Foundations of Fine-Grained Explainability

Sylvain Hallé and Hugo Tremblay

Laboratoire d'informatique formelle
Université du Québec à Chicoutimi, Canada

Abstract. Explainability is the process of linking part of the inputs given to a calculation to its output, in such a way that the selected inputs somehow "cause" the result. We establish the formal foundations of a notion of explainability for arbitrary abstract functions manipulating nested data structures. We then establish explanation relationships for a set of elementary functions, and for compositions thereof. A fully functional implementation of these concepts is finally presented and experimentally evaluated.

1 Introduction

Developers of information systems in all disciplines are facing increasing pressure to come up with mechanisms to describe how or why a specific result is produced —a concept called *explainability*. For example, a web application testing tool that discovers a layout bug can be asked to pinpoint the elements of the page actually responsible for the bug [24]. A process mining system finding a compliance violation inside a business process log can extract a subset of the log's sequence of events that causes the violation [46]. Similarly, an event stream processing system can monitor the state of a server room and, when raising an alarm, identify what machines in the room are the cause of the alarm [40]. All these situations have in common that one is not only interested in an oracle that produces a simple Boolean pass/fail verdict from a given input, but also additional information that somehow links parts of this input to the result.

Explainability is currently handled by *ad hoc* means, if at all. Hence, a developer may write a script that checks a complex condition on some input object; however, for this script to provide an explanation, and not just a verdict, extra code must be written in order to identify, organize, and format the relevant input elements that form an explanation for the result. This extra code is undesirable: it represents additional work, is specific to the condition being evaluated, and relies completely on the developer's intuition as to what constitutes a suitable "explanation". Better yet would be a formal framework where this notion would be defined for arbitrary abstract functions, and accompanied by a generic and systematic way of constructing an explanation.

In this paper, we present the theoretical foundations of a notion of explainability. Our model focuses on abstract functions whose input arguments and output values can be composite objects —that is, data structures that may be composed of multiple parts, such as lists. In contrast with existing works on the

subject, which consider relationships between inputs and outputs as a whole, our framework is fine-grained: it is possible to point to a specific *part* of an output result, and construct an explanation that refers to precise *parts* of the input.

Figure 1 illustrates the approach on a simple example. On the left is an input, which in this case is a text log of comma-separated values. Suppose that on this log, one wishes to extract the second element of each line, and check that the average of any three successive values is always greater than 3. It is easy to see that the condition is false, but where in the input log are the locations that cause it to be violated? By applying the systematic mechanism described in this paper, one can construct an explanation that is represented by the graph at the right. We can observe that the false output of the condition (the graph's root) is linked to several leaves that designate parts of the input (character locations in each line of the file, identified by colors). Moreover, the graph contains Boolean nodes, indicating that the explanation may involve multiple elements ("and"), and that alternate explanations are possible ("or").

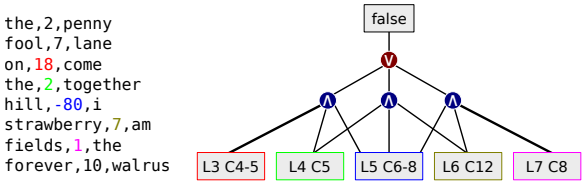

Fig. 1: Left: a simple text file. Right: an explanation graph obtained from the evaluation of a function on this input.

First, in Section 2, we review existing works related to the concept of causality, provenance and taint propagation, which are the notions closest to our concerns. Section 3 lays out the theoretical foundations of our framework: it introduces the notion of parts of composite objects, and formally defines a relation between parts of inputs and outputs of an arbitrary function, called *explanation*. Moreover, it shows how an explanation can be constructed for a function that is a composition of basic functions, by composing their respective explanations. Section 4 then illustrates these definitions by demonstrating what constitutes an explanation on a small yet powerful set of basic functions. Taken together, these results make it possible to easily construct explanations for a wide range of computations.

To showcase the feasibility of this approach, Section 5 presents *Petit Poucet*, a fully-functioning Java library that implements these concepts. The library allows users to create their own complex functions by composing built-in primitives, and can automatically produce an explanation for any result computed by these functions on a given input. Experiments on a few examples provide insight on the time and memory overhead incurred by the handling of explanations. Finally, Section 6 identifies a number of exciting open theoretical questions that arise from our formal framework.

2 Related Work

Explainability can be seen as a particular case of a more general concept called *lineage*, where inputs and outputs of a calculation are put in relation according to various definitions. Related works around this notion can be separated into a few categories, which we briefly describe below.

2.1 Causality

In the field of testing and formal verification of software systems, lineage often takes the form of *causality*. A classical definition is given by Halpern and Pearl [28], based on what is called "counterfactual dependence": A is a cause of B if the absence of A entails the absence of B. According to this principle, some feature of an object causes the failure of a verification or testing procedure if the absence of this feature instead causes the procedure to emit a passing verdict. This notion can be transposed for various types of systems. When a condition is expressed as a propositional logic formula, the cause of the true or false value of the formula can be constructed in the form of an *explanatory sub-model*; informally, it can be thought of as the smallest set of propositional variables whose value implies the value of the formula.

In the case of conditions expressed on state-based systems, the cause of a violation has been taken either as the shortest prefix that guarantees the violation of the property regardless of what follows [23], or as a minimal set of word-level predicates extracted from a failed execution [50]. A platform for hardware formal verification, called RuleBasePE, expands on this latter definition to identify components of a system that are responsible for the violation of a safety property on a single execution trace [5].

Causality has been criticized as an all-or-nothing notion; *responsibility* has been introduced as a refinement on this concept, where the involvement of some element A as the cause of B can be quantified [12]. The problem of deciding causality also has a high computational complexity; as a matter of fact, determining if A is a cause of B in a model only allowing Boolean values to variables is already NP-complete [18]. Tight automata [31, 43] have also been developed to produce minimal counter-examples, which in this case, are sequences of states produced by a traversal of the finite-state machine. Explanatory sub-models can also be extended to the temporal case [19]. Another approach involves the computation of a so-called "minimal debugging window", which is a small segment of the input trace that contains the discovered violation [36].

Finally, distance-based approaches compare a faulty trace with the closest valid trace (according to a given distance metric) [22]; the differences between the two traces are defined as the cause of the failure. A similar principle is applied in a software testing technique called *delta debugging* to identify values of variables in a program that are responsible for a failure [15].

2.2 Provenance in Information Systems

In a completely different direction, a large amount of work on lineage has been done in the field of databases, where this notion is often called *provenance*. A thorough survey of related approaches on provenance [9] reveals that this concept has been studied in several different areas of data management, such as scientific data processing and database management systems.

Research in this field typically distinguishes between three types of provenance. The first type is called *why-provenance* [17]: to each tuple t in the output of a relational database query, why-provenance associates a set of tuples present in the input of the query that helped to "produce" t. *How-provenance*, as its name implies, keeps track not only of what input tuples contribute to the input, but also in which way these tuples have been combined to form the result [21]. For example, a symbolic polynomial like $t^2 + t \cdot t'$ indicates that an output tuple can be explained in two alternative ways: either by using tuple t twice, or by combining t and t'. Finally, *where-provenance* describes where a piece of data is copied from [8]. It is typically expressed at a finer level of granularity, by allowing to link individual values inside an output tuple to individual values of one or more input tuple. One possible way of doing it is through a technique called annotation-propagation, where each part of the input is given symbolic "annotations", which are then percolated and tracked all the way to the output [7].

A recent survey reveals the existence of more than two dozen provenance-aware systems [38]. Where-provenance has been implemented into Polygen [51], DBNotes [11], Mondrian [20], MXQL [48] and Orchestra [29]. The Spider system performs a slightly different task, by showing to a user the "route" from input to output that is being taken by data when a specific database query is executed [10]. The foundations for all these systems are relational databases, where sets of tuples are manipulated by operators from relational algebra, or extensions of SQL.

