & \mathsf{F}(T_2', \widehat{\gamma}') = \widehat{v}_2' \end{array}$$

Thus, we also have $\widehat{\gamma}_0 \eqsim_\rho \widehat{\gamma}_0'$, so we can obtain:

$$\widehat{\gamma}_0[f \mapsto \langle \kappa, \widehat{\gamma}_0 \rangle^a, x \mapsto \widehat{v}_2] \eqsim_{\rho[f \mapsto p_1, x \mapsto p_2]} \widehat{\gamma}_0'[f \mapsto \langle \kappa, \widehat{\gamma}_0' \rangle^a, x \mapsto \widehat{v}_2']$$

By induction, it follows that

$$v^b = \mathsf{F}(T, \widehat{\gamma}_0[f \mapsto \langle \kappa, \widehat{\gamma}_0 \rangle, x \mapsto \widehat{v}_2]) \eqsim_p \mathsf{F}(T_0', \widehat{\gamma}_0'[f \mapsto \langle \kappa, \widehat{\gamma}_0' \rangle, x \mapsto \widehat{v}_2'] = {v'}^{b'}$$

This, together with the fact that $p \neq \Box$, implies that $v \eqsim_p v'$ and $b = b'$, so:

$$\mathsf{F}((T_1\ T_2) \triangleright_\kappa f(x).T, \widehat{\gamma}) = v^{\mathsf{F}_{\mathsf{app}}(a,b)} \eqsim_p (v')^{\mathsf{F}_{\mathsf{app}}(a,b')} = \mathsf{F}((T_1'\ T_2') \triangleright_\kappa f(x).T_0', \widehat{\gamma}')$$

- The cases for `roll` and `unroll` are analogous to the cases for pairing and projection.
- If the slicing derivation is of the form:

$$\frac{\Diamond, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1 \qquad \Diamond, T_2 \xrightarrow{\mathsf{disc}} S_2, \rho_2}{\Diamond, (T_1, T_2) \xrightarrow{\mathsf{disc}} (S_1, S_2), \rho_1 \sqcup \rho_2}$$

  then fix $\widehat{\gamma}', T_1', T_2'$ with $T_i' \sqsupseteq S_i$ and $\widehat{\gamma}' \eqsim_{\rho_1 \sqcup \rho_2} \widehat{\gamma}$. By induction we know that $\mathsf{F}(T_i, \widehat{\gamma}) \eqsim_\Diamond \mathsf{F}(T_i', \widehat{\gamma}')$ (that is, $\mathsf{F}(T_i, \widehat{\gamma}) = \mathsf{F}(T_i', \widehat{\gamma}')$) holds for each $i \in \{1, 2\}$. We reason as follows:

$$\begin{array}{rcl} \mathsf{F}((T_1, T_2), \widehat{\gamma}) & = & (\mathsf{F}(T_1, \widehat{\gamma}), \mathsf{F}(T_2, \widehat{\gamma}))^\perp \\ & = & (\mathsf{F}(T_1', \widehat{\gamma}'), \mathsf{F}(T_2', \widehat{\gamma}'))^\perp \\ & = & \mathsf{F}((T_1', T_2'), \widehat{\gamma}) \end{array}$$

- The other cases in which $p = \Diamond$ are straightforward, following similar reasoning to the above cases where p starts with a value constructor.

$\Box$

A.7 Proof of Lemma 5.24

Proof of Lemma 5.24. The proof is by induction on the structure of derivations of $\rho, T \xrightarrow{\mathsf{obf}} p, S$, and inversion on derivations of $\gamma, e \Downarrow v, T$.

- If the derivations are of the form:

$$\frac{}{\rho, x \xrightarrow{\mathsf{obf}} \rho(x), x} \qquad \frac{}{\gamma, x \Downarrow \gamma(x), x}$$

  then suppose $\gamma' \sqsupseteq \rho$ and $\gamma', x \Downarrow v', T'$. Then $v' = \gamma'(x)$ and $T' = x$, so it suffices to observe that

$$\frac{}{\rho, x \xrightarrow{\mathsf{obf}} \rho(x), x} \qquad \rho(x) \sqsubseteq \gamma'(x)$$

  hold.

