A Core Calculus for Provenance Inspection*†

Wilmer Ricciotti
Laboratory for Foundations of Computer Science
University of Edinburgh
Edinburgh, Scotland
research@wilmer-ricciotti.net

ABSTRACT

Recent research has been devoting increasing attention to provenance, or information describing the origin, derivation, and history of data, due to its relevance to critical issues including transparency, privacy, and security. Engineering a software system to make it provenance-aware by means of ad-hoc instrumentation requires a substantial effort: the development of general-purpose infrastructure is thus very important to achieve the goal of making provenance widely available. In this article we describe a core functional language equipped with a provenance-aware semantics that is sufficiently generic to accomodate many notions of provenance proposed in the literature. While existing proposals typically treat provenance views and provenance extraction as second-class, extralinguistic mechanisms, in our work provenance views are expressed as standard programs and provenance data can be reflected into the language, allowing for programs that inspect their own provenance.

CCS CONCEPTS

• **Theory of computation** → *Rewrite systems*; • **Software and its engineering** → **Functional languages**; **Semantics**;

KEYWORDS

provenance, semantics, auditing

ACM Reference Format:
Wilmer Ricciotti. 2017. A Core Calculus for Provenance Inspection. In *Proceedings of PPDP'17, Namur, Belgium, October 9–11, 2017,* 12 pages.
https://doi.org/10.1145/3131851.3131871

1 INTRODUCTION

Not all information can be trusted: we know from experience that some sources are unreliable; but even trustworthy sources can make mistakes, which is why, to decide whether some information can be believed, we need to consider how it was processed. This shows the importance of an accurate description of the origin, history, and derivation of some information, which has been known for decades as *provenance* [3, 22], but has clearly been studied by epistemologists for a much longer time. Provenance is essential to the development of scientific knowledge: raw experimental data is useless without a thorough explanation of how it was obtained, allowing the public to reproduce the experiment (at least in principle) and confirm it or contradict it. Data collection and processing was originally a human activity (and scientists had to keep an accurate record of it); by means of computer systems, it is now possible to process much greater quantities of data in an efficient way.

Computer systems do not, however, provide the provenance of the data they process for free: propagating it from the input to the output, while logging all intermediate operations, requires additional code; and more code means not only more effort, but also more occasions to introduce bugs. For these reasons, a line of research has devoted its attention to automated techniques to record provenance. Many of these techniques are designed to be employed in interesting but restricted settings (especially scientific workflow computation [5, 12, 19, 23] and databases [4, 7, 11, 14, 15]).

In a 2013 paper [1], Acar et al. have laid the basis for tracking provenance in a general-purpose programming language, by proposing a core functional calculus (*Transparent* ML, or TML) equipped with a provenance extraction framework. They provide a *tracing* big-step semantics, which records the history (or *trace*) of computation; by analysing this trace, the extraction framework is able to "replay" the computation, while at the same time propagating provenance annotations from the source expression to the final value. They show that TML and its provenance framework are expressive enough to extract many forms of provenance.

Despite this, in TML provenance remains a second class citizen: the extraction framework is an external layer combining several user-provided functions (*provenance views*) to process annotations and an execution trace and return the provenance; neither the views, nor the annotations, nor the execution trace are expressed as TML terms. Secondly, traces appear as an unneeded intermediate step, as the extraction framework is required to replay them, essentially executing the program a second time. Furthermore, TML traces closely reflect the big-step semantics of the language, limiting implementations to a call-by-value strategy and essentially requiring provenance inspection to take place after the execution of the program, and making *self-inspecting programs* impossible.

1.1 Summary

We take inspiration from TML and develop a more tightly integrated *provenance inspection calculus* (PIC), in which provenance is

*An extended version of this paper can be found at http://www.wilmer-ricciotti.net/
†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 authorised to reproduce and distribute reprints for their purposes notwithstanding any copyright notation thereon.

PPDP'17, October 9–11, 2017, Namur, Belgium

ACM ISBN 978-1-4503-5291-8/17/10…$15.00
https://doi.org/10.1145/3131851.3131871

expressed as, and manipulated by first-class terms of the language. In this paper, we provide the following contributions:

- a core functional calculus with special constructs to manipulate syntactic locations (represented as lists of integers)
- language types for provenance labellings and provenance *transducers* together with a rich set of combinators
- a small-step, traceless semantics associating to each computation step the corresponding provenance transducer
- a simplified framework to define provenance views as language expressions
- an *inspection* construct allowing reflective reasoning on provenance

Outline. Section 2 introduces some basic notions about provenance. The syntax and semantics of PIC is introduced in Section 3, along with the provenance combinators. In Section 4 we provide a framework for the simplified definition of provenance views. Section 5 integrates PIC with an inspection construct and discusses the full language with an extensive example. Finally in Section 6 we draw conclusions and discuss future work.

2 BACKGROUND

Provenance is most easily defined informally, as a form of metadata describing the origin or history of an artifact, or the process by which it has been constructed. As soon as we try to make this concept more formal, we realize that it depends on choices which are, to a certain degree, arbitrary: the origin of data can include an identifier of the person or organization that provided it; a description of the technical equipment that produced it, along with its accuracy; a timestamp; localization information from a GPS sensor; etc. Likewise, the computational process manipulating data can be described with various levels of precision. We thus do not expect a single, universally acceptable notion of provenance. It is however clear that to mechanize provenance, we need facilities to annotate input and output data with metadata, along with a way to analyze the execution history of a software system.

The core calculus TML [1] provides a big-step tracing semantics, with evaluation judgments in the following form:[1]

$$M \Downarrow V, t$$

The judgment evaluates M and produces a result V and a trace t, which is a data structure describing the computation that produced V (with a slight approximation, t can be considered a linearized version of the evaluation tree). In this setting, the evaluation of an arithmetic expression $N_1 + N_2$ takes the following form:

$$\frac{N_1 \Downarrow 2, t_1 \qquad N_2 \Downarrow 3, t_2 \qquad \hat{+}(2,3) = 5}{N_1 + N_2 \Downarrow 5, t_1 + t_2}$$

where $t_1 + t_2$ is the trace expressing the evaluation of a sum whose two arguments have histories described, respectively, by t_1 and t_2, and $\hat{+}$ is the meta-language sum. The evaluation of a larger term yields a more complex trace:

$$\dfrac{\dfrac{\dfrac{N_1 \Downarrow 2, t_1 \qquad N_2 \Downarrow 3, t_2 \qquad \hat{+}(2,3) = 5}{N_1 + N_2 \Downarrow 5, t_1 + t_2} \qquad N_3 \Downarrow 1, t_3}{\langle N_1 + N_2, N_3 \rangle \Downarrow \langle 5, 1 \rangle, \langle t_1 + t_2, t_3 \rangle}}{\mathtt{fst}(\langle N_1 + N_2, N_3 \rangle) \Downarrow 5, \mathtt{fst}(\langle t_1 + t_2, t_3 \rangle)}$$

[1] For simplicity, we omit evaluation environments.

TML provides a framework to extract various forms of provenance from traces: the user annotates the source term with primitive provenance information and passes it, along with the execution trace, to a *provenance view* which, basically, replays the computation while at the same time propagating the annotations. Views are TML's way of allowing a user to define several notions of provenance: if we annotate the original term as $\mathtt{fst}(\langle N_1^{a_1} + N_2^{a_2}, N_3^{a_3} \rangle^{a_4})$, and we want the provenance of the final value to reflect that of all the portions of the input on which it depends (*dependency provenance* [10]), it is possible to define, by structural recursion on the execution trace, a view that will return $5^{a_1 \cup a_2 \cup a_4}$. Notice that the framework is an external facility: annotations do not have to be TML terms, and views are defined as set-theoretic functions (although it is implied that they should be computable). Other forms of provenance that can be expressed in TML include:

- where-provenance [8], whose labels identify the input location from which each portion of an expression was copied;
- expression provenance [1], whose labels are expression graphs or trees describing how a value was computed in terms of primitive operations.

Motivated by the limitations of TML, we set out to embed the extraction framework into the language itself: as we started investigating the issue of allowing provenance extraction within programs, we realized that evaluating a sub-program to obtain its trace and then processing its trace with a provenance view, essentially evaluating it a second time, made little sense; it should be possible to compute the provenance applying a view immediately, without materializing the execution trace as an intermediate step. To accomplish this task, we decided to replace traces with *transducers*, or functions that transform provenance according to a certain execution history. A major obstacle in defining transducers is the need to undo the substitutions created by the evaluation of functions or pattern matching: to address it, we equip the language with a special *relocation* operator. One property of transducers is that they can be combined by functional composition to express a transitivity rule. This allows us to adopt a small-step semantics, which can be used to reason on the provenance of non-terminating programs.[2]

3 A PROVENANCE-AWARE CALCULUS

We define the *provenance inspection calculus* PIC, a pure functional language equipped with a reduction semantics and extended with facilities to annotate expressions with provenance and propagate it during the computation. Its syntax, shown in Fig. 1, includes a standard type system, with primitive types B (e.g. the singleton type $\mathbf{1}$, booleans $\mathbf{B}$, and natural numbers $\mathbf{N}$), pairs, functions, and lists and optional values on a base type σ (respectively $[\sigma]$ and $\sim\sigma$).

The definition of expressions is standard. We parametrize the syntax and semantics over primitive constants c and primitive operations $\oplus$, whose arguments must be of primitive types. We assume constants and primitive operations including booleans `tt` and `ff`, and natural numbers $0, 1, 2 \ldots$, along with standard arithmetic operations and the equality test on natural numbers $\stackrel{?}{=}$.

[2] A small-step semantics of TML was originally considered for this precise reason, but left for future work. In the words of its proponents, *"moving to a small-step semantics [seems] likely to complicate the trace semantics (and subsequent analysis) considerably"*.