Taken in a broad sense, we also list in this category various works that aim to develop *explainable* Artificial Intelligence [42]. Models used in AI vary widely, ranging from deep neural networks to Bayesian rules and decision trees; consequently, the notion of what constitutes an explanation for each of them is also very variable, and is at times only informally stated.

2.3 Taint Analysis and Information Flow

A last line of works this time considers the linkage between the inputs and the outputs produced by a piece of procedural code, mostly for considerations of security. Dynamic *taint analysis* consists in marking and tracking certain data in a program at run-time. A typical use case for taint analysis is to check whether sensitive information (such as a password) is being leaked into an unprotected memory location, or if a "tainted" piece of input such as a user-provided string is being passed to a function like a database query without having been sanitized first (opening the door to injection attacks).

In this category, TaintCheck has been developed into a system where each memory byte is associated with a 4-byte pointer to a taint data structure [37]; program inputs are marked as tainted, and the system propagates taint markers to other memory locations during the execution of a program; this concept has been extended to the operating system as a whole in an implementation called Asbestos [47]. Hardware implementations of this principle have also been proposed [16, 44]. Dytan [14] is a notable system for taint propagation and analysis on programs written in assembly language. We shall also mention the GIFT system, as well as a compiler based on it called Aussum [32]. On its side, RIFLE focuses on the information flow [45], while TaintBochs is a system that has been used to track the lifetime of sensitive data inside the memory of a program [13].

The capability of following taint markings on the inputs of a program can be used in many ways. For example, a system called COMET uses taint propagation to improve the coverage of an existing test suite [33]. Taint analysis can also be used to quantify the amount of information leak in a system [34]. Stated simply, the propagation of taint markings can be seen as a form of how-provenance, applied to variables and procedural code instead of tuples and relational queries. Note however that it operates in a top-down fashion: mark the inputs, and track these markings on their way to the output. In contrast, we shall see that our notion of explanation is bottom-up: point at a part of the output, and retrace the parts of the input that are related to it.

3 A Formal Definition of Explanation

The problem we consider can be simply stated. Given an abstract function f and an input argument x, establish a formal relationship that explains $f(x)$ based on x. While this is more or less closely related to the works we presented in the previous section, the solution we propose has a few distinguishing features.

First, x and $f(x)$ may be composite objects made of multiple "parts", and it is possible to relate specific parts of an output to specific parts of an input. Second, if f is itself a composition $g \circ h$, it is possible to construct the input-output relationships of f from the individual input-output relationships of g and h. Third, these relations are not defined *ad hoc* for each individual function, but come as consequences of a general definition. Finally, given x and $f(x)$, determining the parts of x in relation with a given part of $f(x)$ is tractable.

In this section, we establish the formal foundations of our proposed approach. We start by defining an abstract "part-of" relation between abstract objects, based on the notion of *designator*, and establish some properties of this relation. We then propose a definition of *explanation* for arbitrary functions, and discuss how it differs from existing causality and lineage relationships mentioned earlier.

3.1 Object Parts

Let $\mathfrak{U} = \bigcup_i \mathcal{O}_i$ be a union of sets of *objects*; the sets $\mathcal{O}_i$ are called *types*. We suppose that $\mathfrak{U}$ contains a special object, noted $\not\square$, that represents "nothing".

Types are disjoint sets, with the exception of $\boxslash$ which, for convenience, is assumed to be part of every type.

Each type $\mathcal{O}$ is associated with a set $\Pi_{\mathcal{O}}$, whose elements are called *parts*. The set $\Pi_{\mathcal{O}}$ contain functions of the form $\pi : \mathcal{O} \to \mathcal{O}'$; it is expected that $\pi(\boxslash) = \boxslash$ for every such function. In addition, we impose that $\Pi_{\mathcal{O}}$ always contains two other functions defined as $\mathbf{1} : x \mapsto x$, and $\mathbf{0} : x \mapsto \boxslash$. We shall override the term "part" and say that an object $o' \in \mathcal{O}'$ is a part of some other object $o \in \mathcal{O}$ if it is the result of applying some $\pi \in \Pi_{\mathcal{O}}$ to o. A *proper part* is any part other than $\mathbf{1}$ and $\mathbf{0}$; we shall use $\Pi^*_{\mathcal{O}}$ to designate the set of proper parts for a given type. A type $\mathcal{O}$ is called *scalar* if $\Pi^*_{\mathcal{O}} = \emptyset$; otherwise it is called *composite*.

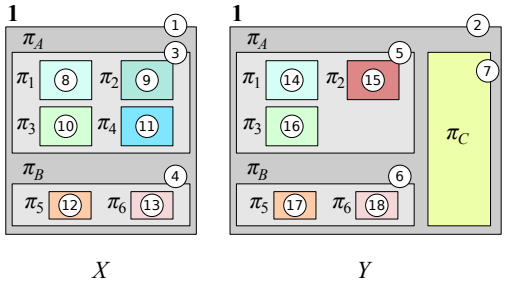

Fig. 2: Illustration of two abstract composite objects of the same type. Composite parts are represented by gray rectangles; scalar parts are represented by colored rectangles.

Figure 2 shows an example of two abstract composite objects X and Y of the same type $\mathcal{O}$, which we suppose has a set of three proper parts called π_A, π_B and π_C. Parts of an object can themselves be of a composite type; we assume here that type A has four parts $\pi_1, \ldots, \pi_4$, type B has two parts π_5, π_6, and type C is a scalar. For example, $\pi_B(X)$ corresponds to the rectangle numbered 4, and $\pi_C(Y)$ is the rectangle numbered 7. Consistent with our definitions, $\mathbf{1}$ designates the whole object, hence $\mathbf{1}(X)$ is the rectangle 1. These parts are not all present in all objects of a given type; for example we see that $\pi_C(X) = \boxslash$.

Parts can be composed in the usual way: if $\pi : \mathcal{O} \to \mathcal{O}'$ and $\pi' : \mathcal{O}' \to \mathcal{O}''$, their composition $\pi \circ \pi'$ is defined as $o \mapsto \pi(\pi'(o))$. This corresponds intuitively to the notion of the part of some part of an object, and will allow us to point to arbitrarily fine-grained portions of input arguments or output values. For example, in Figure 2, we have that $(\pi_1 \circ \pi_A)(X)$ corresponds to the rectangle numbered 8, which indeed corresponds to part π_1 of the part π_A of object X. If $\Pi = \{\pi_1, \ldots, \pi_n\}$ is a set of parts and π is another part, we will abuse notation and write $\Pi \circ \pi$ to mean the set $\{\pi_1 \circ \pi, \ldots, \pi_n \circ \pi\}$.

If π is a part of an object, we shall say that $\pi' \circ \pi$ is a *refinement* of π; inversely, π is a *generalization* of $\pi' \circ \pi$, as it corresponds to a "greater" part of the object. Remark that since $(\mathbf{1} \circ \pi) = (\pi \circ \mathbf{1}) = \pi$, any part π is simultaneously a refinement and a generalization of itself. If π is a proper part of o, all its refinements are also considered as parts of o. The notions of refinement and

generalization can be extended to sets of parts. Given two sets $\Pi = \{\pi_1, \ldots, \pi_m\}$ and $\Pi' = \{\pi'_1, \ldots, \pi'_n\}$, Π is a refinement if there exists an injection μ between Π and Π' such that $\mu(\pi) = \pi'$ if and only if π is a refinement of π'; conversely, Π' is a generalization of Π. Again, a refinement of a set of parts Π picks fewer parts, and more selective parts of an object than Π. In the example of Figure 2, the set $\Pi = \{\pi_1 \circ \pi_A, \pi_C\}$ is a refinement of $\Pi' = \{\pi_A, \pi_5 \circ \pi_B, \pi_C\}$ (the injection here being the two associations $\pi_1 \circ \pi_A \mapsto \pi_A$ and $\pi_C \mapsto \pi_C$).

