A Generic Explainability Framework for Function Circuits

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

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

Abstract

This paper presents the theoretical foundations of a data lineage framework for a model of computation called a *function circuit*, which is a directed graph of elementary calculation units called "functions". The proposed framework allow functions to manipulate arbitrary data structures, and allows lineage relationships to be expressed at a fine level of granularity over these structures by introducing the abstract concept of *designator*. Moreover, the lineage graphs produced by this framework allow multiple alternate explanations of a function's outputs from its inputs. The theoretical groundwork presented in this paper opens the way to the implementation of a generic lineage library for function circuits. *(We limited the paper to 5 pages as to allow a future full-length publication.)*

1 Introduction

Developers of information systems in all disciplines are facing increasing pressure to come up with mechanisms to describe how a specific result is produced –a concept called *explainability*. Although the term is often tied to Artificial Intelligence [13], explainability is desirable in other fields of computation. For example, 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 [15]. Similarly, 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 [7].

In a broad sense, explainability can be seen as a way of linking the outputs of a process to its inputs; this notion is closely related to the concepts of *lineage* or *provenance*. A recent survey [12] reveals the existence of more than two dozen provenance-aware systems, including DBNotes [2], Karma [9], LipStick [1], MONDRIAN [4], SPADE [5], Tioga [14], ORCHESTRA [10], RAMP [11] and Wings [6], among many others. However, the explainability use cases mentioned above have distinctive traits that make these existing provenance frameworks only partially appropriate for the task.

First of all, the majority of provenance systems are focused on relational database queries, scientific workflows, or map-reduce pipelines. The operations involved in our explainability examples correspond to none of these computational models: bug finding in web applications relies on the evaluation of first-order logic expressions, while log analysis corresponds to finding a run in a finite-state machine. In addition, the data being manipulated in our scenarios does not correspond to structures typically used in provenance, which are heavily tuple-oriented: a web page is a *tree* of document nodes, while a log is an ordered *sequence* of event names and attributes.

Another element that must be considered is that explainability requires the expression of fine-grained dependencies over these data structures. A bug in a web page can be explained by pointing to specific *nodes* within a tree; a compliance violation in a log is caused by a *subsequence* of the whole list of events, and is perhaps due to specific values within the events of that subsequence. Finally, the notion of explainability itself requires a special treatment, since the result of a computation can, in some cases, admit multiple *alternate* explanations. That is, different parts of the input, combined in different ways, can each be sufficient to explain a given output; moreover, the presence of such alternate explanations may depend on the actual inputs and outputs, and hence cannot always be determined statically.

The previous observations motivate the development of a generic explainability framework, the core details of which are presented in this paper. In Section 2, we introduce a generic model of computation called a *function circuit*, where each function's inputs and outputs can be of arbitrary data types. In Section 3, we introduce the concept of *designator*, which is based on an abstract "part-of" relationship; this will be used to refer to arbitrary parts of generic objects. Then, in Section 4, we describe how, given a designator and a function, a structure called a *designation graph* can be constructed and link the output of the function to parts of its input. Provided that a *derivation operator* is defined for each function individually and follows a few intuitive properties, the construction of such graphs is then guaranteed for any function circuit.

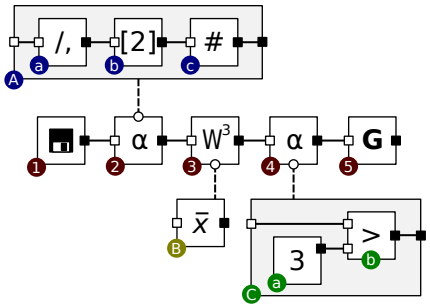

Figure 1: Graphical representation of a function circuit.

2 Function Circuits

Since the problem of lineage applies to an operation executed on some input data, a necessary first step is to define the computational model over which lineage is then to be tracked. This is the purpose of this section, where we briefly introduce a simple system called a *function circuit*. Formally, a *function circuit* is a digraph $C = \langle V, E \rangle$ where vertices represent functions and there can be an arc between the functions f and g if and only if some output of f is used as an input of g. Since functions can have input and output arity grater than 1, multiple arcs between two vertices are allowed. Intuitively, a function circuit is an acyclic graph whose vertices are "functions" –stateless black boxes receiving inputs and producing outputs. Consequently, it is not possible to deduce a function's internal definition by analyzing this graph, only its effect on the other components of the system.

One can see a circuit as a graph where vertices are functions with upstream and downstream "pins" (input and output values), and edges connect downstream pins to upstream pins. Figure 1 shows a graphical representation of a simple function circuit. The convention we shall use is to represent functions as square boxes; input pins are drawn as light squares on the side of the box, and output pins as dark squares. A line between two pins represents a (directed) connection. For functions with more than one input or output, pins are ordered from top to bottom.

For example, in Figure 1, the circuit first turns an arbitrary text file into a list of lines (box 1). For each line (box 2), it applies the circuit represented in box A. This circuit splits each line on commas to create an array (a), extracts the second element of the array (b) and converts it into a number (c). The resulting list is sent to a sliding window function (box 3), which computes the average (box B) of three successive elements. The result is a list of averages; on each element of that list (box 4), the circuit of box C is applied: it checks that each of these values is greater (b) than the constant 3 (a). Finally, function G (box 5) computes the logical conjunction of the resulting list of Booleans; in other words, it returns true only if all the elements of the list are true. The end result is a circuit that evaluates the "property" that the average of the second element of every three successive lines in the input file is greater than 3.