- If the derivations are of the form:

$$\frac{}{\rho, c \xrightarrow{\mathsf{obf}} \rho(x), c} \qquad \frac{}{\gamma, c \Downarrow c, c}$$

then suppose $\gamma' \sqsupseteq \rho$ where $\gamma', c \Downarrow v', T'$. By inversion the only way the latter can be derived is if $v' = c$ and $T' = c$. So we can conclude by observing:

$$\frac{}{\rho, c \xrightarrow{\mathsf{obf}} c, c} \qquad c \sqsubseteq c$$

- If the derivations are of the form:

$$\frac{}{\rho, \texttt{fun}\ \kappa \xrightarrow{\mathsf{obf}} \langle \kappa, \rho \rangle, \texttt{fun}\ \kappa} \qquad \frac{}{\gamma, \texttt{fun}\ \kappa \Downarrow \langle \kappa, \gamma \rangle, \texttt{fun}\ \kappa}$$

then suppose $\gamma' \sqsupseteq \rho$ is given where $\gamma', \texttt{fun}\ \kappa \Downarrow v', T'$. By inversion we must have $v' = \langle \kappa, \gamma' \rangle$ and $T' = \texttt{fun}\ \kappa$. Thus, we can conclude by observing:

$$\frac{}{\rho, \texttt{fun}\ \kappa \xrightarrow{\mathsf{obf}} \langle \kappa, \rho \rangle, \texttt{fun}\ \kappa} \qquad \langle \kappa, \rho \rangle \sqsubseteq \langle \kappa, \gamma' \rangle$$

- If the derivations are of the form:

$$\frac{\rho, T_1 \xrightarrow{\mathsf{obf}} p_1, S_1 \qquad \rho[x \mapsto p_1], T_2 \xrightarrow{\mathsf{obf}} p_2, S_2}{\rho, \texttt{let}\ x = T_1\ \texttt{in}\ T_2 \xrightarrow{\mathsf{obf}} p_2, \texttt{let}\ x = S_1\ \texttt{in}\ S_2} \qquad \frac{\gamma, e_1 \Downarrow v_1, T_1 \qquad \gamma[x \mapsto v_1], e_2 \Downarrow v_2, T_2}{\gamma, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2, \texttt{let}\ x = T_1\ \texttt{in}\ T_2}$$

then suppose $\gamma' \sqsubseteq \rho$ and $\gamma', \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v', T'$. By inversion, this derivation is of the form:

$$\frac{\gamma', e_1 \Downarrow v_1', T_1' \qquad \gamma'[x \mapsto v_1'], e_2 \Downarrow v_2', T_2'}{\gamma', \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2', \texttt{let}\ x = T_1'\ \texttt{in}\ T_2'}$$

and $v' = v_2'$ and $T' = \texttt{let}\ x = T_1'\ \texttt{in}\ T_2'$. By induction, we have $\rho, T_1' \xrightarrow{\mathsf{obf}} p_1, S_1$ and $p_1 \sqsubseteq v_1'$. Thus, $\gamma'[x \mapsto v_1'] \sqsupseteq \rho[x \mapsto p_1]$, so by induction, we have $\rho[x \mapsto p_1], T_2' \xrightarrow{\mathsf{obf}} p_2, S_2$ where $p_2 \sqsubseteq v_2'$. To conclude, we have:

$$\frac{\rho, T_1' \xrightarrow{\mathsf{obf}} p_1, S_1 \qquad \rho[x \mapsto p_1], T_2' \xrightarrow{\mathsf{obf}} p_2, S_2}{\rho, \texttt{let}\ x = T_1'\ \texttt{in}\ T_2' \xrightarrow{\mathsf{obf}} p_2, \texttt{let}\ x = S_1\ \texttt{in}\ S_2} \qquad p_2 \sqsubseteq v_2'$$

- If the derivations are of the form:

$$\frac{\rho, T_1 \xrightarrow{\mathsf{obf}} v_1, S_1 \quad \cdots \quad \rho, T_n \xrightarrow{\mathsf{obf}} v_n, S_n}{\rho, \oplus(T_1, \ldots, T_n) \xrightarrow{\mathsf{obf}} \oplus(v_1, \ldots, v_n), \oplus(S_1, \ldots, S_n)} \qquad \frac{\gamma, e_1 \Downarrow v_1, T_1 \quad \cdots \quad \gamma, e_n \Downarrow v_n, T_n}{\gamma, \oplus(e_1, \ldots, e_n) \Downarrow \hat{\oplus}(v_1, \ldots, v_n), \oplus(T_1, \ldots, T_n)}$$