Given a part π, two objects $o, o' \in \mathcal{O}$ are said to *differ on* π if $\pi(o) \neq \pi(o')$. This can be illustrated in Figure 2. Objects X and Y differ on π_C, since $\pi_C(X) \neq \pi_C(Y)$. They also differ on π_A (since rectangles 3 and 5 are different) and on $\pi_4 \circ \pi_A$ (which produces rectangle 11 for X and $\emptyset$ for Y). However, they do not differ on π_B (rectangles 4 and 6 are identical), and they do not differ on $\pi_3 \circ \pi_A$ (rectangles 10 and 16 are identical). This example shows that if two objects differ on a part π, they do not necessarily differ on a refinement of π (compare π_A and $\pi_1 \circ \pi_A$). Given a set of parts $\Pi = \{\pi_1, \ldots, \pi_n\}$, two objects differ on Π if they differ in some $\pi_i \in \Pi$. Obviously, if objects differ on Π, they also differ on any of its generalizations.

Finally, two parts π and π' *intersect* if there exist two parts π_I and π'_I (different from $\mathbf{0}$) such that whenever $(\pi_I \circ \pi)(o) \neq \emptyset$ and $(\pi'_I \circ \pi')(o) \neq \emptyset$, then $(\pi_I \circ \pi)(o) = (\pi'_I \circ \pi')(o)$. Two sets of parts Π and Π' intersect if at least one of their respective parts intersect. In Figure 2, the sets $\{\pi_A\}$ and $\{\pi_2 \circ \pi_A, \pi_C\}$ intersect (they have in common the part $\pi_2 \circ \pi_A$), while the sets $\{\pi_A\}$ and $\{\pi_5 \circ \pi_B\}$ do not.

3.2 A Definition of Explanation

We are now interested in relations between a part of a function's output, and one or more parts of that function's input. We shall focus on the set of unary functions $f : \mathcal{O} \to \mathcal{O}'$. For the sake of simplicity, functions of multiple input arguments will be modeled as unary functions whose input is a composite type that contains the arguments. These composite types will be used informally to illustrate the notions, and will be formally defined in Section 4.

Definition 1. Let $f : \mathcal{O} \to \mathcal{O}'$ be a function, $\Pi \subseteq \Pi_{\mathcal{O}}$ be a set of parts of the function's input type, and $\pi \in \Pi_{\mathcal{O}'}$ be a part of the function's output type. Consider a set Π such that there exists an object o' that differs from o only on Π, and for which $f(o)$ and $f(o')$ differ on π. We say that Π *explains* π if Π is minimal, meaning that no refinement of Π satisfies the previous condition.

As an example, consider the function $f : \langle x, y \rangle \mapsto xy$, with $x = 1$, $y = 1$. For this particular input object, the part designating the first element of the input is an explanation, as changing it to any other value changes the result of the function; the same argument can be made for the part that designates the second element of the input. Consider now the case where $x = 0$, $y = 1$. This time, the value of the second element is irrelevant: changing it to anything else still produces 0. Therefore, the second element of the input is not an explanation

for the output; the first element of the input alone "explains" the result of 0 in the output.

Based on these simple definitions, we can already put our framework in contrast with some of the papers we surveyed in Section 2. First, note how this definition is different from counterfactual causality [1, 28]. This can be illustrated by considering the previous function, in the case where $x = 0$ and $y = 0$. Argument x is not a counterfactual cause of the output value, as changing it to anything else still produces 0; the same argument shows that y is not a cause of the output either. Therefore, one ends up with the counter-intuitive conclusion that none of the inputs cause the output.

In contrast, there exists a minimal set of parts that satisfies our explanation property, which is the set that contains *both* the first and the second element. Indeed, there exists another input object that differs on both elements, and which produces a different result. This is in line with the intuition that either element is sufficient to explain the null value produced by the function, and that therefore both need to be changed to have an impact. It highlights a first distinguishing feature of our approach: the presence of multiple parts inside a set indicates a form of "disjunction" or "alternate" explanations, something that cannot be easily accounted for in many definitions of causality.[1]

Why-provenance is expressed on tuples manipulated by a relational query [17], but our simple case can easily be adapted by assuming that x, y and $f(\langle x, y\rangle)$ each are tuples with a dummy attribute a. The definition then leads to the conclusion that both x and y are considered to "produce" the output, whereas explanation rather concludes that *either* explains the output; the use of how-provenance [21] would produce a similar verdict. Where-provenance [8] is even less appropriate here, as it makes little sense to ask whether the product of two numbers "copies" any of the input arguments to its output.

Since a single explanation is a set of parts, the set of all explanations is therefore a set of sets of parts. As we have shown, a set of parts intuitively represents an alternative (either part is an explanation). In turn, elements of a set of set of parts represents the fact that each of them is an explanation. Therefore, a concise graphical notation for representing sets of sets of parts are and-or trees, such as the one shown in Figure 3. In the present case, leaves of the tree each represent a (single) object part, while non-leaf nodes are labeled either with "and" ($\wedge$) or "or" ($\vee$). For example, this tree represents the set of sets $\{\{\pi_1\}, \{\pi_2 \circ \pi_3, \pi_5 \circ \pi_6, \pi_4\}, \{\pi_2 \circ \pi_3, \pi_7 \circ \pi_6, \pi_4\}\}$. [2]

4 Building Explanations for Functions

Equipped with these abstract definitions, we shall now establish properties of a handful of elementary functions, namely logical and arithmetic operations, and list manipulations. The reader may find that this section is stating the obvious,

[1] Case in point, in [1] a cause is assumed to be a *conjunction* of assertions of the form $X = x$.

[2] Obviously, there exist multiple equivalent trees for the same set of sets of parts.

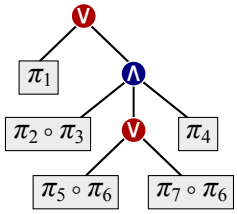

Fig. 3: An example of and-or tree where leaves are object parts.

as many of the results we present correspond to very intuitive notions. The main interest of our approach is that these seemingly trivial conclusions are not defined *ad hoc*, but rather come as consequences of the general definition of explanation introduced in the previous section.

We first consider as *scalar* types the sets of Boolean values $\mathbb{B}$, the set of real numbers $\mathbb{R}$, and the set of characters $\mathbb{S}$. Then, we shall denote by $\mathbb{V}\langle\mathcal{O}_1,\ldots,\mathcal{O}_n\rangle$ the set of vectors of size n, where the i-th element is of type $\mathcal{O}$. Its set of parts $\mathit{\Pi}_{\mathbb{V}\langle\mathcal{O}_1,\ldots,\mathcal{O}_n\rangle}$ contains $\mathbf{1}$ and $\mathbf{0}$, as well as all functions $[i] : \mathbb{V}\langle\mathcal{O}_1,\ldots,\mathcal{O}_n\rangle \to \mathcal{O}_i$ defined as:

$$[i] : \langle o_1,\ldots,o_n\rangle \mapsto \begin{cases} o_i & \text{if } 1 \leq i \leq n \\ \not\square & \text{otherwise.} \end{cases}$$

In other words, the proper parts of a vector are each of its elements. We shall designate for finite vectors of variable length and uniform type $\mathcal{O}$ by $\mathbb{V}\langle\mathcal{O}^*\rangle$. Finally, character strings will be viewed as the type $\mathbb{V}\langle\mathbb{S}^*\rangle$, i.e. finite words over the alphabet of symbols $\mathbb{S}$. We stress that, although our concept of explanation is illustrated on a small set of functions operating on these types, it is by no means limited to these functions or these types.