Prim. types	B	$::= \mathbf{1} \mid \mathbf{B} \mid \mathbf{N} \mid \ldots$
Types	σ, τ	$::= B \mid \sigma \times \tau \mid \sigma \supset \tau \mid [\sigma] \mid {\sim}\sigma$
Contexts	Γ	$::= [x_1 : \tau_1, \ldots, x_n : \tau_n]$
Expressions	L, M, N	$::= c \mid x \mid M\ N \mid f(x).M \mid \oplus(\overrightarrow{M})$
		$\mid \bullet \mid {\sim}M \mid \epsilon \mid M \# N \mid \mathtt{case}\ M\ \mathtt{of}\ \pi$
		$\mid \mathtt{fst}(M) \mid \mathtt{snd}(M) \mid \langle M, N\rangle \mid \rho(\overrightarrow{x}.M)$
Patterns	π	$::= \{\mathtt{tt} \mapsto M; \mathtt{ff} \mapsto N\}$
		$\mid \{\bullet \mapsto M; {\sim}x \mapsto N\} \mid \{\epsilon \mapsto M; x \# y \mapsto N\}$
Views	F	$::= \langle M_\beta, M_{\mathtt{fst}}, M_{\mathtt{snd}}, M_\bullet, M_\sim,$
		$M_\epsilon, M_\#, M_{\mathtt{tt}}, M_{\mathtt{ff}}, M_\oplus\rangle$
Values	U, V	$::= c \mid x \mid \bullet \mid {\sim}U \mid \epsilon \mid U \# V \mid \langle U, V\rangle$
		$\mid f(x).M \mid \rho(\overrightarrow{x}.M)$

Figure 1: Syntax of the language

We fix a multi-sorted signature Σ, containing type declarations for constants (e.g.: $c : \sigma$) and primitive operations ($\oplus : \overrightarrow{\tau_m} \supset \tau'$, where $\overrightarrow{\tau_m}$ is a sequence of m types $\tau_1, \ldots, \tau_m$). Each operation symbol $\oplus$ is associated to a metalanguage function $\hat{\oplus}$ from values to values, expressing its semantics, thus defining an algebra for Σ.

Expressions $f(x).M$ represent recursive functions f with formal argument x and body M, where both f and x are bound in M; non-recursive lambda-expressions $\lambda x.M$ are treated as recursive functions $\_(x).M$, under the condition that the irrelevant function name $\_$ is free in M.

For optional values and lists, we use the following notation: an expression of type ${\sim}\sigma$ can be obtained either by encapsulating some M of type σ using the syntax ${\sim}M$, or by means of the "none" constructor represented by $\bullet$; list constructors include the empty list ϵ and the *cons* operation $M\#N$. For explicit lists, we will use the notation $[M_1; \ldots; M_k]$, which is syntactic sugar for $M_1 \# \ldots \# M_k \# \epsilon$.

Two case analysis constructs are used to decompose lists and optional values; similarly, case analysis also supports reasoning on booleans. We will allow ourselves to use an extended case analysis notation, which should be considered syntactic sugar:

$$\begin{array}{l}\mathtt{case}\ M\ \mathtt{of}\\ \quad \{\epsilon \mapsto N_1\\ \quad ; 1 \# l_1 \mapsto N_2\\ \quad ; 2 \# 1 \# l_2 \mapsto N_3\\ \quad ; \_ \mapsto N_4\}\end{array}$$

$$\begin{array}{l}\mathtt{case}\ M\ \mathtt{of}\\ \quad \{\epsilon \mapsto N_1\\ \quad ; h_1 \# l_1 \mapsto \mathtt{case}\ h_1 \stackrel{?}{=} 1\ \mathtt{of}\\ \qquad \{\mathtt{tt} \mapsto N_2\\ \qquad ; \mathtt{ff} \mapsto \mathtt{case}\ h_1 \stackrel{?}{=} 2\ \mathtt{of}\\ \qquad\quad \{\mathtt{tt} \mapsto \mathtt{case}\ l_1\ \mathtt{of}\\ \qquad\qquad \{\epsilon \mapsto N_4\\ \qquad\qquad ; h_2 \# l_2 \mapsto \mathtt{case}\ h_2 \stackrel{?}{=} 2\ \mathtt{of}\\ \qquad\qquad\qquad \{\mathtt{tt} \mapsto N_3; \mathtt{ff} \mapsto N_4\}\\ \qquad\qquad \}\\ \qquad\quad ; \mathtt{ff} \mapsto N_4\}\\ \qquad \}\\ \quad \}\end{array}$$

As an example, we can use recursion and case analysis to define a list map function of type $(\mathbf{N} \supset \mathbf{N}) \supset [\mathbf{N}] \supset [\mathbf{N}]$ as follows:

$$\begin{array}{l}\mathtt{map} := \lambda f.mapf(l).\mathtt{case}\ l\ \mathtt{of}\\ \qquad \{\epsilon \mapsto \epsilon; h\#t \mapsto (f\ h)\#(mapf\ t)\}\end{array}$$

The main feature of PIC is that its semantics is expressed by reduction judgments in the following form:

$$F \vdash M \overset{T}{\hookrightarrow} N$$

The intended reading is that M reduces to N; additionally, F is a parameter describing a *provenance view*, or a set of rules that express how the initial provenance data contributes to the final provenance, and T is a *provenance transducer*, a normal language expression that, given provenance data for M, returns the provenance for N.

The reduct N does not depend on the view F, therefore we will write $M \hookrightarrow N$ when we evaluate an expression, but have no interest in its provenance; on the other hand, T is constructed based on the content of F: thus different views will produce different transducers.

One last expression kind, which we introduce in this calculus, is the *relocation* $\rho(\overrightarrow{x}.M)$, of type $[\mathbf{N}] \supset [\mathbf{N}]$. This operation, which expresses a mapping from locations in a (simultaneously) substituted term $M[\overrightarrow{N}/\overrightarrow{x}]$ to locations in its components $M, \overrightarrow{N}$, cannot be defined in terms of other language constructs and is thus taken as primitive. Notice that, since this mapping depends on the syntactic structure of M, no evaluation can take place within ρ.

3.1 Locations

To explain how relocations work, we need to formally describe what locations are and how they are manipulated in our language. A syntactic *location* is an entity that can be used to denote a specific subexpression within a larger language expression. We want to express provenance as a labelling, i.e. a function assigning a label to each location in an expression; it should also be possible to represent such labellings as language expressions, thus we need to provide a representation for locations as well.

We choose to represent syntactic locations by means of lists of natural numbers: given M, a list $[n_1, \ldots, n_k]$ identifies one of its subexpressions N by means of the abstract syntax tree path leading to N from the root of M; each number n_i in the list means that the next node in the path is the n_i-th child of the current node; the root location is represented by the empty list ϵ. For better clarity, we will write Loc rather than $[\mathbf{N}]$ when lists of natural numbers are used as locations: this is just a presentational choice, and the two notations express the same type.

$$\begin{array}{l}\mathsf{locs}(c) = \mathsf{locs}(x) = \mathsf{locs}(\bullet) = \mathsf{locs}(\epsilon) = \mathsf{locs}(\rho(\overrightarrow{x}.M)) = \{\epsilon\}\\ \mathsf{locs}(f(x).M) = \mathsf{locs}(\mathtt{fst}(M)) = \mathsf{locs}(\mathtt{snd}(M)) = \mathsf{locs}({\sim}M)\\ \quad = \{\epsilon\} \cup (1 \# \mathsf{locs}(M))\\ \mathsf{locs}(M\ N) = \mathsf{locs}(\langle M, N\rangle) = \mathsf{locs}(M \# N)\\ \quad = \{\epsilon\} \cup (1 \# \mathsf{locs}(M)) \cup (2 \# \mathsf{locs}(N))\\ \mathsf{locs}(\mathtt{case}\ M_1\ \mathtt{of}\ \{\_ \mapsto M_2; \_ \mapsto M_3\}))\\ \quad = \{\epsilon\} \cup (\bigcup_{i=1\ldots3} i \# \mathsf{locs}(M_i))\\ \mathsf{locs}(\oplus(\overrightarrow{M_n})) = \{\epsilon\} \cup (\bigcup_{i=1\ldots n} i \# \mathsf{locs}(M_i))\end{array}$$

where we have abused the cons notation to apply it to sets of terms $\mathcal{S}$ as follows:

$$n \# \mathcal{S} \triangleq \{n \# M : M \in \mathcal{S}\}$$

A lookup meta-operation $M\big|_\ell$ computes the subterm of M to which the location ℓ points; if, while scanning M, the operation reaches a variable before consuming the whole location, it returns

the variable together with the remaining, unconsumed part of the location. The location lookup operation is defined in Fig. 2.

In all other cases, the location is inconsistent with the term being scanned and lookup is undefined.

Given a location ℓ_N valid for N, we can obtain the corresponding location ℓ valid for $\langle M, N\rangle$ as $\ell = 2 \# \ell_N$. More generally, given a composite n-ary expression $k(M_1, \ldots, M_n) = k(\overrightarrow{M_n})$, we can map locations for M_i into locations for $k(\overrightarrow{M_n})$ by means of a *location injection* $\Im^{\mathsf{Loc}}_{n.i} = \lambda\ell.\, i\#\ell$. Dually, a location for $k(\overrightarrow{M_n})$ can be processed by case analysis as follows: we map the root location ϵ to a given N of type τ; moreover, for each $i = 1, \ldots, n$, we provide a function p_i from Loc to a fixed type τ which is used to map locations $i \# \ell$ for the i-th subexpressions to $p_i\ \ell$. This case analysis, which we call *location elimination*, can be expressed as a term of the language, which we will refer to as $\mathfrak{E}^{\mathsf{Loc}}_n$.

3.2 Labels and provenance labellings

A provenance labelling for an expression M is a function that maps all locations valid for M to expressions of an arbitrary option type $\sim\tau$, where $\bullet$ is taken to represent a default, non-informative label. We will denote the type of τ-valued labellings by $\mathsf{Prov}_\tau = \mathsf{Loc} \supset \sim\tau$; we will also allow ourselves to write Prov when the intended τ is obvious. We will use the metavariables a and $\widetilde{a}$, possibly decorated with subscripts or superscripts, to denote base labels and optional labels respectively. In general, we will define provenance labellings only for the locations in $\mathsf{locs}(M)$, even though our type system does not guarantee that labellings will be applied consistently: we assume that if a labelling is not explicitly defined for a location, it returns $\bullet$. The default labelling $\lambda\_.\bullet$ will be denoted by $\bot$.

The labelling for a composite n-ary expression $k(\overrightarrow{M_n})$ can be obtained by putting together labellings p_i for every sub-expression M_i, and providing an additional label a for the root. The corresponding n-ary *labelling injection* operator $\Im^{\mathsf{Prov}}_n$ has the same definition of the location elimination $\mathfrak{E}^{\mathsf{Loc}}_n$. Dually, a labelling for a composite expression $k(\overrightarrow{M_n})$ induces a projection labelling for each of its subexpressions M_i. The *labelling projections* $\mathfrak{E}^{\mathsf{Prov}}_{n.i}$ defined as $\lambda p.p \circ \Im^{\mathsf{Loc}}_{n.i}$ compose a labelling for $k(\overrightarrow{M_n})$ with the i-th injection.