then suppose $\gamma' \sqsupseteq \rho$ and $\gamma', \oplus(e_1, \ldots, e_n) \Downarrow v', T'$. By inversion the derivation must be of the form:

$$\frac{\gamma', e_1 \Downarrow v_1', T_1' \quad \cdots \quad \gamma', e_n \Downarrow v_n', T_n'}{\gamma', \oplus(e_1, \ldots, e_n) \Downarrow \hat{\oplus}(v_1', \ldots, v_n'), \oplus(T_1', \ldots, T_n')}$$

where $v' = \hat{\oplus}(v_1', \ldots, v_n')$ and $T' = \oplus(T_1', \ldots, T_n')$. By induction, we know that for each i, $\rho, T_i' \xrightarrow{\mathsf{obf}} v_i, S_i$ and $v_i \sqsubseteq v_i'$. The latter implies $v_i = v_i'$ since v_i is a constant value. Thus, we can conclude:

$$\frac{\rho, T_1' \xrightarrow{\mathsf{obf}} v_1, S_1 \quad \cdots \quad \rho, T_n' \xrightarrow{\mathsf{obf}} v_n, S_n}{\rho, \oplus(T_1', \ldots, T_n') \xrightarrow{\mathsf{obf}} \oplus(v_1, \ldots, v_n), \oplus(S_1, \ldots, S_n)} \qquad \hat{\oplus}(v_1, \ldots, v_n) = \hat{\oplus}(v_1', \ldots, v_n') = v'$$

- If the derivations are of the form:

$$\frac{\rho, T_i \xrightarrow{\mathsf{obf}} \square, S_i \qquad (\text{for some } i \in 1, \dots, n)}{\rho, \oplus(T_1, \dots, T_n) \xrightarrow{\mathsf{obf}} \square, \square} \quad \frac{\gamma, e_1 \Downarrow v_1, T_1 \qquad \cdots \qquad \gamma, e_n \Downarrow v_n, T_n}{\gamma, \oplus(e_1, \dots, e_n) \Downarrow \hat{\oplus}(v_1, \dots, v_n), \oplus(T_1, \dots, T_n)}$$

then suppose $\gamma' \sqsupseteq \rho$ and $\gamma', \oplus(e_1, \dots, e_n) \Downarrow v', T'$. By inversion the derivation must be of the form:

$$\frac{\gamma', e_1 \Downarrow v_1', T_1' \qquad \cdots \qquad \gamma', e_n \Downarrow v_n', T_n'}{\gamma', \oplus(e_1, \dots, e_n) \Downarrow \hat{\oplus}(v_1', \dots, v_n'), \oplus(T_1', \dots, T_n')}$$

where $v' = \hat{\oplus}(v_1', \dots, v_n')$ and $T' = \oplus(T_1', \dots, T_n')$. By induction, we know that $\rho, T_i' \xrightarrow{\mathsf{obf}} \square, S_i$ and $\square \sqsubseteq v_i'$. Thus, we can conclude:

$$\frac{\rho, T_i' \xrightarrow{\mathsf{obf}} \square, S_i \qquad (\text{for some } i \in 1, \dots, n)}{\rho, \oplus(T_1', \dots, T_n') \xrightarrow{\mathsf{obf}} \square, \square} \qquad \square \sqsubseteq \hat{\oplus}(v_1', \dots, v_n') = v'$$

- Suppose the derivations are of the form:

$$\frac{\rho, T_1 \xrightarrow{\mathsf{obf}} p_1, S_1 \qquad \rho, T_2 \xrightarrow{\mathsf{obf}} p_2, S_2}{\rho, (T_1, T_2) \xrightarrow{\mathsf{obf}} (p_1, p_2), (S_1, S_2)} \quad \frac{\gamma, e_1 \Downarrow v_1, T_1 \qquad \gamma, e_2 \Downarrow v_2, T_2}{\gamma, (e_1, e_2) \Downarrow (v_1, v_2), (T_1, T_2)}$$

and suppose $\gamma' \sqsupseteq \rho$ is given, where $\gamma', (e_1, e_2) \Downarrow v', T'$. By inversion, the derivation must have the form:

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

so $v' = (v_1', v_2')$ and $T' = (T_1', T_2')$. By induction we have $\rho, T_1' \xrightarrow{\mathsf{obf}} p_1, S_1$ and $p_1 \sqsubseteq v_1'$ and $\rho, T_2' \xrightarrow{\mathsf{obf}} p_2, S_2$ and $p_2 \sqsubseteq v_2'$. So we can conclude:

$$\frac{\rho, T_1' \xrightarrow{\mathsf{obf}} p_1, S_1 \qquad \rho, T_2' \xrightarrow{\mathsf{obf}} p_2, S_2}{\rho, (T_1', T_2') \xrightarrow{\mathsf{obf}} (p_1, p_2), (S_1, S_2)} \qquad (p_1, p_2) \sqsubseteq (v_1', v_2')$$

- Suppose the derivations are of the form:

$$\frac{\rho, T \xrightarrow{\mathsf{obf}} (p_1, p_2), S}{\rho, \mathtt{fst}(T) \xrightarrow{\mathsf{obf}} p_1, \mathtt{fst}(S)} \quad \frac{\gamma, e \Downarrow (v_1, v_2), T}{\gamma, \mathtt{fst}(e) \Downarrow v_1, \mathtt{fst}(T)}$$

and suppose $\gamma' \sqsupseteq \rho$ is given, where $\gamma', \mathtt{fst}(e) \Downarrow v', T'$. By inversion, the derivation must have the form:

$$\frac{\gamma', e \Downarrow (v_1', v_2'), T'}{\gamma', \mathtt{fst}(e) \Downarrow v_1', \mathtt{fst}(T')}$$

so $v' = v_1'$ and $T' = \mathtt{fst}(T')$. By induction, we know that $\rho, T' \xrightarrow{\mathsf{obf}} (p_1, p_2), S$ where $(p_1, p_2) \sqsubseteq (v_1', v_2')$. So we can conclude:

$$\frac{\rho, T' \xrightarrow{\mathsf{obf}} (p_1, p_2), S}{\rho, \mathtt{fst}(T') \xrightarrow{\mathsf{obf}} p_1, \mathtt{fst}(S)} \quad p_1 \sqsubseteq v_1'$$

- Suppose the derivations are of the form:

$$\frac{\rho, T \xrightarrow{\mathsf{obf}} \Box, S}{\rho, \mathtt{fst}(T) \xrightarrow{\mathsf{obf}} \Box, \Box} \quad \frac{\gamma, e \Downarrow (v_1, v_2), T}{\gamma, \mathtt{fst}(e) \Downarrow v_1, \mathtt{fst}(T)}$$

and suppose $\gamma' \sqsupseteq \rho$ is given, where $\gamma', \mathtt{fst}(e) \Downarrow v', T'$. By inversion, the derivation must have the form:

$$\frac{\gamma', e \Downarrow (v_1', v_2'), T'}{\gamma', \mathtt{fst}(e) \Downarrow v_1', \mathtt{fst}(T')}$$

so $v' = v_1'$ and $T' = \mathtt{fst}(T')$. By induction, we know that $\rho, T' \xrightarrow{\mathsf{obf}} (p_1, p_2), S$ where $\Box \sqsubseteq (v_1', v_2')$. So we can conclude:

$$\frac{\rho, T' \xrightarrow{\mathsf{obf}} \Box, S}{\rho, \mathtt{fst}(T') \xrightarrow{\mathsf{obf}} \Box, \Box} \quad \Box \sqsubseteq v_1'$$

- The cases for $\mathtt{snd}(e)$ are symmetric.
- Suppose the derivations are of the form:

$$\frac{\rho, T \xrightarrow{\mathsf{obf}} p, S}{\rho, \mathtt{inl}(T) \xrightarrow{\mathsf{obf}} \mathtt{inl}(p), \mathtt{inl}(S)} \quad \frac{\gamma, e \Downarrow v, T}{\gamma, \mathtt{inl}(e) \Downarrow \mathtt{inl}(v), \mathtt{inl}(T)}$$

and suppose $\gamma' \sqsupseteq \rho$ is given, where $\gamma', \mathtt{inl}(e) \Downarrow v', T'$. By inversion, the derivation must have the form:

$$\frac{\gamma', e \Downarrow v_0', T_0'}{\gamma', \mathtt{inl}(e) \Downarrow \mathtt{inl}(v'), \mathtt{inl}(T_0')}$$

so $v' = \mathtt{inl}(v_0')$ and $T' = \mathtt{inl}(T_0')$. By induction we have $\rho, T_0' \xrightarrow{\mathsf{obf}} p, S$ and $p \sqsubseteq v_0'$. So we can conclude:

$$\frac{\rho, T' \xrightarrow{\mathsf{obf}} p, S}{\rho, \mathtt{inl}(T') \xrightarrow{\mathsf{obf}} \mathtt{inl}(p), \mathtt{inl}(S)} \quad \mathtt{inl}(p) \sqsubseteq \mathtt{inl}(v_0')$$