4.1 Conservative Generalizations

Some of the functions we shall consider return objects that may be composite; these functions introduce the additional complexity that one may want to refer not only to the whole output of the function, but also to a single part of that function's output. What is more, the inputs of these functions can also be composite objects, and explicitly enumerating all their minimal sets of parts for explanation may not be possible.

Take for example the function $f : \mathbb{V}\langle\mathcal{O}\rangle \to \mathbb{V}\langle\mathcal{O}\rangle$, which simply returns its input vector as is. Suppose that we focus on $\pi = [1]$, the first element of the output vector. Clearly, the set $\{[1]\}$, pointing to the first element of the input vector, should be recognized as the only one that explains this output. However, if $\mathcal{O}$ is a composite type, this set is not minimal, and should be further broken down into all the parts of $\mathcal{O}$. Besides being unmanageable, this enumeration also misses the intuition that what explains the first element of the output is simply the first element of the input.

In the following, we employ an alternate approach, which will be to define a set of *conservative generalizations* of the function's input parts. For most functions,

the principle will be the same: given an output part π, we shall define a set of sets of input parts $\Pi = \{\Pi_1, \ldots, \Pi_n\}$, and demonstrate that any input part Π' that explains π on some input intersects with one of the Π_i. It follows that any minimal input part that explains the output is a refinement of one of the Π_i.

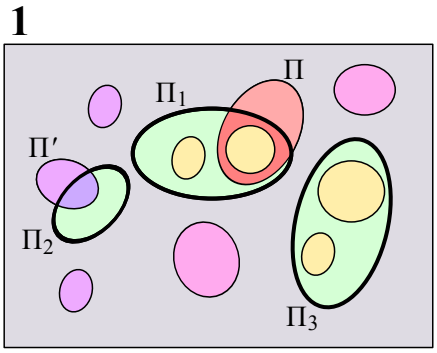

Fig. 4: The sets of parts Π_1, Π_2 and Π_3 (in green) represent a conservative generalization of the minimal sets of explanation parts (yellow circles).

This is illustrated in Figure 4, where the minimal sets of explanations of an input are illustrated in yellow. Here, the set $\{\Pi_1, \Pi_2, \Pi_3\}$ has been identified as the target set of sets of input parts. It is possible to see that if one establishes that any set lying outside of the green ovals is not an explanation, it follows that the minimal sets of parts for explanation are all contained inside one of Π_1, Π_2 and Π_3. Note that this is a generalization, as, for example, Π_2 does not contain any minimal set. However, this generalization is conservative, in the sense that all minimal sets are indeed contained within a green oval.

The goal is therefore to come up with generalizations that are, in a sense, as tight as possible. In the example of function f above, we could easily demonstrate that any set of input parts Π that has an impact on the first element of the output must contain a refinement of the first element of the input, and therefore identify $\Pi = \{\{[1]\}\}$ as a sufficient set that "covers" all the minimal explanation input parts. It so happens that this set corresponds exactly to the intuitive result we expected in the first place: the first element of the input vector contains all the parts that impact the output, and no other part of the input has this property.

4.2 Explanation for Scalar Functions

In the following, we provide the formal definition of conservative generalizations of the explanation relationship for a number of elementary functions. We start with functions performing basic arithmetic operations returning a scalar value. Establishing explanation for addition over a vector of numbers is trivial.

Theorem 1. Let $f : \mathbb{V}\langle\mathbb{R}^*\rangle \to \mathbb{R}$ be the function defined as $\langle x_1, \ldots, x_n\rangle \mapsto x_1 + \cdots + x_n$. For any input $\langle x_1, \ldots, x_n\rangle$, Π explains $\mathbf{1}$ on $\langle x_1, \ldots, x_n\rangle$ if and only if $\Pi = \{[i]\}$ for some $1 \leq i \leq n$.

In other words, any single element of the input vector explains the result; the case of subtraction is defined identically. Multiplication, however, has a different definition. This is caused by the fact that 0 is an absorbing element, hence its presence suffices for a product to yield zero, as is explained by the following theorem.

Theorem 2. Let $f : \mathbb{V}\langle\mathbb{R}^*\rangle \to \mathbb{R}$ be the function defined as $\langle x_1, \ldots, x_n\rangle \mapsto x_1 \cdots x_n$. For a given input $\langle x_1, \ldots, x_n\rangle$, $\mathit{\Pi}$ explains $\mathbf{1}$ if and only if:

- $\mathit{\Pi} = \{[i]\}$ for some $1 \leq i \leq n$, if for all $1 \leq j \leq n$, $x_j \neq 0$
- $\mathit{\Pi} = \bigcup_{i \in \{j : 1 \leq j \leq n \text{ and } x_j = 0\}} \{[i]\}$ otherwise.

Proof. Suppose that all elements of the input vector are non-null. Let $\mathit{\Pi} = \{[i]\}$ for some $1 \leq i \leq n$. Clearly, any input that differs from $\langle x_1, \ldots, x_n\rangle$ only in the i-th element produces a different product, and hence is a minimal explanation. Suppose now that at least one element of the vector is null; in such a case, the function returns 0. Let $S = \{j : 1 \leq j \leq n \text{ and } x_j = 0\}$ be the set of all such vector indices. A vector must differ from the input in at least all these positions in order to produce a different output, otherwise the function still returns zero. The only refinements of this set are its strict subsets, but none is sufficient to change the output, and hence the defined set is the only minimal set satisfying the explanation property. □

The same argument can be made of the Boolean function that computes the conjunction of a vector of Boolean values. If all elements of the vector are $\top$, changing any of them produces a change of value, and hence each $\{[i]\}$ explains the output. Otherwise, the set $\mathit{\Pi} = \bigcup_{i \in \{j : 1 \leq j \leq n \text{ and } x_j = \bot\}} \{[i]\}$ is the only minimal set of input parts that explains the output, by a reasoning similar to the case of multiplication above. A dual argument can be done for disjunction by swapping the roles of $\top$ and $\bot$.

The case of the remaining usual arithmetic and Boolean functions can be dispatched easily. Functions taking a single argument (abs, etc.) obviously have this single argument as their only minimal explanation part.

We finally turn to the case of a function that extracts the k-th element of a vector. We recall that vectors can be nested, and hence this element may itself be composite. The intuition here is that what explains a part of the output is that same part in the k-th element of the input.

Theorem 3. Let $f_k : \mathbb{V}\langle\mathcal{O}^n\rangle \to \mathcal{O}$ be the function defined as $\langle x_1, \ldots, x_n\rangle \mapsto x_k$ if $1 \leq k \leq n$, and $\langle x_1, \ldots, x_n\rangle \mapsto \boxslash$ otherwise. Let $\pi \in \mathit{\Pi}_{\mathcal{O}}$ be an arbitrary part of $\mathcal{O}$. For any input $\langle x_1, \ldots, x_n\rangle$, $\mathit{\Pi}$ explains π for $\langle x_1, \ldots, x_n\rangle$ if and only if $\mathit{\Pi} = \{\pi \circ [k]\}$ and $1 \leq k \leq n$.

Proof. If $k < 0$ or $k > n$, f_k produces $\boxslash$ regardless of its argument, and so no set of input explains the result. Otherwise, it suffices to observe that $(\pi \circ [i])(\langle x_1, \ldots, x_n\rangle) = \pi(x_i) = \pi(f_k(\langle x_1, \ldots, x_n\rangle))$, and hence $\{\pi \circ [i]\}$ explains the output. No other set satisfies this condition. □

$$
\begin{aligned}
[i](\langle o_1 \dots o_n \rangle) &= \begin{cases} \langle o_i \rangle & \text{if } 1 \leq i \leq n \\ \boxslash & \text{otherwise} \end{cases} \\
\alpha_f(\langle o_1, \dots, o_n \rangle) &= \langle f(o_1), \dots, f(o_n) \rangle \\
\omega_f^{k,o}(\langle o_1, \dots, o_n \rangle) &= \langle f(\langle o_1, \dots, o_k \rangle), \dots, f(\langle o_{n-k}, \dots, o_n \rangle) \rangle \\
¿(\langle b, o, o' \rangle) &= \begin{cases} o & \text{if } b = \top \\ o' & \text{otherwise} \end{cases}
\end{aligned}
$$

Table 1: Definition of elementary vector functions studied in this paper.