3.3 Provenance transducers

The way a labelling is transformed by evaluation allows us to express different kinds of provenance, including where-provenance, dependency provenance, and expression provenance: whenever an expression M reduces to N, there should be a way to transform any labelling for M into a labelling for N. This transformation will be expressed intra-linguistically, by means of functions of type $\mathsf{Prov}_\tau \supset \mathsf{Prov}_\tau$ which we call *provenance transducers*; this type will also be written Trans_τ for short. Rather than defining an ad-hoc transducer for each possible reduction, which would be of little use, we will give rules to synthesize transducers based on the reduction rules of the language: for example, we will have a transducer for the reduction of function application, two transducers for the two reductions of case analysis over option types, and similarly two transducers for case analysis over lists, etc.

$$
\begin{array}{rrr}
x\big|_\ell = x, \ell & M\big|_\epsilon = M, \epsilon & \oplus(\overrightarrow{M_n})\big|_{i\#\ell} = M_i\big|_\ell \\
(f(x).M)\big|_{1\#\ell} = M\big|_\ell & (M_1\ M_2)\big|_{i\#\ell} = M_i\big|_\ell & \langle M_1, M_2\rangle\big|_{i\#\ell} = M_i\big|_\ell \\
\mathtt{fst}(M)\big|_{1\#\ell} = M\big|_\ell & \mathsf{snd}(M)\big|_{1\#\ell} = M\big|_\ell & (\sim M)\big|_{1\#\ell} = M\big|_\ell \\
(M_1 \# M_2)\big|_{i\#\ell} = M_i\big|_\ell & &
\end{array}
$$

$$
\mathtt{case}\ M_1\ \mathtt{of}\ \{\_ \mapsto M_2; \_ \mapsto M_3\}\big|_{i\#\ell} = M_i\big|_\ell
$$

Figure 2: Location lookup

In the special case of congruence rules, the corresponding transducer will not be provided by the user: the language provides default transducers that reflect the administrative nature of these rules. For example, given a transducer T for the reduction $M \hookrightarrow M'$, the transducer for the congruence reduction $M\ N \hookrightarrow M'\ N$ can and should be obtained by lifting T.

In general, given a reduction $M_i \hookrightarrow M_i'$ and a corresponding transducer T, the transducer for the congruence $k(M_1, \ldots, M_i, \ldots, M_n) \hookrightarrow k(M_1, \ldots, M_i', \ldots, M_n)$ must perform the following actions:

(1) obtain a labelling p for the full unreduced expression;
(2) apply labelling projections to obtain the labellings $p_1, \ldots, p_n$ for each of the sub-expressions $M_1, \ldots, M_n$;
(3) apply T to p_i to obtain a labelling for M_i';
(4) use the labelling injection $\Im^{\mathsf{Prov}}_n$ to compose the new labelling for M_i' with the old labellings for the unchanged M_j $(j \neq i)$.

The above actions are expressed by the operator $\Im_{n.i}$ (i-th *transducer congruence* of order n), defined in terms of $\Im^{\mathsf{Prov}}_n$ and $\mathfrak{E}^{\mathsf{Prov}}_{n.j}$.

Fig. 3 summarizes the definition of location injections and elimination, labelling injection and projections, and transducer congruences. We will refer to these operations, collectively, by the name of *provenance combinators*.

3.4 Semantics

We now use the notions introduced in the previous sections to formally define a provenance-aware small-step semantics: its rules are shown in Fig. 4. The reduction judgment $F \vdash M \overset{T}{\hookrightarrow} N$ of provenance-aware semantics involves a provenance transducer T, which has type Trans_τ for some τ, and relates the provenance of M to that of N. As anticipated, some of the reduction rules make use of the relocation operator $\rho(\ldots)$: to understand why, it is sufficient to see that a transducer ultimately needs to translate every location in the reduct into a location in the redex. Sometimes this translation can be expressed as an elementary program: for instance, when reducing a pair projection

$$
\mathtt{fst}\ \langle R, S\rangle \hookrightarrow R
$$

a location ℓ for R corresponds to $1 \# 1 \# \ell$ in $\mathtt{fst}\ \langle R, S\rangle$. However, for reduction rules involving substitutions, the correspondence is not so simple. In the case of a function application reduction

$$
(f(x).M)\ N \hookrightarrow M[f(x).M, N/f, x]
$$

the only way to map locations in the reduct to locations in the redex is by means of a structural analysis of the expressions involved in the reduction, which is not allowed by standard language constructs. We thus extend the language with relocation expressions $\rho(\overrightarrow{x}.M)$

Location injection: $\mathsf{Loc} \supset \mathsf{Loc}$
$\Im^{\mathsf{Loc}}_{n.i} := \lambda \ell.\ i \# \ell$ (n is irrelevant)

Location elimination: $(\mathsf{Loc} \supset \tau) \supset \cdots \supset (\mathsf{Loc} \supset \tau) \supset \tau \supset \mathsf{Loc} \supset \tau$
$\mathfrak{E}^{\mathsf{Loc}}_{n} := \lambda p_1, \ldots, p_n, a, \ell.\ \mathsf{case}\ \ell\ \mathsf{of}\ \{\epsilon \mapsto a; 1 \# \ell' \mapsto p_1\ \ell'; \ldots; n \# \ell' \mapsto p_n\ \ell'\}$

Provenance injection: $\mathsf{Prov}_\tau \supset \cdots \supset \mathsf{Prov}_\tau \supset {\sim}\tau \supset \mathsf{Prov}_\tau$
$\Im^{\mathsf{Prov}}_{n} := \mathfrak{E}^{\mathsf{Loc}}_{n}$

Provenance projection: $\mathsf{Prov}_\tau \supset \mathsf{Prov}_\tau$
$\mathfrak{E}^{\mathsf{Prov}}_{n.i} := \lambda p.\ p \circ \Im^{\mathsf{Loc}}_{n.i}$

Transducer congruence: $\mathsf{Trans}_\tau \supset \mathsf{Trans}_\tau$
$\Im_{n.i} := \lambda t, p.\ \Im^{\mathsf{Prov}}_{n}\ (\mathfrak{E}^{\mathsf{Prov}}_{n.1}\ p) \cdots (\mathfrak{E}^{\mathsf{Prov}}_{n.(i-1)}\ p)\ (t\ (\mathfrak{E}^{\mathsf{Prov}}_{n.i}\ p))\ (\mathfrak{E}^{\mathsf{Prov}}_{n.(i+1)}\ p) \cdots (\mathfrak{E}^{\mathsf{Prov}}_{n.n}\ p)\ (p\ \epsilon)$

Figure 3: Definition of the provenance combinators

which allow us to map redex locations to unreduced locations by essentially "undoing" the substitution. Relocation operations are functions with type $\mathsf{Loc} \supset \mathsf{Loc}$: when $\rho(\overrightarrow{x_n}.M)$ is applied to a location ℓ in $\mathsf{locs}(M[\overrightarrow{N_n}/\overrightarrow{x_n}])$, it first computes the lookup $M\big|_\ell$: if the lookup yields x_i, ℓ', for $i = 1, \ldots, n$ – i.e. one of the variables declared by the relocation, together with an unconsumed location – then ℓ' represents the path to the subexpression of N_i which corresponds to location ℓ in the substituted expression; in this case, the relocation maps ℓ to $i \# \ell'$. If instead the lookup $M\big|_\ell$ returns another subterm of M together with ϵ, the relocation maps ℓ to $(n+1)\#\ell$; in all the other cases, ℓ is an invalid location for $M[\overrightarrow{N_n}/\overrightarrow{x_n}]$, and we return a (meaningless) default location ϵ.

Thanks to relocations, we can map the locations of a beta-reduced expression back to the original function application. For instance, if $\ell \in \mathsf{locs}(M[f(x).M, N/f, x])$, $\rho(f, x.M)\ \ell$ will reduce to one among $1\#\ell_1$ (where $\ell_1 \in \mathsf{locs}(f(x).M)$), $2\#\ell_2$ (where $\ell_2 \in \mathsf{locs}(N)$, or $3\#\ell_3$ (where $\ell_3 \in \mathsf{locs}(M)$). Mapping these locations to $\mathsf{locs}((f(x).M)\ N)$ is then a matter of a simple case analysis, which we provide as a relocation helper β (with type $(\mathsf{Loc} \supset \mathsf{Loc}) \supset \mathsf{Loc} \supset \mathsf{Loc}$):

$$\beta := \lambda r, \ell.\mathsf{case}\ r\ \ell\ \mathsf{of}\ \{3 \# \ell_3 \mapsto 1 \# 1 \# \ell_3; \_ \mapsto \ell\}$$

Then for all N, the expression $\beta\ \rho(f, x.M)$ maps locations in $\mathsf{locs}(M[f(x).M, N/f, x])$ to locations in $\mathsf{locs}((f(x).M)\ N)$.

Non-recursive function applications $(\lambda x.M)\ N \hookrightarrow M[x \mapsto N]$ are a special case of recursive functions, thus the same operator can be used to express their relocation function as well.

The view F is, in essence, a collection of basic transducers that correspond to the basic reduction rules. This is especially clear in the rules for the contraction of pair projections and the simpler forms of case reductions, which simply return the transducers $F_{\mathtt{fst}}$, $F_{\mathtt{snd}}$, $F_\bullet$, F_ϵ, $F_{\mathtt{tt}}$, and $F_{\mathtt{ff}}$. The reduction of primitive operations is similar, save for the fact that we allow a transducer $F_\oplus$ to receive the actual arguments of $\oplus$ as parameters; furthermore, notice that primitive operations only reduce when applied to values.