- The case for $\mathtt{inl}(e)$ is symmetric.
- Suppose the derivations are of the form:

$$\frac{\rho, T \xrightarrow{\mathsf{obf}} \mathtt{inl}(p), S \qquad \rho[x_1 \mapsto p], T_1 \xrightarrow{\mathsf{obf}} p_1, S_1}{\rho, \mathtt{case}\ T \rhd_{\mathtt{inl}} x_1.T_1 \xrightarrow{\mathsf{obf}} p_1, \mathtt{case}\ S \rhd_{\mathtt{inl}} x_1.S_1}$$

$$\frac{(\mathtt{inl}(x_1).e_1 \in m) \qquad \gamma, e \Downarrow \mathtt{inl}(v), T \qquad \gamma[x_1 \mapsto v], e_1 \Downarrow v_1, T_1}{\gamma, \mathtt{case}\ e\ \mathtt{of}\ m \Downarrow v_1, \mathtt{case}\ T \rhd_{\mathtt{inl}} x_1.T_1}$$

and suppose $\gamma' \sqsupseteq \rho$ is given, where $\gamma', \mathtt{case}\ e\ \mathtt{of}\ m \Downarrow v', T'$. By inversion of this derivation, there are two cases. If the derivation is of the form:

$$\frac{(\mathtt{inl}(x_1).e_1 \in m) \qquad \gamma', e \Downarrow \mathtt{inl}(v_0'), T_0' \qquad \gamma'[x_1 \mapsto v_0'], e_1 \Downarrow v_1', T_1'}{\gamma', \mathtt{case}\ e\ \mathtt{of}\ m \Downarrow v_1', \mathtt{case}\ T' \rhd_{\mathtt{inl}} x_1.T_1'}$$

then $v' = v_1'$ and $T' = \mathtt{case}\ T' \rhd_{\mathtt{inl}} x_1.T_1'$. By induction, we have $\rho, T_0' \xrightarrow{\mathsf{obf}} \mathtt{inl}(p), S$ and $\mathtt{inl}(p) \sqsubseteq \mathtt{inl}(v_0')$, so $p \sqsubseteq v_0'$ and $\gamma'[x \mapsto v_0'] \sqsupseteq \rho[x \mapsto p]$. Again by induction, we have $\rho[x \mapsto p], T_1' \xrightarrow{\mathsf{obf}} p_1, S_1$ where $p_1 \sqsubseteq v_1'$. So we can conclude:

$$\frac{\rho, T' \xrightarrow{\mathsf{obf}} \mathtt{inl}(p), S \qquad \rho[x_1 \mapsto p], T_1' \xrightarrow{\mathsf{obf}} p_1, S_1}{\rho, \mathtt{case}\ T' \rhd_{\mathtt{inl}} x_1.T_1' \xrightarrow{\mathsf{obf}} p_1, \mathtt{case}\ S \rhd_{\mathtt{inl}} x_1.S_1} \quad p_1 \sqsubseteq v_1'$$

If the derivation of $\gamma', \texttt{case}\ e\ \texttt{of}\ m \Downarrow v', T'$ is of the form

$$\frac{(\texttt{inr}(x_2).e_2 \in m) \qquad \gamma, e \Downarrow \texttt{inr}(v), T \qquad \gamma[x_2 \mapsto v], e_2 \Downarrow v_2, T_2}{\gamma, \texttt{case}\ e\ \texttt{of}\ m \Downarrow v_2, \texttt{case}\ T \rhd_{\texttt{inr}} x_2.T_2}$$

then by induction we can derive $\rho, T' \xrightarrow{\textsf{obf}} \texttt{inl}(p), S$ and $\texttt{inl}(p) \sqsubseteq \texttt{inr}(v_0')$, which is absurd, so this case is vacuous.