4.3 Explanation for Vector Functions

We shall delve into more detail on basic functions that produce a value that may be of non-scalar type, summarized in Table 1 (function $[i]$ has already been discussed earlier). Here, we must consider the fact that an explanation may refer to a part of an element of their output, i.e. designators of the form $\pi \circ [i]$, with π some arbitrary designator.

The first function, noted α_f, applies a function f on each element of an input vector, resulting in an output vector of same cardinality.

Theorem 4. On a given input $\langle x_1, \dots, x_n \rangle$, Π explains $\pi \circ [i]$ of α_f if and only if $\Pi = \{\Pi' \circ [i]\}$ for some set of parts Π', and Π' explains π for x_i of f.

Proof. The i-th element of the output of α_f is $f(x_i)$; if Π' explains x_i of f, then $\Pi' \circ [i]$ explains $\langle x_1, \dots, x_n \rangle$ of α_f. Conversely, suppose that $\Pi' \circ [i]$ explains $\langle x_1, \dots, x_n \rangle$ of α_f; by definition, there exists another input $\langle x'_1, \dots, x'_n \rangle$ that differs on $\Pi' \circ [i]$ and such that the output of α_f differs on $\pi \circ [i]$. Then x_i and x'_i differ on Π', and by the definition of α_f, $f(x_i)$ and $f(x'_i)$ differ on π. If Π' admits a proper part that satisfies this property, then Π is not an explanation, which contradicts the hypothesis. Hence Π' is minimal, and it therefore explains π for x_i of f. □

Function $\omega_f^{k,o}$ applies a function f on a sliding window of width k to the input vector. That is, the first element of the output vector is the result of evaluating f on the first k elements; the second element is the evaluation of f on elements at positions 2 to $k+1$, and so on. If the input vector has fewer than k elements, the function is defined to return a predefined value o. To establish the set of minimal explanations, we define a special function σ_i; given a set of parts Π such that all parts are of the form $\pi \circ [j]$, replaces each of them by $\pi \circ [j - i]$. In other words, parts pointing to a part π of the j-th element of a vector end up pointing to the same part π of the $(j - i)$-th element of a vector.

Theorem 5. On a given input $\langle x_1, \dots, x_n \rangle$, Π explains $\pi \circ [i]$ of $\omega_f^{k,o}$ if and only if $n \geq k$, $i \geq n - k$, $\Pi = \{\Pi' \circ [i]\}$ for some set of parts Π'. and $\sigma_i(\Pi')$ explains π for x_i of f.

Proof. The proof is almost identical as for α_f, with the added twist that in the i-th window, an explanation for f referring to a part of the j-th element of its input vector actually refers to the $(j-i)$-th element of the input vector given to ω_f; this explains the presence of σ_i. We omit the details. □

Finally, function "¿" acts as a form of if-then-else construct: depending on the value of its (Boolean) first argument, it returns either its second or its third argument (which can be arbitrary objects). Defining its explanation requires a few cases, depending on whether the second and third element of the input are equal.

Theorem 6. On a given input $\langle x_1, x_2, x_3\rangle$, if $x_2 \neq x_3$, then $\{[1]\}$ always explains π of ¿; moreover $\{\pi \circ [2]\}$ explains π of ¿ if $x_1 = \top$, and $\{\pi \circ [3]\}$ explains π of ¿ if $x_1 = \bot$. However, if $x_2 = x_3$, then: 1. if $x_1 = \top$, Π explains π of ¿ if and only if $\Pi = \{\pi \circ [2]\}$ or $\Pi = \{[1], \pi \circ [3]\}$; 2. if $x_1 = \bot$, Π explains π of ¿ if and only if $\Pi = \{\pi \circ [3]\}$ or $\Pi = \{[1], \pi \circ [2]\}$.

Proof. Direct from the fact that ¿$(\langle \top, x_2, x_3\rangle) = x_2$, and ¿$(\langle \bot, x_2, x_3\rangle) = x_3$. The only corner case is when $x_2 = x_3$; if $x_1 = \top$, one must change either x_2, or *both* x_1 and x_3 in order to produce a different result (and dually when $x_1 = \bot$). □

4.4 Explanation for Composed Functions

Defining and proving conservative generalizations is a task that can quickly become tedious for complex functions, as the previous examples have shown us. Moreover, this process must be done from scratch for each new function one wishes to consider, as the proofs for each of them are quite different. In this section, we consider the situation where a complex function f is built through the composition of simpler functions.

We first demonstrate a recipe for building conservative generalizations for compositions of functions. In such a case, it is possible to derive a conservative generalization for f by chaining and combining the generalizations already obtained for the simpler functions it is made of. To ease notation, we shall write $\Pi \Vdash_o \pi$ to indicate that Π is a conservative generalization of all minimal input parts that explain π for input o. We extend the notion of generalization to sets of output parts; for a set of parts Π', we have that $\Pi \Vdash_o \Pi'$ if $\Pi \Vdash_o \pi'$ for every $\pi' \in \Pi'$. We first trivially observe the following:

Theorem 7. For a given function $f : \mathcal{O} \to \mathcal{O}'$ and a given input $o \in \mathcal{O}$, if $\Pi_1 \Vdash_o \pi_1$ and $\Pi_2 \Vdash_o \pi_2$, then $\Pi_1 \cup \Pi_2 \Vdash_o \{\pi_1, \pi_2\}$.

Thus, given a set of output parts Π', a conservative generalization can be obtained by taking the union of the generalizations for each individual part in Π'. We can then establish a result for the composition of two functions.

Theorem 8. Let π be an output part of some function f, and a given input $o \in \mathcal{O}$. Let Π_f be a set of sets of parts such that $\Pi_f \Vdash_o \pi$ for function f. Let Π_g be a set of sets of parts such that $\Pi_g \Vdash_{f(o)} \Pi_f$ for some function g. Then $\Pi_g \Vdash_o \pi$ for function $f \circ g$.

Proof. Suppose that there is a set of parts Π that does not intersect with Π_g, and such that for two inputs o and o' that differ on π_g, $\pi(f \circ g)(o) \neq \pi(f \circ g)(o')$. Let $x = g(o)$ and $y = g(o')$; since $\pi(f(x)) \neq \pi(f(y))$, then x and y differ on a set of parts Π'. By definition, a refinement of Π', called Π'', is also a refinement of Π_f. Since $\Pi_g \Vdash_{f(o)} \Pi_f$, this implies that o and o' differ on a part that intersects with Π_g, which contradicts the hypothesis. It follows that all sets of parts of $f \circ g$ that explain o for π intersect with Π_g, and hence $\Pi_g \Vdash_{f(o)} \pi$ for function $f \circ g$. □

Thanks to this result, one can obtain a conservative generalization for $f \circ g$ by first finding a conservative generalization Π of π for g, and then finding a conservative generalization of Π' of Π for f. This spares us from defining an input-output explanation relation for each possible function, at the price of obtaining a conservative approximation of the actual relation. When expressing these explanations as and-or trees, this simply amounts to appending the root of an explanation (the output of a function) to the leaf designating the corresponding input of the function it is composed with.

We recall that one of the claimed features of our proposed approach was tractability. The theorems stated throughout this section give credence to this claim. One can see that, for each of the elementary functions we studied in Sections 4.2 and 4.3, determining the sets of input parts that are (conservative) explanations of an output part can be done by applying simple rules that require no particular calculation. Then, building an explanation for a composed function is not much harder, and requires properly matching the output parts of a function to the input parts of the one it calls.