The rule for function application must express the propagation of provenance under substitution; this requires us to produce a transducer capable of mapping locations in the substitution $M[f(x).M, N/f, x]$ back to their label in the unreduced application $(f(x).M)\ N$. To accomplish this task, F_β will receive the relocation function $\beta\ \rho(f, x.M)$ as a parameter. Case analysis can reduce to substituted expressions as well, which explains why $F_\sim$ and $F_\#$ receive relocations $\beta_\sim\ \rho(x.N)$ and $\beta_\#\ \rho(h, t.N)$ as parameters; the concrete definition of $\beta_\sim$ and $\beta_\#$ should not be too surprising:

$$\begin{aligned}\beta_\sim &:= \lambda r, \ell.\mathsf{case}\ r\ \ell\ \mathsf{of}\\ &\qquad \{1 \# \ell_1 \mapsto 1 \# 1 \# \ell_1; 2 \# \ell_2 \mapsto 3 \# \ell_2; \_ \mapsto \ell\}\\ \beta_\# &:= \lambda r, \ell.\mathsf{case}\ r\ \ell\ \mathsf{of}\\ &\qquad \{1 \# \ell_1 \mapsto 1 \# 1 \# \ell_1; 2 \# \ell_2 \mapsto 1 \# 2 \# \ell_2; \_ \mapsto \ell\}\end{aligned}$$

(notice that in $\beta_\sim$ the last branch of the case analysis can only be reached when ℓ is ϵ or an invalid location).

Finally, three rules define the computational behaviour of the relocator ρ. The first rule applies to locations ℓ that, within $M[\overrightarrow{N_n}/\overrightarrow{x_n}]$, reference a proper subterm of M that is not one of the substituted variables: this is converted to the location $(n + 1) \# \ell$. The second one explains what to do when the lookup of location ℓ within M references the i-th substituted variable, possibly with an unconsumed ℓ' (which is a right-hand sublist of the original ℓ): the reduced expression is ℓ', prepended by the natural number i. Finally, the third rule provides the evaluation of a relocation applied to an invalid location: this situation is meaningless, but we reduce to ϵ, with no special meaning other than it being a canonical location.

Since relocations are usually tools to extract provenance, and not expressions whose provenance should be extracted, for both reduction rules involving ρ we provide a forgetful transducer $\lambda\_.\bot$.

Congruence rules use the transducer congruences from Fig. 3. The generic form of a congruence is that of an n-ary composite expression $k(M_1, \ldots, M_i, \ldots, M_n)$, which reduces to $k(M_1, \ldots, M'_i, \ldots, M_n)$ whenever M_i reduces to M'_i. Given a transducer T from M_i to M'_i we can obtain the congruence transducer by means of the combinator $\Im_{n.i}$.

3.5 Type system

The type system of PIC (Fig. 5) is defined in a standard way. Aside from mentioning that it is parametric on the signature Σ used to type constants and primitive operations, the only rule that deserves

Reduction rules

$$\frac{}{F \vdash (f(x).M)\ N \xhookrightarrow{F_\beta\ (\beta\ \rho(f,x.M))} M[f(x).M, N/f, x]} \qquad \frac{}{F \vdash \oplus(\overrightarrow{V_n}) \xhookrightarrow{F_\oplus\ \overrightarrow{V_n}} \hat{\oplus}(\overrightarrow{V_n})}$$

$$\frac{}{F \vdash \mathtt{fst}(\langle M, N\rangle) \xhookrightarrow{F_{\mathtt{fst}}} M} \qquad \frac{}{F \vdash \mathtt{snd}(\langle M, N\rangle) \xhookrightarrow{F_{\mathtt{snd}}} N} \qquad \frac{(\mathtt{tt} \mapsto N) \in m}{F \vdash \mathtt{case}\ \mathtt{ff}\ \mathtt{of}\ m \xhookrightarrow{F_{\mathtt{tt}}} N} \qquad \frac{(\mathtt{ff} \mapsto N) \in m}{F \vdash \mathtt{case}\ \mathtt{ff}\ \mathtt{of}\ m \xhookrightarrow{F_{\mathtt{ff}}} N}$$

$$\frac{(\bullet \mapsto N) \in m}{F \vdash \mathtt{case}\ \bullet\ \mathtt{of}\ m \xhookrightarrow{F_\bullet} N} \qquad \frac{(\sim x \mapsto N) \in m}{F \vdash \mathtt{case}\ {\sim}M\ \mathtt{of}\ m \xhookrightarrow{F_\sim\ (\beta_\sim\ \rho(x.N))} N[M/x]}$$

$$\frac{(\epsilon \mapsto N) \in m}{F \vdash \mathtt{case}\ \epsilon\ \mathtt{of}\ m \xhookrightarrow{F_\epsilon} N} \qquad \frac{(h \,\#\, t \mapsto N) \in m}{F \vdash \mathtt{case}\ M_1 \,\#\, M_2\ \mathtt{of}\ m \xhookrightarrow{F_\#\ (\beta_\#\ \rho(h,t.N))} N[M_1, M_2/h, t]}$$

$$\frac{M\big|_\ell = M', \epsilon \qquad M' \notin \overrightarrow{x_n}}{F \vdash \rho(\overrightarrow{x_n}.M)\ \ell \xhookrightarrow{\lambda\_.\bot} (n+1) \,\#\, \ell} \qquad \frac{M\big|_\ell = x_i, \ell' \qquad 1 \le i \le n}{F \vdash \rho(\overrightarrow{x_n}.M)\ \ell \xhookrightarrow{\lambda\_.\bot} i \,\#\, \ell'}$$

$$\frac{M\big|_\ell = N, \ell' \qquad \ell' \neq \epsilon \qquad \text{for all } i \text{ s.t. } 1 \le i \le n : N \neq x_i}{F \vdash \rho(\overrightarrow{x_n}.M)\ \ell \xhookrightarrow{\lambda\_.\bot} \epsilon}$$

Congruence rules

$$\frac{F \vdash M_i \overset{T}{\hookrightarrow} M_i'}{F \vdash \oplus(\overrightarrow{M_n}) \xhookrightarrow{\Im_{n.i}\ T} \oplus(M_1, \ldots, M_i', \ldots, M_n)} \qquad \frac{F \vdash M \overset{T}{\hookrightarrow} M'}{F \vdash M\ N \xhookrightarrow{\Im_{2.1}\ T} M'\ N} \qquad \frac{F \vdash N \overset{T}{\hookrightarrow} N'}{F \vdash M\ N \xhookrightarrow{\Im_{2.2}\ T} M\ N'}$$

$$\frac{F \vdash M \overset{T}{\hookrightarrow} M'}{F \vdash \langle M, N\rangle \xhookrightarrow{\Im_{2.1}\ T} \langle M', N\rangle} \qquad \frac{F \vdash N \overset{T}{\hookrightarrow} N'}{F \vdash \langle M, N\rangle \xhookrightarrow{\Im_{2.2}\ T} \langle M, N'\rangle}$$

$$\frac{F \vdash M \overset{T}{\hookrightarrow} M'}{F \vdash \mathtt{fst}(M) \xhookrightarrow{\Im_{1.1}\ T} \mathtt{fst}(M')} \qquad \frac{F \vdash M \overset{T}{\hookrightarrow} M'}{F \vdash \mathtt{snd}(M) \xhookrightarrow{\Im_{1.1}\ T} \mathtt{snd}(M')}$$

$$\frac{F \vdash M \overset{T}{\hookrightarrow} M'}{F \vdash \mathtt{case}\ M\ \mathtt{of}\ m \xhookrightarrow{\Im_{3.1}\ T} \mathtt{case}\ M'\ \mathtt{of}\ m}$$

Figure 4: Provenance-carrying semantics

to be mentioned here is the one concerning relocations. Relocations, which are functions from Loc to Loc, are well typed provided that their argument is well-typed in a properly extended context.

Even though provenance views as such are not PIC-expressions, they are used in reductions and must be well-typed to function properly. To typecheck a view, we use an auxiliary definition $\mathcal{T}_\sigma$ (Fig. 6): each of the transducers F_κ within a view F must have type $\mathcal{T}_\sigma(\kappa)$, where σ is a type of provenance labels and κ identifies one of the basic reduction rules. Each $\mathcal{T}_\sigma(\kappa)$ returns a provenance transducer of type Trans_σ, which may be parametrized by a relocation (for $\kappa = \beta, \sim, \#$) or by the actual arguments of the primitive operation being reduced (for $\kappa = \oplus$). Type-safety for PIC follows immediately.

Theorem 3.1 (Preservation and progress). (1) *If* $\Gamma \vdash M : \sigma$, $\Gamma \vdash F : \mathcal{T}_\tau$, *and* $F \vdash M \overset{T}{\hookrightarrow} M'$, *then* $\Gamma \vdash M' : \sigma$ *and* $\Gamma \vdash T : \mathsf{Trans}_\tau$.

(2) *If* $\vdash M : \sigma$ *and* M *is not a value, then for all* F, τ *s.t.* $\vdash F : \mathcal{T}_\tau$, *there exist* T, M' *such that* $F \vdash M \overset{T}{\hookrightarrow} M'$.

4 PROVENANCE VIEWS

The provenance-carrying semantics we defined allows us to choose a view F with great flexibility. Consider for example the reduction:

$$F \vdash \mathtt{fst}(\langle M, N\rangle) \xhookrightarrow{F_{\mathtt{fst}}} M$$

Expression typing

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash \overrightarrow{M} : \overrightarrow{\sigma} \qquad \oplus : \overrightarrow{\sigma} \supset \tau \in \Sigma}{\Gamma \vdash \oplus(\overrightarrow{M}) : \tau} \qquad \frac{\Gamma, \overrightarrow{x} : \overrightarrow{\sigma} \vdash M : \tau}{\Gamma \vdash \rho(\overrightarrow{x}.M) : \mathsf{Loc} \supset \mathsf{Loc}}$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash \langle M, N \rangle : \sigma \times \tau} \qquad \frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \mathtt{fst}(M) : \sigma} \qquad \frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \mathsf{snd}(M) : \tau}$$

$$\frac{\Gamma, f : \sigma \supset \tau, x : \sigma \vdash M : \tau}{\Gamma \vdash f(x).M : \sigma \supset \tau} \qquad \frac{\Gamma \vdash M : \sigma \supset \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau} \qquad \frac{\Gamma \vdash M : \mathbf{B} \qquad \Gamma \vdash N : \tau \qquad \Gamma \vdash R : \tau}{\Gamma \vdash \mathsf{case}\ M\ \mathsf{of}\ \{\mathtt{tt} \mapsto N; \mathtt{ff} \mapsto R\} : \tau}$$

$$\frac{}{\Gamma \vdash \bullet : {\sim}\sigma} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash {\sim}M : {\sim}\sigma} \qquad \frac{\Gamma \vdash M : {\sim}\sigma \qquad \Gamma \vdash N : \tau \qquad \Gamma, x : \sigma \vdash R : \tau}{\Gamma \vdash \mathsf{case}\ M\ \mathsf{of}\ \{\bullet \mapsto N; {\sim}x \mapsto R\} : \tau}$$

$$\frac{}{\Gamma \vdash \epsilon : [\sigma]} \qquad \frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : [\sigma]}{\Gamma \vdash M \# N : [\sigma]} \qquad \frac{\Gamma \vdash M : [\sigma] \qquad \Gamma \vdash N : \tau \qquad \Gamma, h : \sigma, t : [\sigma] \vdash R : \tau}{\Gamma \vdash \mathsf{case}\ M\ \mathsf{of}\ \{\epsilon \mapsto N; h \# t \mapsto R\} : \tau}$$