- Suppose the derivations are of the form:

$$\frac{\rho, T \xrightarrow{\textsf{obf}} \Box, S}{\rho, \texttt{case}\ T \rhd_{\texttt{inl}} x_1.T_1 \xrightarrow{\textsf{obf}} \Box, \Box} \qquad \frac{(\texttt{inl}(x_1).e_1 \in m) \qquad \gamma, e \Downarrow \texttt{inl}(v), T \qquad \gamma[x_1 \mapsto v], e_1 \Downarrow v_1, T_1}{\gamma, \texttt{case}\ e\ \texttt{of}\ m \Downarrow v_1, \texttt{case}\ T \rhd_{\texttt{inl}} x_1.T_1}$$

and suppose $\gamma' \sqsupseteq \rho$ is given, where $\gamma', \texttt{case}\ e\ \texttt{of}\ m \Downarrow v', T'$. By inversion of this derivation, there are two cases. If the derivation is of the form:

$$\frac{(\texttt{inl}(x_1).e_1 \in m) \qquad \gamma', e \Downarrow \texttt{inl}(v_0'), T_0' \qquad \gamma'[x_1 \mapsto v_0'], e_1 \Downarrow v_1', T_1'}{\gamma', \texttt{case}\ e\ \texttt{of}\ m \Downarrow v_1', \texttt{case}\ T' \rhd_{\texttt{inl}} x_1.T_1'}$$

then $v' = v_1'$ and $T' = \texttt{case}\ T' \rhd_{\texttt{inl}} x_1.T_1'$. By induction, we know that $\rho, T_0' \xrightarrow{\textsf{obf}} \Box, S$ and $\Box \sqsubseteq \texttt{inl}(v_0')$, so we can conclude:

$$\frac{\rho, T_0' \xrightarrow{\textsf{obf}} \Box, S}{\rho, \texttt{case}\ T_0' \rhd_{\texttt{inl}} x_1.T_1' \xrightarrow{\textsf{obf}} \Box, \Box} \quad \Box \sqsubseteq v_1'$$

If the derivation of $\gamma', \texttt{case}\ e\ \texttt{of}\ m \Downarrow v', T'$ is of the form

$$\frac{(\texttt{inr}(x_2).e_2 \in m) \qquad \gamma, e \Downarrow \texttt{inr}(v_0'), T_0' \qquad \gamma[x_2 \mapsto v], e_2 \Downarrow v_2', T_2'}{\gamma, \texttt{case}\ e\ \texttt{of}\ m \Downarrow v_2', \texttt{case}\ T' \rhd_{\texttt{inr}} x_2.T_2'}$$

then the same reasoning applies: by induction on the first subderivation we can obtain $\rho, T_0' \xrightarrow{\textsf{obf}} \Box, S$ and $\Box \sqsubseteq \texttt{inr}(v_0')$, and conclude:

$$\frac{\rho, T_0' \xrightarrow{\textsf{obf}} \Box, S}{\rho, \texttt{case}\ T_0' \rhd_{\texttt{inr}} x_1.T_1' \xrightarrow{\textsf{obf}} \Box, \Box} \quad \Box \sqsubseteq v_2'$$

- The cases for $\texttt{case}\ T \rhd_{\texttt{inr}} x_2.T_2$ are symmetric.
- If the derivations are of the form:

$$\frac{\rho, T_1 \xrightarrow{\textsf{obf}} \langle \kappa, \rho_0 \rangle, S_1 \qquad \rho, T_2 \xrightarrow{\textsf{obf}} p_2, S_2 \qquad \rho_0[f \mapsto \langle \kappa, \rho_0 \rangle, x \mapsto p_2], T \xrightarrow{\textsf{obf}} p, S}{\rho, (T_1\ T_2) \rhd_\kappa f(x).T \xrightarrow{\textsf{obf}} p, (S_1\ S_2) \rhd_\kappa f(x).S}$$

$$\frac{\gamma, e_1 \Downarrow \langle \kappa, \gamma_0 \rangle, T_1 \qquad (\kappa = f(x).e) \qquad \gamma, e_2 \Downarrow v_2, T_2 \qquad \gamma_0[f \mapsto \langle \kappa, \gamma_0 \rangle, x \mapsto v_2], e \Downarrow v, T}{\gamma, (e_1\ e_2) \Downarrow v, (T_1\ T_2) \rhd_\kappa f(x).T}$$