5 Implementation and Experiments

Combined, the previous results make it possible to systematically construct explanations for a wide range of computations. It suffices to observe that nested lists-of-lists, coupled with the functions defined in Section 4, represent a significant fragment of a functional programming language such as Lisp.

To illustrate this point, the concepts introduced above have been concretely implemented into *Petit Poucet*[3], an open source Java library.[4] The library allows users to create complex functions by composing the elementary functions studied earlier, to evaluate these functions on inputs, and to generate the corresponding explanation graphs. This library is meant as a proof of concept that serves two goals: 1. show the feasibility of our proposed theoretical framework and provide initial results on its running time and memory consumption; 2. provide a test bench allowing us to study the explanation graphs of various functions for various inputs.

[3] In English *Hop-o'-My-Thumb*, a fairy tale by French writer Charles Perrault where the main character uses stones to mark a trail that enables him to successfully lead his lost brothers back home.

[4] `https://github.com/liflab/petitpoucet`. Version 1.0 is considered in this paper.

5.1 Library Overview

Petit Poucet provides a set of ready-made `Function` objects; in its current implementation, it contains all the functions defined in Section 4, in addition to a few others for number comparison, type conversion, descriptive statistics (i.e. average), basic I/O (reading and writing to files) and string and list manipulation. Composed functions are created by adding elementary functions into an object called `CircuitFunction`, and manually connecting the output of each function instance to the input argument of another.

Figure 5 shows a graphical representation of a complex function that can be created by instantiating and composing elementary functions of the library. Each white box represents an elementary function; composition is illustrated by lines connecting the output (dark square) of a function to the input (light square) of another. For functions taking other functions as parameters, namely α_f and ω_f, the parameterized function is represented by a rectangle attached with a dotted line (such as box A attached to box 2). The composed function shown here is exactly the one from our example in the introduction: from a CSV file (box 1), the second element of each line is extracted and cast to a number (boxes 2 and A), the average over a sliding window is taken (boxes 3 and B), each value is checked to be greater than 3 (boxes 4 and C), and the logical conjunction of all these values is taken (box 5).

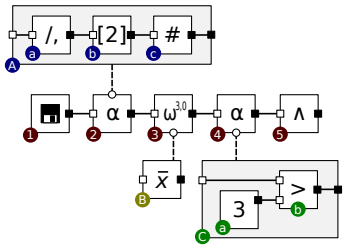

Fig. 5: Evaluating a condition on the average of values over a sliding window.

Once created, a function can be evaluated with input arguments. When this happens, it returns a special object called a `Queryable`. The purpose of the queryable is to retain the information about the function's evaluation necessary to answer "queries" about it at a later time. Each evaluation of the function produces a distinct queryable object with its own memory. Given a designator pointing to a part of the function's output, calling a `Queryable`'s method `query` produces the corresponding explanation and-or tree. Typically, one is interested in a simplified rendition displaying only the root, leaves and Boolean nodes, hiding the intermediate nodes made of the input/output of the intermediate functions in the explanation. On the CSV file shown in the introduction, the library produces the tree that is shown in Figure 1.[5]

[5] Or more precisely a directed acyclic graph, since leaf nodes with the same designator are not repeated.

One can see that the false result produced by the function admits three alternate explanations (the three sub-trees under the "or" node). The first explanation involves the numerical values in lines 3-4-5 (children of the first "and" node), the second includes lines 4-5-6, and the third explanation is made of the numbers in lines 5-6-7. This corresponds exactly to the three windows of successive value whose average is not greater than 3. Indeed, the presence of either of these three windows, and nothing else, suffices for our global condition to evaluate to false. Note how the explanation generation mechanism correctly and automatically identifies these, and also how, thanks to our concept of designator, the explanation can refer to specific locations inside specific lines of the input object.

In Petit Poucet, lineage capabilities are *built-in*. The user is not required to perform any special task in order to keep track of provenance information. In addition, it shall be noted that one does not need to declare in advance what designation graph will be asked for. The construction of the `Queryable` objects is the same, regardless of the output part being used as the starting point. Finally, the library follows a modular architecture where the set of available functions can easily be extended by creating packages defining new descendants of `Function`. It suffices for each function to produce a `Queryable` object that computes its specific input-output relationships; an explanation can then be computed for any composed function that uses it.

5.2 Experiments

To test the performance of the library, we selected various data processing tasks and implemented them as composed functions in Petit Poucet.

Get All Numbers Represents a simple operation that takes an input comma-separated list of elements, and produces a vector containing only the elements of the file that are numerical values. The explanation we ask is to point at a given element of the output vector, and retrace the location in the input string that corresponds to this number.

Sliding Window Average Given a CSV file, this task extracts the numerical value in each line, compute the average of each set of n successive values and check that it is below some threshold t (similar to the example we discussed earlier). Computing an aggregation over a sliding window is a common task in the field of event stream processing [26] and runtime verification [4], and is also provided by most statistical software, such as R's `smooth` package. It can be seen as a basic form of trend deviation detection [41], where the end result of the calculation is an "alarm" indicating that the expected trend has not been followed across the whole data file; a classical example of this is the detection of temperature peaks in a server rack [40]. The explanation we ask is to point at the output Boolean value (true or false), and retrace the locations in the file corresponding to the numbers explaining the result.

Triangle Areas Given a list of arbitrary vectors, this task checks that each vector contains the lengths of the three sides of a valid triangle; if so, it computes their area using Heron's formula[6] and sums the area of all valid triangles. It was chosen because it involves multiple if-then-else cases to verify the sides of a triangle. It also involves a slightly more involved arithmetical calculation to get the area, which is completely implemented by composing basic arithmetic operators in a composed function. The whole function is the composition of 59 elementary functions.

This example is notable, because the explanations it generates may take different forms. An explanation for the output value (total area) includes an explanation for each vector: if it represents a valid triangle, it refers to its three sides; if it does not, the condition can fail for different reasons: the vector may not have three elements, or have one of its elements that is not a number, or contain a negative value, or violate the triangle inequality. Each condition, when violated, produces different explanations pointing at different elements of the vector, or the vector as a whole. Moreover, each vector in the list may not be a valid triangle for different reasons, and hence a different explanation will be built for each of them. For example, given the list $[\langle a, 4, 2\rangle, \langle 3, 5, 6\rangle, \langle 2, 3\rangle]$, a tree will be produced that describes an explanation involving three elements: the first points at the element a of the first vector (not a number), the second points at all three components of the second vector (valid triangle), and the last points at the whole third vector (wrong number of elements).

Nested Bounding Boxes Given a DOM tree[7], this task checks that each element has a bounding box (width and height) larger than all of its children. This condition is the symptom of a layout bug which shows visually as an element protruding from its parent box inside the web browser's window. This corresponds to one of the properties that is evaluated by web testing tools such as Cornipickle [24] and ReDeCheck [49] on real web pages. In this task, trees are represented as nested lists-of-lists, with each DOM node corresponding to a triplet made of its width, height, and a list of its children nodes. The explanation we ask is to point at the Boolean output of the condition, and retrace the nodes of the tree that violate it.

In all these tasks, the inputs given to the function are randomly generated structures of the corresponding type. The experiments were implemented using the LabPal testing framework [25], which makes it possible to bundle all the necessary code, libraries and input data within a single self-contained executable file, such that anyone can download and independently reproduce the experiments. A downloadable lab instance containing all the experiments of this paper can be obtained online [27]. Overall, our empirical measurements involve 56 individual experiments, which together generated 224 distinct data points. All the experiments were run on a AMD Athlon II X4 640 1.8 GHz running Ubuntu 18.04, inside a Java 8 virtual machine with 3566 MB of memory.