Provenance view typing

$$\frac{\text{for all } \kappa\text{: } \Gamma \vdash F_\kappa : \mathcal{T}_\sigma(\kappa)}{\Gamma \vdash F : \mathcal{T}_\sigma}$$

Figure 5: Typing rules

$$\begin{aligned}
\mathcal{T}_\sigma(\beta) &= (\mathsf{Loc} \supset \mathsf{Loc}) \supset \mathsf{Trans}_\sigma \\
\mathcal{T}_\sigma(\sim) &= (\mathsf{Loc} \supset \mathsf{Loc}) \supset \mathsf{Trans}_\sigma \\
\mathcal{T}_\sigma(\#) &= (\mathsf{Loc} \supset \mathsf{Loc}) \supset \mathsf{Trans}_\sigma \\
\mathcal{T}_\sigma(\oplus) &= \overrightarrow{\tau} \supset \mathsf{Trans}_\sigma \qquad (\text{if } \oplus : \overrightarrow{\tau} \supset \tau' \in \Sigma) \\
\mathcal{T}_\sigma(_) &= \mathsf{Trans}_\sigma
\end{aligned}$$

Figure 6: The type of views

Here, the transducer $F_{\mathtt{fst}}$ is allowed to transform the input labelling without any restriction: the most obvious choice would be to propagate the labels of the first element of the source pair, but other choices are possible. F could erase all labels, or add a static label to arbitrary subterms of M; it could even provide a new labelling of M using labels from N, even though N is not part of the reduced term: the only limitation is that $F_{\mathtt{fst}}$ be expressible as a language function. The generality with which provenance views can be defined is one of the strengths of our approach, but it comes at a cost: views will usually be defined by pattern matching on locations, and the transducers for function applications and case analysis will often have to use a relocation function explicitly, which is rather cumbersome. For many notions of provenance such a generality is not needed and is only an element of confusion. Consider, for example, a `fst`-projection labelled with three distinguished provenance annotations in the form $\mathtt{fst}(\langle M^{\widetilde{a}_1}, N\rangle^{\widetilde{a}_2})^{\widetilde{a}_3}$ (for better readability, we represent provenance annotations as superscripts, rather than providing an explicit function expression from locations to annotations). When reducing the projection, we might want the outermost label of the reduced M to depend only on those $\widetilde{a}_i$ (ignoring the annotations of N) and the inner subexpressions of M to keep the same labels that they had before the reduction. In other words, we would like the labelling to be transformed as follows:

$$\mathtt{fst}(\langle M^{\widetilde{a}_1}, N\rangle^{\widetilde{a}_2})^{\widetilde{a}_3} \rightsquigarrow M^{v_{\mathtt{fst}}\ \widetilde{a}_3\ \widetilde{a}_1\ \widetilde{a}_2}$$

where $v_{\mathtt{fst}}$ is some expression with type ${\sim}\tau \supset {\sim}\tau \supset {\sim}\tau \supset {\sim}\tau$, for an annotation type ${\sim}\tau$.

Similarly, when reducing $((f(x).M^{\widetilde{a}_1})^{\widetilde{a}_2}\ N^{\widetilde{a}_3})^{\widetilde{a}_4}$, we often expect the reduced term to be, essentially, $M^{\widetilde{a}_1}$ where free occurrences of f and x have been replaced by $(f(x).M^{\widetilde{a}_1})^{\widetilde{a}_2}$ and $N^{\widetilde{a}_3}$; the outermost label of the reduced term could additionally depend on $\widetilde{a}_2$ and $\widetilde{a}_4$ (which annotate AST nodes that are destroyed by the reduction). A transducer F_β satisfying these conditions could be defined as:

$$\lambda r, p, \ell.\ \mathsf{case}\ \ell\ \mathsf{of}\ \{\epsilon \mapsto v_\beta\ (p\ \epsilon)\ (p\ [1])\ (p\ (r\ \epsilon)); \_ \mapsto p\ (r\ \ell)\}$$

where v_β is an arbitrary term combining three annotations into one, $(p\ \epsilon)$ and $(p\ [1])$ return the annotations corresponding to $\widetilde{a}_4$ and $\widetilde{a}_2$ in the example. The argument r receives a relocation function in the form $\beta\ \rho(f, x.M)$, provided by the standard provenance-carrying semantics, which is used to retrieve the "natural" annotation arising from the substitution of a labelled term into another labelled term.

These examples show that while manipulating raw provenance labellings is tricky, we can keep things simple by providing functions like $v_{\mathtt{fst}}$ and v_β, which combine a few distinguished annotations and propagate all the others. Based on such considerations, we provide the following *lifting framework* that lifts functions v_κ provenance annotations to provenance transducers F_κ:

$$\begin{aligned}
F_\beta(v_\beta) &:= \lambda r, p, \ell.\ \mathsf{case}\ \ell\ \mathsf{of} \\
&\qquad \{\epsilon \mapsto v_\beta\ (p\ \epsilon)\ (p\ [1])\ (p\ (r\ \epsilon)); _ \mapsto p\ (r\ \ell)\} \\
F_{\mathtt{fst}}(v_{\mathtt{fst}}) &:= \lambda p, \ell.\ \mathsf{case}\ \ell\ \mathsf{of} \\
&\qquad \{\epsilon \mapsto v_{\mathtt{fst}}\ (p\ \epsilon)\ (p\ [1])\ (p\ [1;1]); _ \mapsto p\ (1 \# 1 \# \ell)\} \\
F_{\mathsf{snd}}(v_{\mathsf{snd}}) &:= \lambda p, \ell.\ \mathsf{case}\ \ell\ \mathsf{of} \\
&\qquad \{\epsilon \mapsto v_{\mathsf{snd}}\ (p\ \epsilon)\ (p\ [1])\ (p\ [1;2]); _ \mapsto p\ (1 \# 2 \# \ell)\} \\
F_{\mathtt{tt}}(v_{\mathtt{tt}}) &:= \lambda p, \ell.\ \mathsf{case}\ \ell\ \mathsf{of} \\
&\qquad \{\epsilon \mapsto v_{\mathtt{tt}}\ (p\ \epsilon)\ (p\ [1])\ (p\ [2]); _ \mapsto p\ (2 \# \ell)\} \\
F_{\mathtt{ff}}(v_{\mathtt{ff}}) &:= \lambda p, \ell.\ \mathsf{case}\ \ell\ \mathsf{of} \\
&\qquad \{\epsilon \mapsto v_{\mathtt{tt}}\ (p\ \epsilon)\ (p\ [1])\ (p\ [3]); _ \mapsto p\ (3 \# \ell)\}
\end{aligned}$$

$$
\begin{aligned}
F_\bullet(v_\bullet) &:= \lambda p, \ell.\ \mathsf{case}\ \ell\ \mathsf{of} \\
&\qquad \{\epsilon \mapsto v_\bullet\ (p\ \epsilon)\ (p\ [1])\ (p\ [2]); \_ \mapsto p\ (2\ \#\ \ell)\} \\
F_\sim(v_\sim) &:= \lambda r, p, \ell.\ \mathsf{case}\ \ell\ \mathsf{of} \\
&\qquad \{\epsilon \mapsto v_\sim\ (p\ \epsilon)\ (p\ [1])\ (p\ (r\ \epsilon)); \_ \mapsto p\ (r\ \ell)\} \\
F_\epsilon(v_\epsilon) &:= \lambda p, \ell.\ \mathsf{case}\ \ell\ \mathsf{of} \\
&\qquad \{\epsilon \mapsto v_\epsilon\ (p\ \epsilon)\ (p\ [1])\ (p\ [2]); \_ \mapsto p\ (2\ \#\ \ell)\} \\
F_\#(v_\#) &:= \lambda r, p, \ell.\ \mathsf{case}\ \ell\ \mathsf{of} \\
&\qquad \{\epsilon \mapsto v_\#\ (p\ \epsilon)\ (p\ [1])\ (p\ (r\ \epsilon)); \_ \mapsto p\ (r\ \ell)\} \\
F_\oplus(v_\oplus) &:= \lambda \overrightarrow{x_n}, p, \ell.\ \mathsf{case}\ \ell\ \mathsf{of} \\
&\qquad \{\epsilon \mapsto v_\oplus\ \overrightarrow{x_n}\ (p\ \epsilon)\ (p\ [1])\ (p\ [2]) \cdots (p\ [n]) \\
&\qquad ; \_ \mapsto \epsilon\}
\end{aligned}
$$

$$F(v) := \langle F_\beta(v_\beta), \ldots, F_\oplus(v_\oplus)\rangle$$

We will now use the lifting framework to define where-provenance, expression provenance, and dependecy provenance views. We will also state their correctness properties, whose proofs can be found in the extended version of this paper.