then suppose $\gamma' \sqsupseteq \rho$ and $\gamma', (e_1\ e_2) \Downarrow v', T'$. Then by inversion the derivation must have the form:

$$\frac{\gamma', e_1 \Downarrow \langle \kappa', \gamma_0' \rangle, T_1' \qquad (\kappa' = f(x).e') \qquad \gamma', e_2 \Downarrow v_2', T_2' \qquad \gamma_0'[f \mapsto \langle \kappa', \gamma_0' \rangle, x \mapsto v_2'], e' \Downarrow v', T_0'}{\gamma', (e_1\ e_2) \Downarrow v', (T_1'\ T_2') \rhd_{\kappa'} f(x).T_0'}$$

so $T' = (T_1'\ T_2') \rhd_{\kappa'} f(x).T_0'$. By induction, we know that $\rho, T_1' \xrightarrow{\textsf{obf}} \langle \kappa, \rho_0 \rangle, S_1$ and $\langle \kappa, \rho_0 \rangle \sqsubseteq \langle \kappa', \gamma_0' \rangle$, from which it follows that $\kappa = \kappa'$ and $\gamma_0' \sqsupseteq \rho_0$. Similarly, by induction we know that $\rho, T_2' \xrightarrow{\textsf{obf}} p_2, S_2$. Furthermore,

note that $\rho_0[f \mapsto \langle \kappa, \rho_0 \rangle, x \mapsto p_2] \sqsubseteq \gamma'_0[f \mapsto \langle \kappa, \gamma'_0 \rangle, x \mapsto v'_2]$, and since $\kappa = \kappa'$, we have $e = e'$ so by induction on the third subderivation we have

$$\rho_0[f \mapsto \langle \kappa, \rho_0 \rangle, x \mapsto p_2], T'_0 \xrightarrow{\mathsf{obf}} p, S$$

and $p \sqsubseteq v'$. To conclude, we have:

$$\frac{\rho, T'_1 \xrightarrow{\mathsf{obf}} \langle \kappa, \rho_0 \rangle, S_1 \qquad \rho, T'_2 \xrightarrow{\mathsf{obf}} p_2, S_2 \qquad \rho_0[f \mapsto \langle \kappa, \rho_0 \rangle, x \mapsto p_2], T'_0 \xrightarrow{\mathsf{obf}} p, S}{\rho, (T'_1\ T'_2) \triangleright_\kappa f(x).T'_0 \xrightarrow{\mathsf{obf}} p, (S_1\ S_2) \triangleright_\kappa f(x).S} \qquad p \sqsubseteq v'$$

as desired.

- If the derivations are of the form:

$$\frac{\rho, T_1 \xrightarrow{\mathsf{obf}} \Box, S_1}{\rho, (T_1\ T_2) \triangleright_\kappa f(x).T \xrightarrow{\mathsf{obf}} \Box, \Box}$$

$$\frac{\gamma, e_1 \Downarrow \langle \kappa, \gamma_0 \rangle, T_1 \qquad (\kappa = f(x).e) \qquad \gamma, e_2 \Downarrow v_2, T_2 \qquad \gamma_0[f \mapsto \langle \kappa, \gamma_0 \rangle, x \mapsto v_2], e \Downarrow v, T}{\gamma, (e_1\ e_2) \Downarrow v, (T_1\ T_2) \triangleright_\kappa f(x).T}$$

then suppose $\gamma' \sqsupseteq \rho$ and $\gamma', (e_1\ e_2) \Downarrow v', T'$. Then by inversion the derivation must have the form:

$$\frac{\gamma', e_1 \Downarrow \langle \kappa', \gamma'_0 \rangle, T'_1 \qquad (\kappa' = f(x).e') \qquad \gamma', e_2 \Downarrow v'_2, T'_2 \qquad \gamma'_0[f \mapsto \langle \kappa', \gamma'_0 \rangle, x \mapsto v'_2], e' \Downarrow v', T'_0}{\gamma', (e_1\ e_2) \Downarrow v', (T'_1\ T'_2) \triangleright_{\kappa'} f(x).T'_0}$$

so $T' = (T'_1\ T'_2) \triangleright_{\kappa'} f(x).T'_0$. By induction, we know that $\rho, T'_1 \xrightarrow{\mathsf{obf}} \Box, S_1$ and $\Box \sqsubseteq \langle \kappa', \gamma'_0 \rangle$. To conclude, we have:

$$\frac{\rho, T'_1 \xrightarrow{\mathsf{obf}} \Box, S_1}{\rho, (T'_1\ T'_2) \triangleright_\kappa f(x).T'_0 \xrightarrow{\mathsf{obf}} \Box, \Box} \quad \Box \sqsubseteq v'$$

as desired.