[6] $A = \sqrt{s(s-a)(s-b)(s-c)}$, where $s = \frac{a+b+c}{2}$.
[7] A DOM tree represents the structure of elements in an HTML document [2].

Memory Consumption The first experiment aims to measure the amount of memory used up by the `Queryable` objects generated by the evaluation of a function, and the impact of the size of the input on global memory consumption. To this end, we ran various functions on inputs of different size; for each, we measured the amount of memory consumed, with explainability successively turned on and off. This is possible thanks to a switch provided by Petit Poucet, and which allows users to completely disable tracking if desired.

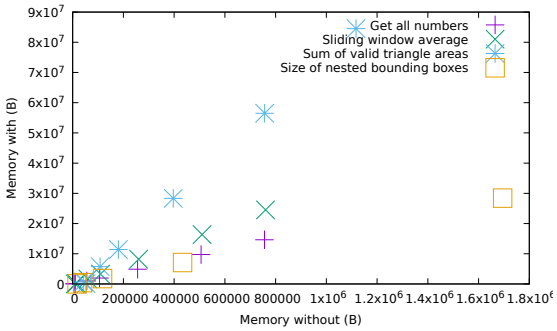

Fig. 6: Impact of explainability on memory consumption.

Figure 6 shows a plot that compares the amount of memory consumed by `Function` objects. Each point in the plot corresponds to a pair of experiments: the x coordinate corresponds to the memory consumed by a function without explainability, and the y coordinate corresponds to the memory consumed by the same function on the same input, but with explainability enabled. All the points for the same task have been grouped into a category and are given the same color.

Analyzing this plot brings both bad news and good news. The "bad" news is that the additional memory required for explainability is high when expressed in relative terms. For example, a composed that requires 498 kB to be evaluated on an input requires close to 15 MB once explainability is enabled. The "good" news is twofold. First, this consumption is still reasonable in the absolute: at this rate, it takes an input file of 42 million lines before filling up the available RAM in a 64 GB machine. Second, and most importantly, the relationship between memory consumption with and without lineage is linear: for all the tasks we tested, if m is the memory used without lineage, then the memory m' used when lineage is enabled is in $O(m)$, i.e. the ratio m'/m does not depend on the size of the input.

These figures should be put in context by comparing the overhead incurred by other systems mentioned in Section 2. Related systems for provenance in databases (namely Polygen [51], Mondrian [20], MXQL [48], DBNotes [11] pSQL [7] and Orchestra [29]) do not divulge their storage overhead for provenance data. A recent technical report on a provenance-aware database management system measures an overhead ranging between 19% and 702% [3]. Dynamic taint propagation systems report a memory overhead reaching 4× for TaintCheck [37],

240× [14] for Dytan, and "an enormity" of logging information for RIFLE ([13], authors' quote). Although these systems operate at a different level of abstraction, this shows that explainability is inherently costly regardless of the approach chosen.

Computation Time We performed the same experiments, but this time measuring computation time. The results are shown in Figure 7; similar to memory consumption, they compare the running time of the same function on the same input, both with and without explanation tracking. The largest slowdown observed across all instances is 6.7×. For a task like *Sliding Window Average*, the average slowdown observed is 1.93× across all inputs. Although this slowdown is non-negligible, it is reasonable nevertheless, adding at most a few hundreds of milliseconds on the problems considered in our benchmark.

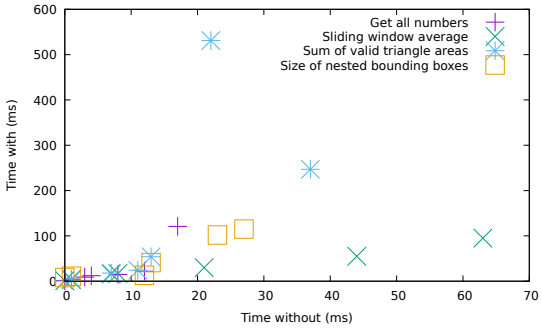

Fig. 7: Impact of explainability on computation time.

Again, these results should be put in context with respect to existing works that include a form of lineage. The MONDRIAN system reports an average slowdown of 3×; pSQL ranges between 10× and 1,000×; the remaining tools do not report CPU overhead. For taint analysis tools, Dytan reports a 30–50× slowdown; GIFT-compiled programs are slowed down by up to 12×; TaintCheck has a slowdown of around 20×, 1–2× for RIFLE and around 20× for TaintCheck. Of course, these various systems compute different types of lineage information, but these figures give an outlook of the order of magnitude one should expect from such systems.

6 Conclusion

This paper provided the formal foundations for a generic and granular explainability framework. An important highlight of this model is its capability to handle abstract composite data structures, including character strings or lists of elements. The paper then defined the notion of designator, which are functions that can point to and extract *parts* of these data structures. An explainability relationship

on functions has been formally defined, and conservative approximations of this relation have been proved for a set of elementary functions. A point in favor of this approach is that explanations of composed functions can be built by composing the explanations for elementary functions. Combined, these concepts make it possible to automatically extract the explanation of a result for generic functions at a fine level of granularity. These concepts have been implemented into a proof-of-concept, yet fully functional library called *Petit Poucet*, and evaluated experimentally on a number of data processing tasks. These experiments revealed that the amount of memory required to track explainability metadata is relatively high, but more importantly, showed that it is *linear* in the size of the memory required to evaluate the function in the first place.

Obviously, Petit Poucet is not intended to replace programs written using other languages and following different paradigms. However, it could be used as a library by other tools that could benefit from its explanation features. In particular, testing libraries such as JUnit could be extended by assertions written as Petit Poucet functions, and provide a detailed explanation of a test failure without requiring extra code. Explainability functionalities could also easily be retrofitted into existing (Java) software, with minimal interference on their current code. Case in point, we already identified the Cornipickle web testing tool [24] and the BeepBeep event stream processing engine [26] as some of the first targets for the addition of explainability based on Petit Poucet. A lineage-aware version of the GRAL plotting library[8] is also considered.

The existence of a definition of fine-grained explainability opens the way to multiple exciting theoretical questions. For example: for a given function, is there a part of the input that is present in all explanations? We can see an example of this in Figure 1, with the leaf pointing to value -80. Intuitively, this tends to indicate that some parts of an input have a greater "responsibility" than others in the result, and could provide an alternate way of quantifying this notion than what has been studied so far [12]. On the contrary, is there a part of the input that never explains the production of the output, regardless of the input? This latter question could shed a different light on an existing notion called *vacuity* [6], expressed not in terms of elements of the specification, but on the parts of the input it is evaluated on. More generally, explainability can be viewed as a particular form of static analysis for functions; it would therefore be interesting to recast our model in the abstract interpretation framework [35, 39] in order to further assess its strengths and weaknesses.

Finally, explanations could also prove useful from a testing and verification standpoint. The explanation graph could be used for log trace and bug triaging [30]: if two execution traces violate the same condition, one could keep one trace instance for each distinct explanation they induce, as representatives of traces that fail for different reasons. This could help reduce the amount of log data that needs to be preserved, by keeping only one log instance of each type of failure.