4.1 Where-provenance

The where-provenance view W uses label-propagating transducers; any optional type of labels $\sim\tau$ is allowed. When a reduction copies certain data, the corresponding transducer will propagate the labels along with the data; when a reduction transforms the term beyond recognizability (for example, when reducing primitive operations), there is no label to propagate, therefore the corresponding transducer will produce the default, uninformative label $\bullet$.

$$
\begin{aligned}
w_\oplus &:= \lambda \overrightarrow{x_n}, \widetilde{a}, \overrightarrow{\widetilde{a}_n}.\bullet \\
w_\kappa &:= \lambda \_, \_, \widetilde{a}.\widetilde{a} \qquad\qquad (\kappa \neq \oplus) \\
W &:= F(w)
\end{aligned}
$$

Example 4.1. Let $f := \lambda x, y.\mathsf{case}\ x \stackrel{?}{=} y\ \mathsf{of}\ \{\mathtt{tt} \mapsto x; \mathtt{ff} \mapsto y + 1\}$. We map $f\ 2$ to a list $[1; 2; 3]$:

$$\mathsf{map}\ (f\ 2)[1; 2; 3] \stackrel{*}{\hookrightarrow} [2; 2; 4]$$

Now consider the following provenance labelling L assigning annotations $\widetilde{a}, \widetilde{a}_1, \widetilde{a}_2, \widetilde{a}_3$ as follows:

$$\mathsf{map}\ (f\ 2^{\widetilde{a}})\ [1^{\widetilde{a}_1}; 2^{\widetilde{a}_2}; 3^{\widetilde{a}_3}]$$

(subexpressions without an annotation are considered to labelled by $\bullet$). A where-provenance reduction acts as follows:

$$\mathsf{map}\ (f\ 2^{\widetilde{a}})\ [1^{\widetilde{a}_1}; 2^{\widetilde{a}_2}; 3^{\widetilde{a}_3}] \stackrel{*}{\hookrightarrow} [2; 2^{\widetilde{a}}; 4]$$

In the output list, only the second element is annotated with a, meaning that it was copied from the argument of f; the other two elements were not copied, but obtained by incrementing previous values in the same position, and thus receive the $\bullet$ annotation.

As a general rule, after each execution step we expect subexpressions labelled with $\sim a$ to be copied from parts of the original expression having the same label. The actual well-behavedness property of where-provenance, in our setting, is less naïve than that: consider the reduction

$$(\lambda x.(\lambda y.(x\ y)^a))\ N \hookrightarrow \lambda y.(N\ y)^a$$

The subexpression $(N\ y)$ is indeed derived from the similarly labelled $(x\ y)$, but the two are only equal up to the substitution $[N/x]$. The relation between the two expressions can be made formal:

Definition 4.2. $M \sqsubseteq N \iff \exists s.M[s] \stackrel{*}{\hookrightarrow} N$, where s is a substitution.

Lemma 4.3. (1) $\sqsubseteq$ *is an order relation;*
(2) *if* $M \sqsubseteq N$ *and* $N \stackrel{*}{\hookrightarrow} R$, *then* $M \sqsubseteq R$.

We can then prove that W satisfies the following well-behavedness property of where-provenance:

Theorem 4.4 (well-behavedness of W). *Suppose* $W \vdash M \stackrel{T}{\hookrightarrow} N$: *then for all labellings* L *and* $\ell \in \mathrm{locs}(N)$, *if* $T\ L\ \ell \rightsquigarrow \sim a$, *there exists* $\ell' \in \mathrm{locs}(M)$ *such that* $L\ \ell' \rightsquigarrow \sim a$ *and* $M\big|_{\ell'} \sqsubseteq N\big|_\ell$.

4.2 Expression provenance

To define expression provenance, we allow provenance labels to be simplified syntax trees with constants or primitive annotations as leaves and primitive operation symbols as inner nodes. More formally, we assume a type of labels τ containing the following elements:

$$a ::= \bar{b} \mid k_c \mid k_\oplus(\overrightarrow{a_n})$$

where b belongs to a type of basic annotations τ_b, and k_c and $k_\oplus$, defined for all constants c and all n-ary basic operations $\oplus$, are reified representations of language expressions. As usual, τ will be wrapped in an optional type $\sim\tau$.

Accordingly, a numeric value, say 2, could have several possible labels: a label $\sim\bar{b}$ means that 2 was copied from a part of the input with the same label; a label $\sim k_+(a_1, a_2)$ means it was obtained by adding together two values originally labelled with $\sim a_1$ and $\sim a_2$; the label $\bullet$, as usual, provides no information; finally the label $\sim k_2$, merely indicates a literal 2, but does not specify its provenance otherwise, therefore it can be considered a variant of $\bullet$.

The expression provenance view can be defined by means of a simple variation on where-provenance. The only transducer that needs a different definition, unsurprisingly, is the one associated with the evaluation of primitive operations ($e_\oplus$):

$$
\begin{aligned}
e_\oplus &:= \lambda \overrightarrow{x_n}, \widetilde{a}, \overrightarrow{\widetilde{a}_n}.k_\oplus(\overrightarrow{\widetilde{a_n} * x_n}) \\
e_\kappa &:= \lambda \_, \_, \widetilde{a}.\widetilde{a} \qquad\qquad (\kappa \neq \oplus)
\end{aligned}
$$

$$E := F(e)$$

In the definition, we use the following $*$ operation:

$$(\widetilde{a}) * x = \begin{cases} \sim k_x & \text{if } \widetilde{a} = \bullet \\ \widetilde{a} & \text{else} \end{cases}$$

where k_x is the label corresponding to the constant x (remember that primitive operations are reduced when applied to constants, thus we must have $x = c$ for some c). This allows us to conjure an informative annotation for constants that have not been given a provenance.

Example 4.5. We start with the same labelling as in the where-provenance example:

$$\texttt{map}\ (f\ 2^{\widetilde{a}})\ [1^{\widetilde{a}_1}; 2^{\widetilde{a}_2}; 3^{\widetilde{a}_3}]$$

If we evaluate the term according to expression provenance, we obtain the following labelled value:

$$[2^{k_+(\widetilde{a}_1,k_1)}; 2^{\widetilde{a}}; 4^{k_+(\widetilde{a}_3,k_1)}]$$

Just like in where-provenance, $2^{\widetilde{a}}$ indicates a term that was copied from the input; the term $2^{k_+(\widetilde{a}_1,k_1)}$, instead, has been obtained by adding together $1^{\widetilde{a}_1}$ and an unannotated literal 1.

Expression provenance annotations should allow us to recompute the annotated expression. To make this formal, let us consider metalanguage functions h mapping basic annotation values in τ_b to arbitrary language values. We then define the extension of such an h to full annotations as the metalanguage function $\hat{h}$ mapping values in τ to arbitrary language values, as follows:

$$\hat{h}(\bar{b}) = h(b)$$
$$\hat{h}(k_c) = c$$
$$\hat{h}(k_\oplus(\widetilde{a}_1, \ldots, \widetilde{a}_n)) = \hat{\oplus}(\hat{h}(\widetilde{a}_1), \ldots, \hat{h}(\widetilde{a}_n))$$

We then introduce *consistent mappings* (a modified version of the analogous concept used in [1]) as those functions h which agree with a certain annotated expression.

Definition 4.6. Let M be a term, and L a provenance labelling for M of type Prov_τ. We say that h is a *consistent mapping* for $L \triangleright M$ (and write $h \Vdash L \triangleright M$) if and only if for all $\ell \in \text{locs}(M)$ there exist an annotation $\widetilde{a}$ such that $L\ \ell \rightsquigarrow \widetilde{a}$ and, if $\widetilde{a} = {\sim}a'$, then $\hat{h}(a') \sqsubseteq M\big|_\ell$.

Finally, we prove that reduction under the view E preserves consistent mappings.

Theorem 4.7 (well-behavedness of E). *If $h \Vdash L \triangleright M$ and $E \vdash M \overset{T}{\hookrightarrow} N$, then $h \Vdash T\ L \triangleright N$.*

4.3 Dependency provenance

Dependency provenance associates to each expression location an annotation containing a set of labels. A full definition of dependency provenance would therefore require us to encode sets as a type of the language, providing at the same time an implementation of standard set operations. We could for example use the list encoding of sets, which is a simple programming exercise; for the purposes of this paper, we will abstract from the actual definition of the type of dependency annotations, assuming that a sound implementation of set operations like union and membership test exists. In particular, we assume that the default annotation $\bullet$ be interpreted as the empty set of labels $\emptyset$ rather than the "none" optional value.

The dependency view then merely amounts to taking the union of all the dependency sets involved in a reduction:

$$d_\oplus := \lambda\overrightarrow{x_n}, \overrightarrow{\widetilde{a}_{n+1}}. \bigcup \overrightarrow{\widetilde{a}_{n+1}}$$
$$d_\kappa := \lambda\widetilde{a}_1, \widetilde{a}_2, \widetilde{a}_3. \widetilde{a}_1 \cup \widetilde{a}_2 \cup \widetilde{a}_3 \qquad (\kappa \neq \oplus)$$

$$D := F(d)$$

Example 4.8. We adapt our example to dependency provenance by extending the language of provenance labels to accept sets of annotations:

$$\texttt{map}\ \langle f\ 2^{\{a\}}, [1^{\{a_1\}}; 2^{\{a_2\}}; 3^{\{a_3\}}]\rangle$$

(where subexpressions without an annotation are considered to labelled by $\emptyset$). After evaluation, we get the following dependency provenance labelling:

$$[2^{\{a_1,a\}}; 2^{\{a_2,a\}}; 4^{\{a_3,a\}}]$$

The new labelling reflects the fact that each element of the list depends upon both the corresponding element in the source list, and the argument to the function f.

To state the well-behavedness property for the view D we use a relation $L \triangleright M \approx_a L' \triangleright M'$ that holds when the annotated terms $L \triangleright M$ and $L' \triangleright M'$ are *quasi-equal*, up to subterms annotated with a label a (e.g. $\langle 1, 2^{\mathcal{S}}\rangle \approx_a \langle 1, 3^{\mathcal{S}'}\rangle$ provided that $a \in \mathcal{S} \cap \mathcal{S}'$). The idea is that, if starting with $L \triangleright M$ we perform a reduction step $D \vdash M \overset{T}{\hookrightarrow} N$ whose redex is entirely guarded by the label a, we obtain a term that is still quasi-equal to $L' \triangleright M'$. If instead the redex is not guarded by a, then we can find a reduction $D \vdash M' \overset{T'}{\hookrightarrow} N'$ such that $T\ L \triangleright N \approx_a T'\ L' \triangleright N'$.

To indicate reductions guarded and not guarded by an annotation, we use the notation:

$$D \vdash (L \triangleright M) \overset{T}{\hookrightarrow}_a N$$
$$D \vdash (L \triangleright M) \overset{T}{\hookrightarrow}_{\bar{a}} N$$

(a formal definition is given in the extended version of this paper).

Theorem 4.9 (well-behavedness of D). *Suppose $L \triangleright M \approx_a L' \triangleright M'$. Then:*

1. *If $D \vdash (L \triangleright M) \overset{T}{\hookrightarrow}_a N$, we have $T\ L \triangleright N \approx_a L' \triangleright M'$.*
2. *If $D \vdash (L \triangleright M) \overset{T}{\hookrightarrow}_{\bar{a}} N$, there exist T', N' such that $D \vdash M' \overset{T'}{\hookrightarrow} N'$ and $T\ L \triangleright N \approx_a T'\ L'. \triangleright N'$.*

Admittedly, this property is weaker than the *dependency-correctness* of [10], as we do not consider redexes that are *partially guarded* by a, such as $(f(x).M)^{\{a\}}\ N$. We believe that full dependency-correctness also holds, but cannot be proved without resorting to a more complex argument involving well-typedness; we thus leave this proof as future work.