- If the derivations are of the form:

$$\frac{\rho, T \xrightarrow{\mathsf{obf}} p, S}{\rho, \mathtt{roll}(T) \xrightarrow{\mathsf{obf}} \mathtt{roll}(p), \mathtt{roll}(S)} \quad \frac{\gamma, e \Downarrow v, T}{\gamma, \mathtt{roll}(e) \Downarrow \mathtt{roll}(v), \mathtt{roll}(T)}$$

then let $\gamma' \sqsupseteq \rho$ be given, and assume $\gamma', \mathtt{roll}(e) \Downarrow v', T'$. By inversion the derivation must be of the form:

$$\frac{\gamma', e \Downarrow v'_0, T'_0}{\gamma', \mathtt{roll}(e) \Downarrow \mathtt{roll}(v'_0), \mathtt{roll}(T'_0)}$$

so by induction we have $\rho, T'_0 \xrightarrow{\mathsf{obf}} p, S$ and $p \sqsubseteq v'_0$. We can conclude that:

$$\frac{\rho, T'_0 \xrightarrow{\mathsf{obf}} p, S}{\rho, \mathtt{roll}(T'_0) \xrightarrow{\mathsf{obf}} \mathtt{roll}(p), \mathtt{roll}(S)} \quad \mathtt{roll}(p) \sqsubseteq \mathtt{roll}(v'_0)$$

- If the derivations are of the form:

$$\frac{\rho, T \xrightarrow{\mathsf{obf}} \mathtt{roll}(p), S}{\rho, \mathtt{unroll}(T) \xrightarrow{\mathsf{obf}} p, \mathtt{unroll}(S)} \quad \frac{\gamma, e \Downarrow \mathtt{roll}(v), T}{\gamma, \mathtt{unroll}(e) \Downarrow v, \mathtt{unroll}(T)}$$

then let $\gamma' \sqsupseteq \rho$ be given and assume $\gamma', \texttt{unroll}(e) \Downarrow v', T'$. By inversion this derivation must be of the form:

$$\frac{\gamma', e \Downarrow \texttt{roll}(v'), T'_0}{\gamma', \texttt{unroll}(e) \Downarrow v', \texttt{unroll}(T'_0)}$$

so $T' = \texttt{unroll}(T'_0)$. By induction we have that $\rho, T'_0 \xrightarrow{\textsf{obf}} \texttt{roll}(p), S$ where $\texttt{roll}(p) \sqsubseteq \texttt{roll}(v')$, so we can conclude:

$$\frac{\rho, T'_0 \xrightarrow{\textsf{obf}} \texttt{roll}(p), S}{\rho, \texttt{unroll}(T'_0) \xrightarrow{\textsf{obf}} p, \texttt{unroll}(S)} \quad p \sqsubseteq v'$$

- If the derivations are of the form:

$$\frac{\rho, T \xrightarrow{\textsf{obf}} \Box, S}{\rho, \texttt{unroll}(T) \xrightarrow{\textsf{obf}} \Box, \Box} \quad \frac{\gamma, e \Downarrow \texttt{roll}(v), T}{\gamma, \texttt{unroll}(e) \Downarrow v, \texttt{unroll}(T)}$$

then let $\gamma' \sqsupseteq \rho$ be given and assume $\gamma', \texttt{unroll}(e) \Downarrow v', T'$. By inversion this derivation must be of the form:

$$\frac{\gamma', e \Downarrow \texttt{roll}(v'), T'_0}{\gamma', \texttt{unroll}(e) \Downarrow v', \texttt{unroll}(T'_0)}$$

so $T' = \texttt{unroll}(T'_0)$. By induction we have that $\rho, T'_0 \xrightarrow{\textsf{obf}} \Box, S$ where $\Box \sqsubseteq \texttt{roll}(v')$, so we can conclude:

$$\frac{\rho, T'_0 \xrightarrow{\textsf{obf}} \Box, S}{\rho, \texttt{unroll}(T'_0) \xrightarrow{\textsf{obf}} \Box, \Box} \quad \Box \sqsubseteq v'$$

$\Box$