[8] `https://github.com/eseifert/gral`

References

1. G. Aleksandrowicz, H. Chockler, J. Y. Halpern, and A. Ivrii. The computational complexity of structure-based causality. *J. Artif. Intell. Res.*, 58:431–451, 2017.
2. V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. L. Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document object model (DOM) level 1 specification. Technical report, World Wide Web Consortium, 1998. `https://www.w3.org/DOM/`, Accessed November 17th, 2019.
3. B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Formal foundations of reenactment and transaction provenance. Technical Report IIT/CS-DB-2016-01, Illinois Institute of Technology, 2016.
4. D. A. Basin, F. Klaedtke, S. Marinovic, and E. Zalinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods Syst. Des.*, 46(3):262–285, 2015.
5. I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. J. Trefler. Explaining counterexamples using causality. *Formal Methods Syst. Des.*, 40(1):20–40, 2012.
6. S. Ben-David, F. Copty, D. Fisman, and S. Ruah. Vacuity in practice: temporal antecedent failure. *Formal Methods Syst. Des.*, 46(1):81–104, 2015.
7. D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *VLDB J.*, 14(4):373–396, 2005.
8. P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In J. V. den Bussche and V. Vianu, editors, *Proc. ICDT 2001*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2001.
9. J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2007.
10. L. Chiticariu and W. C. Tan. Debugging schema mappings with routes. In U. Dayal, K. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y. Kim, editors, *Proc. VLDB 2006*, pages 79–90. ACM, 2006.
11. L. Chiticariu, W. C. Tan, and G. Vijayvargiya. DBNotes: a post-it system for relational databases based on provenance. In F. Özcan, editor, *Proc. SIGMOD 2005*, pages 942–944. ACM, 2005.
12. H. Chockler and J. Y. Halpern. Responsibility and blame: A structural-model approach. *J. Artif. Intell. Res.*, 22:93–115, 2004.
13. J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In M. Blaze, editor, *Proc. USENIX Security 2004*, pages 321–336. USENIX, 2004.
14. J. A. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In D. S. Rosenblum and S. G. Elbaum, editors, *Proc. ISSTA 2007*, pages 196–206. ACM, 2007.
15. H. Cleve and A. Zeller. Locating causes of program failures. In G. Roman, W. G. Griswold, and B. Nuseibeh, editors, *Proc. ICSE 2005*, pages 342–351. ACM, 2005.
16. J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proc. MICRO-37*, pages 221–232. IEEE Computer Society, 2004.
17. Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.
18. T. Eiter and T. Lukasiewicz. Complexity results for structure-based causality. *Artificial Intelligence*, 142(1):53–89, 2002.
19. T. Ferrère, O. Maler, and D. Nickovic. Trace diagnostics using temporal implicants. In B. Finkbeiner, G. Pu, and L. Zhang, editors, *Proc. ATVA 2015*, volume 9364 of *Lecture Notes in Computer Science*, pages 241–258. Springer, 2015.

20. F. Geerts, A. Kementsietsidis, and D. Milano. MONDRIAN: annotating and querying databases through colors and blocks. In L. Liu, A. Reuter, K. Whang, and J. Zhang, editors, *Proc. ICDE 2006*, page 82. IEEE Computer Society, 2006.
21. T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In L. Libkin, editor, *Proc. PODS 2007*, pages 31–40. ACM, 2007.
22. A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *STTT*, 8(3):229–247, 2006.
23. S. Hallé. Causality in message-based contract violations: A temporal logic "who-dunit". In *Proc. EDOC 2011*, pages 171–180. IEEE Computer Society, 2011.
24. S. Hallé, N. Bergeron, F. Guérin, G. Le Breton, and O. Beroual. Declarative layout constraints for testing web applications. *J. Log. Algebr. Meth. Program.*, 85(5):737–758, 2016.
25. S. Hallé, R. Khoury, and M. Awesso. Streamlining the inclusion of computer experiments in a research paper. *IEEE Computer*, 51(11):78–89, 2018.
26. S. Hallé. *Event Stream Processing With BeepBeep 3: Log Crunching and Analysis Made Easy*. Presses de l'Université du Québec, 2018.
27. S. Hallé and H. Tremblay. Measuring the impact of lineage tracking in the Petit Poucet library (version v1.0), April 2021.
28. J. Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach. part i: Causes. *British J. for Philosophy of Sci.*, 56(4):843–887, 2005.
29. G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In A. K. Elmagarmid and D. Agrawal, editors, *Proc. SIGMOD 2010*, pages 951–962. ACM, 2010.
30. R. Khoury, S. Gaboury, and S. Hallé. Three views of log trace triaging. In F. Cuppens, L. Wang, N. Cuppens-Boulahia, N. Tawbi, and J. García-Alfaro, editors, *Proc. FPS 2016*, volume 10128 of *Lecture Notes in Computer Science*, pages 179–195. Springer, 2016.
31. O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods Syst. Des.*, 19(3):291–314, 2001.
32. L. Lam and T.-c. Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. In *Proc. ACSAC 2006*, pages 463–472, Miami Beach, FL, USA, Dec. 2006. IEEE.
33. T. Leek, R. Brown, M. Zhivich, and R. Lippmann. Coverage maximization using dynamic taint tracing. Technical Report 1112, Massachusetts Institute of Technology, 2007.
34. S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In R. Gupta and S. P. Amarasinghe, editors, *Proc. PLDI 2008*, pages 193–205. ACM, 2008.
35. A. Møller and M. I. Schwartzbach. Static program analysis, October 2018. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.
36. S. Mukherjee and P. Dasgupta. Computing minimal debugging windows in failure traces of AMS assertions. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 31(11):1776–1781, 2012.
37. J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. NDSS 2005*. The Internet Society, 2005.
38. B. Pérez, J. Rubio, and C. Sáenz-Adán. A systematic review of provenance systems. *Knowl. Inf. Syst.*, 57(3):495–543, 2018.
39. X. Rival and K. Yi. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. MIT Press, 2020.

40. T. Rohrmann. Introducing complex event processing (CEP) with Apache Flink, 2016. `https://flink.apache.org/news/2016/04/06/cep-monitoring.html`, Accessed November 17th, 2019.
41. M. Roudjane, D. Rebaïne, R. Khoury, and S. Hallé. Detecting trend deviations with generic stream processing patterns. *Information Systems*, (101446):1–24, 2019. To appear, DOI: `10.1016/j.is.2019.101446`.
42. W. Samek, T. Wiegand, and K.-R. Müller. Explainable Artificial Intelligence: Understanding, Visualizing and Interpreting Deep Learning Models. *ITU Journal*, (1), Aug. 2017. arXiv: 1708.08296.
43. V. Schuppan and A. Biere. Shortest counterexamples for symbolic model checking of LTL with past. In N. Halbwachs and L. D. Zuck, editors, *Proc. TACAS 2005*, volume 3440 of *Lecture Notes in Computer Science*, pages 493–509. Springer, 2005.
44. G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In S. Mukherjee and K. S. McKinley, editors, *Proc. ASPLOS 2004*, pages 85–96. ACM, 2004.
45. N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: an architectural framework for user-centric information-flow security. In *Proc. MICRO-37*, pages 243–254. IEEE Computer Society, 2004.
46. S. J. van Zelst, A. Bolt, M. Hassani, B. F. van Dongen, and W. M. P. van der Aalst. Online conformance checking: relating event streams to process models using prefix-alignments. *Int. J. Data Sci. Anal.*, 8(3):269–284, 2019.
47. S. Vandebogart, P. Efstathopoulos, E. Kohler, M. N. Krohn, C. Frey, D. Ziegler, M. F. Kaashoek, R. T. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4):11, 2007.
48. Y. Velegrakis, R. J. Miller, and J. Mylopoulos. Representing and querying data transformations. In K. Aberer, M. J. Franklin, and S. Nishio, editors, *Proc. ICDE 2005*, pages 81–92. IEEE Computer Society, 2005.
49. T. A. Walsh, P. McMinn, and G. M. Kapfhammer. Automatic detection of potential layout faults following changes to responsive web pages. In *Proc. ASE 2015*, page 709–714. ACM, 2015.
50. C. Wang, Z. Yang, F. Ivancic, and A. Gupta. Whodunit? causal analysis for counterexamples. In S. Graf and W. Zhang, editors, *Proc. ATVA 2006*, volume 4218 of *Lecture Notes in Computer Science*, pages 82–95. Springer, 2006.
51. Y. R. Wang and S. E. Madnick. A Polygen model for heterogeneous database systems: The source tagging perspective. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proc. VLDB 1990*, pages 519–538. Morgan Kaufmann, 1990.