5 SELF-INSPECTION

The language described in Section 3, together with the view definition framework of Section 4, can be compared to TML. Although we have simplified the type system by renouncing recursive types, the two languages are quite similar; the provenance extraction framework can also be compared to our view definition framework. Our approach, however, does provide a few enhancements:

- our small-step semantics is not limited to call-by-value evaluation, like their big-step definition, but accomodates a variety of evaluation strategies;
- provenance views and provenance data are expressed in the same language as the programs whose provenance is being considered;

$$\begin{array}{lrl} \text{Expressions} & L, M, N & ::= \ldots \mid \iota(F, L \triangleright M) \\ \text{Views} & F & ::= \langle M_\beta, M_{\mathtt{fst}}, M_{\mathsf{snd}}, M_\bullet, M_\sim, \\ & & \quad M_\epsilon, M_\#, M_\oplus, M_\iota \rangle \end{array}$$

$$\mathsf{locs}(\iota(F, L \triangleright M)) = \{\epsilon\}$$

$$\frac{\Gamma \vdash F : \mathcal{T}_\sigma \qquad \Gamma \vdash L : \mathsf{Prov}_\sigma \qquad \Gamma \vdash M : \tau}{\Gamma \vdash \iota(F, L \triangleright M) : \tau \times \mathsf{Prov}_\sigma}$$

$$\mathcal{T}_\sigma(\iota) = \mathsf{Trans}_\sigma$$
$$F(v_\iota) = \lambda p, \ell.\mathsf{case}\ \ell\ \mathsf{of}\ \{\epsilon \mapsto v_\iota\ (p\ \epsilon); \_ \mapsto \bullet\}$$

$$\frac{G \vdash M \overset{T}{\hookrightarrow} N}{F \vdash \iota(G, L \triangleright M) \overset{I}{\hookrightarrow} \iota(G, T\ L \triangleright N)} \qquad \frac{\mathrm{FV}(V) = \emptyset}{F \vdash \iota(G, L \triangleright V) \overset{F_\iota}{\hookrightarrow} \langle V, L \rangle}$$

Figure 7: Syntax, semantics, and typing of inspection

- in our language, we need not take the extra step of materializing the execution trace of a program before extracting its provenance;
- as a consequence, to compute the provenance of a program we do not have to replay its execution trace.

The most important reason why these features matter is that they make it possible for our language to be extended with an introspective inspection operation, which we will denote by the following syntax:

$$\iota(F, L_M \triangleright M)$$

The intended semantics of inspections, informally, is that M will be evaluated to V; at the same time, the provenance view F will be applied to L_M to obtain a new labelling L_V for V; finally, evaluation will produce the pair $\langle V, L_V \rangle$, making both the value and its provenance available for the rest of the program.

Provenance inspection is introduced by means of the extension described in Figure 7. From the point of view of location lookup, inspections are treated like opaque boxes: the locs operation only returns the singleton $\{\epsilon\}$ when applied to an inspection, so access to its syntactic subterms is not allowed. The typing rule checks that the type of the provenance view F agree with that of the provenance labelling L; the term under inspection can be of any type.

Two different reduction rules are applied depending on whether the term under inspection has been fully evaluated or not. When the term M under inspection is not a value, it can be reduced to N, by means of the first reduction rule, using the view G specified by the inspection; this returns a transducer T that we apply to the initial provenance labelling L to obtain a new labelling $T\ L$ suitable for N.

When, after a certain number n of reduction steps, the inspected term becomes a value V, with a certain provenance $L' = T_n\ (T_{n-1} \cdots (T_0\ L) \cdots)$; then the inspection can be concluded by a final step, using the second rule, which merely returns the value V together with its labelling L'. The premise requiring that V should be a closed term is needed because free variables might be replaced by any expression, including expressions containing redexes, and in such a case the inspection should continue; however, since we do not allow reduction inside binders, this check is only necessary if we are interested in evaluating open terms.

Unlike TML, the syntax of PIC allows free nesting of provenance inspections. For this reason we also need to explain what an external observer can see when an inspection is performed. Here we need to make a choice and decide whether inner reductions can be observed or not. Since inspections come with a local provenance labelling, they cannot be easily reconciled with a non-local provenance view; then the simplest policy, which we adopt, is to make inner reductions *not* observable: this is obtained by returning the identity $I = \lambda x.x$ as the transducer for inner reductions. The final inspection step, instead, can be observed: it employs the transducer F_ι, which is defined in the outer view F.

The presence of nested inspections has implications for privacy and confidentiality: on one hand, inner inspections decide whether to share provenance information with external observers, and to what extent; on the other hand, the external observer is aware of hidden computations thanks to the transducer F_ι. This interaction is not unexpected: implications of provenance for confidentiality, privacy, availability, and other security properties have been investigated by some previous work ([6, 9, 13, 17]). We will now elaborate further by considering a security application example.

5.1 Example: inspection of dynamic linking

In the previous sections we have discussed provenance as a property of data, and how it evolves during the execution of a program, but we have not provided concrete examples of how provenance can be used in a concrete setting.

We will here provide a demonstration of how provenance can be used to detect possible security issues. Our example considers dynamic linking, a technique widely available in modern operating systems, which allows the linking of object code from several sources to happen at load time (or in certain cases at run time), rather than compile time. When an executable file is built from source code containing references to functions defined in a shared (or *dynamic-link*) library, those references remain unresolved; instead, special directives are added to the executable file header, including a record of the required library functions (sometimes called *import table*). When loading the executable file, the operating system is expected to find the required shared libraries and load them into memory beside the program code, and link the two by resolving the function references declared in the import table.

Dynamic linking allows a system to avoid duplication of frequently used code, both on disk and in RAM (when two processes using the same library are running concurrently, the memory image of the library can be shared across the two addressing spaces by means of paging). While 30 years ago programs usually employed less than 5 dynamic-link libraries, today's software systems can require several even hundreds of libraries provided by different developers. Libraries from trusted sources may coexist with libraries from untrusted sources, and they may use each other's services.

To model a simple dynamic linker, we assume a type $[\mathbf{N}] \supset \mathbf{N}$ for shared library functions – an acceptable assumption, given that executable file formats often enforce a loose or trivial typing discipline for these functions. A store of type $[\mathbf{id} \times ([\mathbf{N}] \supset \mathbf{N})]$ associates function identifiers to the corresponding shared function.

A program comprises executable code, which we model as a function of type $[[\mathbf{N}] \supset \mathbf{N}] \supset [\mathbf{N}] \supset \mathbf{N}$, and an import table, which in our setting is a list of identifiers of shared functions that should be loaded. The code receives as its first parameter a list of concrete functions matching the imported identifiers from the import table, while the second argument must be a list of input parameters for the program; at the end, a natural number is returned.

The goal of the dynamic linker is to serve the list of imported functions to the program code. If the store and the import table are both ordered by increasing values of identifiers, this can be achieved by the following procedure:

$$\begin{aligned}&\texttt{dynalink} := \lambda \mathit{store}, \mathit{code}, \mathit{imptab}.\\ &\quad \texttt{let } \mathit{imps} := \texttt{filter\_map } (\lambda\langle x, f\rangle.x \in \mathit{imptab})\ (\lambda\langle x, f\rangle.f)\ \mathit{store}\\ &\quad \texttt{in } \mathit{code}\ \mathit{imps}\end{aligned}$$

where `filter_map` $p\ f\ l$ ignores the elements of the list l that do not satisfy the predicate p, but is otherwise the same as `map` $f\ l$.

We now consider a toy program whose purpose is to store passwords to permanent memory. The program employs *salt* and a hash function to make dictionary attacks unfeasible in case a malicious agent gained access to the password file:

$$\begin{aligned}\texttt{savepw} := &\lambda \mathit{imps}, \mathit{pw}.\\ &\texttt{let } \mathit{salt} := \texttt{\$RND [] in}\\ &\texttt{\$WRITE } [\mathit{salt}];\\ &\texttt{\$WRITE } [\texttt{\$HASH } [(\mathit{salt} \| \mathit{pw})]]\end{aligned}$$

where the semi-colon is an infix binary operator on natural numbers simply returning the second argument, and $\|$ combines two natural numbers into one, for example by concatenating their bit representations, or by means of Cantor's encoding of pairs. `HASH`, `RND`, and `WRITE` are identifiers for three imported functions: the import table for `savepw` is thus the following:

$$[\texttt{HASH}; \texttt{RND}; \texttt{WRITE}]$$

Finally, we write `$ID` as syntactic sugar for an imported function call; concretely, this involves a lookup by ordinal number in the import list *imps*: e.g `$WRITE` = `nth` *imps* 2 (where `nth` returns the element of a list referred to by a zero-based index).

Suppose that we do not have access to the source code of `savepw`: we can still read its import table and see that the program requires the use of a cryptographic hash function, a random number generator, and persistent storage. This is not suspicious, but for increased trust we may want to inspect the operation of `savepw` and verify that only a properly encrypted password has been stored.

For this purpose, we use the following provenance label type:

$$\begin{aligned}\sim\texttt{label} ::= \bullet \mid &\sim\texttt{private} \mid \sim\texttt{random} \mid \sim\texttt{rndpriv} \mid \sim\texttt{breach}\\ \mid &\sim\texttt{standard} \mid \sim\texttt{unsafe} \mid \sim\texttt{oneway}\end{aligned}$$

The first four labels are used to express the provenance of data: the default $\bullet$ labels data that is public or irrelevant as far as confidentiality goes; data labelled by `~private`, on the contrary, is confidential; `~random` is used for non-private data coming from a random source; finally, `~rndpriv` labels data containing both private and random information. A label `~breach`, also used to annotate data, is only generated when a potential security breach is detected.

The last three labels are used to annotate functions:

- `~standard` is used for pure functions that manipulate the input in an unknown way; they can throw away part of the input, but may not create private data out of thin air: since the randomized part of the input may not be preserved, this label combines with labelled data as follows

$$\begin{aligned}\sim\texttt{standard}(\bullet) &= \bullet\\ \sim\texttt{standard}(\sim\texttt{random}) &= \bullet\\ \sim\texttt{standard}(\sim\texttt{private}) &= \sim\texttt{private}\\ \sim\texttt{standard}(\sim\texttt{rndpriv}) &= \sim\texttt{private}\\ \sim\texttt{standard}(\sim\texttt{breach}) &= \sim\texttt{breach}\end{aligned}$$

- `~unsafe` annotates functions that should not receive private data, because they contain untrusted code with side effects (e.g. writing to disk, sending data over a network, displaying information on a terminal): it behaves as follows

$$\begin{aligned}\sim\texttt{unsafe}(\bullet) &= \bullet\\ \sim\texttt{unsafe}(\sim\texttt{random}) &= \bullet\\ \sim\texttt{unsafe}(\sim\texttt{private}) &= \sim\texttt{breach}\\ \sim\texttt{unsafe}(\sim\texttt{rndpriv}) &= \sim\texttt{breach}\\ \sim\texttt{unsafe}(\sim\texttt{breach}) &= \sim\texttt{breach}\end{aligned}$$

- `~oneway` is similar to `standard` but is used with functions known to be *one-way*; thus when applied to randomized private data, it produces non-confidential output:

$$\begin{aligned}\sim\texttt{oneway}(\bullet) &= \bullet & \sim\texttt{oneway}(\sim\texttt{random}) &= \bullet\\ \sim\texttt{oneway}(\sim\texttt{private}) &= \sim\texttt{private} & \sim\texttt{oneway}(\sim\texttt{rndpriv}) &= \bullet\\ \sim\texttt{oneway}(\sim\texttt{breach}) &= \sim\texttt{breach}\end{aligned}$$

This annotation propagation policy can be implemented as a provenance view S. The labelling $L_{\texttt{store}}$ for the shared library functions will usually be fixed, and we assume that the concrete functions `hash`, `rnd`, `write` are labelled by `oneway`, `random`, `unsafe`. Given the password `pw` (which, for simplicity, will be a natural number), we make it confidential by defining its labelling as

$$L_{\texttt{pw}} := \lambda\ell.\texttt{case } \ell \texttt{ of } \{\epsilon \mapsto \sim\texttt{private}; \_ \mapsto \bullet\}$$

As previously mentioned, an outer inspection cannot see what happens within an inner one, so an additional task of S, besides propagating annotations, is to flag inner inspections as possible security breaches, by means of the transducer

$$S_\iota := \lambda\_, \_.\texttt{breach}$$

Now we can apply the dynamic linker to `savepw` and try running it by means of the syntax:

$$M = \texttt{dynalink store savepw [HASH; RND; WRITE] pw}$$

where the provenance labelling for M can be obtained by combining $L_{\texttt{store}}$, $L_{\texttt{pw}}$, and trivial labellings for all the other parts of M:

$$L = \mathfrak{I}_2^{\text{Prov}}\ (\mathfrak{I}_2^{\text{Prov}}\ (\mathfrak{I}_2^{\text{Prov}}\ (\mathfrak{I}_2^{\text{Prov}}\ \bot\ L_{\texttt{store}}\ \bullet)\ \bot\ \bullet)\ \bot\ \bullet)\ L_{\texttt{pw}}\ \bullet$$

Finally, we perform an inspection:

$$\iota(S, L \triangleright M) \overset{*}{\hookrightarrow} \langle 0, L'\rangle$$

where 0 is the value returned by the call to `WRITE`, and $L'\ \epsilon \overset{*}{\hookrightarrow} \sim\texttt{public}$ confirms that no security breaches were detected.

6 CONCLUSIONS

The language PIC that we have described represents provenance by means of labelling functions. This seems to be a natural approach in a calculus allowing provenance to be manipulated as a first-class expression because in this way all the provenance annotations for the same term are gathered in the same place. Other approaches based on annotation propagation would scatter this information across various subterms, and thus require additional effort to extract the provenance and separate it from the term it describes. A key element of our language is the presence of a relocation operator, whose lack makes it impossible to compute provenance past beta-reductions in other languages with inspection [2, 20]

Provenance labellings also seems to have a relatively elegant, albeit slightly low-level, theory of combinators, which guides the definition of the provenance-aware semantics. Although defining provenance views by combining annotations locally (as in TML) is simpler than doing so by handling whole labellings (as in PIC), we have provided a simplified framework to bridge this gap.

Although PIC is a pure functional language, we envision extending it with imperative constructs. Experience in the related area of program slicing ([21]) tells us that such an extension would not be straightforward, but might provide a good starting point.

We have chosen to make provenance inspection local: an outer inspection cannot observe the computation happening within an inner inspection; this ensures a certain confidentiality of metadata, which cannot be shared by accident with the public. However, this is not the only possible way to handle nested inspections: a *transparent* provenance inspection could also be defined using the following evaluation rule:

$$\frac{G \vdash M \xhookrightarrow{G[\mathfrak{R}]} N}{F \vdash \iota(G, L \triangleright M) \xhookrightarrow{\Im_{1.1}\,(F[\mathfrak{R}])} \iota(G, T\ L \triangleright N)}$$

where $\mathfrak{R}$ identifies the reduction rule used in the premise, and $F[\mathfrak{R}]$ returns the trasducer for rule $\mathfrak{R}$ according to the view F: this allows reduction within ι to be treated like any other congruence. Apart from a slight notational complication, this extension appears straightforward. Another possibility, given $\iota(G, L \triangleright M)$ is to make M invisible to an external observer, but to allow an outer inspection to provide a labelling for G and L. Indeed, since a provenance view contains standard language functions, it needs not be statically determined: it is perfectly admissible to receive provenance transducers as the arguments of a program, and use them to construct a view to be used in an inspection. In other words, views may have a non-trivial provenance, which could provide a reason to inspect the *provenance of provenance*. This concept appears not to be entirely new (see e.g. [16, 18]); further investigation of relations between nested inspections and provenance of provenance will be the subject of future work.

REFERENCES

[1] U. A. Acar, A. Ahmed, J. Cheney, and R. Perera. 2013. A core calculus for provenance. *Journal of Computer Security* 21 (2013), 919–969.

[2] F. Bavera and E. Bonelli. 2015. Justification logic and audited computation. *Journal of Logic and Computation* (2015). Published online, June 19, 2015.

[3] David A. Bearman. 1985. The Power of the Principle of Provenance. *Archivaria* 21 (1985), 14âĂŞ27.

[4] Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. 2005. An annotation management system for relational databases. *VLDB Journal* 14, 4 (2005), 373–396.

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

[6] Uri Braun, Avraham Shinnar, and Margo I. Seltzer. 2008. Securing Provenance. In *HotSec*.

[7] Peter Buneman, James Cheney, and Stijn Vansummeren. 2008. On the expressiveness of implicit provenance in query and update languages. *ACM Transactions on Database Systems* 33, 4 (November 2008), 28.

[8] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *ICDT (LNCS)*. 316–330.

[9] J. Cheney. 2011. A formal framework for provenance security. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 281–293.

[10] James Cheney, Amal Ahmed, and Umut a. Acar. 2011. Provenance As Dependency Analysis. *Mathematical. Structures in Comp. Sci.* 21, 6 (Dec. 2011), 1301–1337. https://doi.org/10.1017/S0960129511000211

[11] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.* 25, 2 (2000), 179–227.

[12] Susan B. Davidson and Juliana Freire. 2008. Provenance and Scientific Workflows: Challenges and Opportunities. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 1345–1350. https://doi.org/10.1145/1376616.1376772

[13] Susan B. Davidson, Sanjeev Khanna, Sudeepa Roy, and Sarah Cohen Boulakia. 2010. Privacy Issues in Scientific Workflow Provenance. In *Proceedings of the 1st International Workshop on Workflow Approaches to New Data-centric Science (Wands '10)*. ACM, New York, NY, USA, Article 3, 6 pages. https://doi.org/10.1145/1833398.1833401

[14] Boris Glavic and Gustavo Alonso. 2009. Provenance for Nested Subqueries. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '09)*. ACM, New York, NY, USA, 982–993. https://doi.org/10.1145/1516360.1516472

[15] Boris Glavic, Renée J. Miller, and Gustavo Alonso. 2013. Using SQL for Efficient Generation and Querying of Provenance Information. In *In Search of Elegance in the Theory and Practice of Computation*. 291–320.

[16] Kwan Hee Han, Seock Kyu Yoo, and Bohyun Kim. 2009. Qualitative and Quantitative Analysis of Workflows Based on the UML Activity Diagram and Petri Net. *WSEAS Trans. Info. Sci. and App.* 6, 7 (July 2009), 1249–1258. http://dl.acm.org/citation.cfm?id=1639420.1639437

[17] Ragib Hasan, Radu Sion, and Marianne Winslett. 2007. Introducing Secure Provenance: Problems and Challenges. In *Proceedings of the 2007 ACM Workshop on Storage Security and Survivability (StorageSS '07)*. ACM, New York, NY, USA, 13–18. https://doi.org/10.1145/1314313.1314318

[18] Zachary Hensley, Jibonananda Sanyal, and Joshua New. 2013. Provenance in Sensor Data Management. *Queue* 11, 12, Article 50 (Dec. 2013), 14 pages. https://doi.org/10.1145/2559899.2574836

[19] Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. 2007. A Formal Model of Dataflow Repositories. In *Proceedings of the 4th International Conference on Data Integration in the Life Sciences (DILS'07)*. Springer-Verlag, Berlin, Heidelberg, 105–121. http://dl.acm.org/citation.cfm?id=1768933.1768947

[20] Wilmer Ricciotti and James Cheney. 2017. Strongly Normalizing Audited Computation. In *26th EACSL Annual Conference on Computer Science Logic (CSL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Valentin Goranko and Mads Dam (Eds.), Vol. 82. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 36:1–36:21. https://doi.org/10.4230/LIPIcs.CSL.2017.36

[21] Wilmer Ricciotti, Jan Stolarek, Roly Perera, and James Cheney. 2017. Imperative functional programs that explain their work. In *ICFP 2017*. In press.

[22] T. R. Schellenberg. 1965. The Principle of Provenance and Modern Records in the United States. *The American Archivist* 28, 1 (1965), 39–41.

[23] Yogesh Simmhan, Beth Plale, and Dennis Gannon. 2005. A survey of data provenance in e-science. *SIGMOD Record* 34, 3 (2005), 31–36.