This computational model is reminiscent of the "data flows" presented by Woodruff *et al.* [16]. However, whereas data flows are tuple-oriented, our proposed computational model intends to be more generic, and accommodate arbitrary functions and data types. The data elements manipulated in this circuit include character strings, lists of strings, scalar numbers and Boolean values. Another distinctive element of function circuits is that some functions are parameterized by another circuit; this is represented by a box linked to them by a dashed line. For instance box 2 is a function that receives a list as its input, and evaluates another circuit on each of the elements of the list. Therefore, a single evaluation of box 2 results in multiple distinct "sub-evaluations" of box A, one for each list element. As we shall see in the following, we will also allow each sub-evaluation to have a distinct input/output explanation.

We assume circuits are well-formed: they do not contain cycles, and the domain of each of a function's input pin includes at least all of the image of the function's output pin to which it is connected. Function circuits differ from imperative programming by the absence of explicit variables and loops, and in this sense are a closer parent to the functional programming paradigm. While not the purpose of this paper, it is relatively easy to convince oneself that, with a suitably defined set of primitive functions, circuits can represent a wide range of computations over various types of data. For example, the computation pipelines of the BeepBeep event stream processing engine [8], composed of a graph of independent units called "processors", can be modeled as function circuits. Similarly, such a model can accommodate a variety of other functions, such as Boolean connectives, quantifiers, path selectors in a tree structure, and so on. We can hence consider it suitably generic to encompass the explainability use cases described in the beginning.

3 Designators

Most provenance works developed so far rely implicitly on the concept of "part of" to define provenance relationships. Take for example the definition from Cui *et al.* [3]: given an output tuple t resulting from some query q and input tables $T_1, \ldots, T_n$, the tuple derivation for q is the *smallest subset* of the T_i such that t *appears* in the output. Here, two different "part of" relationships are used: one to express the fact that a set of tuples is included in a set of tables, and another to express the fact that a tuple t is an element of the set of tuples produced as output. Such "part of" relationships are straightforward to define when the universe of discourse is made of tuples and sets of tuples, as is the case for database systems.

However, these notions are no longer appropriate or sufficient in a framework, such as our proposed function cir-

cuits, where the objects that are manipulated can be numerical scalars, lists, or character strings. Therefore, the second part of our lineage framework consists of defining generic topological relations between various data structures. To this end, we introduce a notation that can refer to a function's inputs and outputs, but also to parts of such inputs and outputs, for various types of objects.

Let $\mathfrak{U} = \bigcup_i O_i$ be a union of disjoint sets of *objects*. The sets O_i are called *types*. Among the possible types we consider are numbers ($\mathbb{R}$), characters ($\mathbb{S}$) and character strings ($\mathbb{S}^*$), lists of elements of type T ($\mathbb{L}\langle T\rangle$), and sets of elements of type T (2^T). Functions will also be considered as objects and are represented by the set $\mathbb{F}$. We suppose that $\mathfrak{U}$ contains a special object, noted $\boxslash$, that represents "nothing".

For any object $o \in \mathfrak{U}$, a part of o is a tuple (p,o) for some object $p \in \mathfrak{U}$. We assume this induces a partial order relation $\sqsubseteq$ on $\mathfrak{U}$. This relation can be arbitrary; however, we expect it to follow a few intuitive properties: 1. $\boxslash$ is the unique object that is a part of every object; 2. the only parts of a number or character $o \in \mathbb{R} \cup \mathbb{S}$ are $\boxslash$ and o itself; 3. a part of a set $o \in 2^T$ is either a part of one of its elements, or a subset composed of parts of its elements; 4. a part of a list $o \in \mathbb{L}\langle T\rangle$ is either a part of one of its elements, or a list made from parts of a contiguous sub-sequence of its elements; 5. a part of a function $f \in \mathbb{F}$ is a part of one of its input or output pins, or a set of such parts. As one can see, this definition generalizes the notion of set inclusion and element membership used in existing provenance definitions based on sets and tuples.

Once the notion of "object part" is formally defined, we introduce functions whose purpose is to extract parts from objects. An *atomic designator* is a function $d : O \to O'$, where O and O' are two types, such that $d(o) \sqsubseteq o$ for every $o \in O$. For example, the function $[m:n]$ that returns a range of characters in a string is an atomic designator; so is the function $[n]$ that returns the n-th element in a list. We also introduce two special designators noted $\uparrow_i$ and $\downarrow^i$. The first represents the i-th input pin of a function, while the second represents its i-th output pin. The set $\mathfrak{D}$ represents all atomic designators.

Designators can be composed in the traditional sense of the term, i.e. for two designators d and d', $(d \circ d')(x) \triangleq d(d'(x))$. A designator that is the composition of atomic designators is called a *compound* designator. Any designator (atomic or compound) can be represented as a finite vector of atomic designators; intuitively, the list $\overline{d} = \langle d_1, \ldots, d_k\rangle$ represents the compound designator $d_1 \circ \cdots \circ d_k$. For this reason, we shall denote as $\mathfrak{D}^*$ (the set of all finite sequences of elements of $\mathfrak{D}$) the set of all designators. The *tail* of a compound designator corresponds to the last atomic designator of the vector.

Remark that the definition of designator is intentionally broad. It is indeed expected that the user defines his own suitable designators.

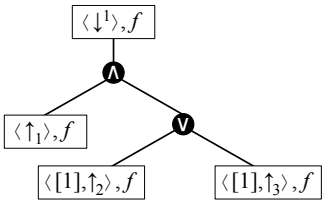

Figure 2: A simple designation graph.

4 Designation Graphs and Derivation

Designators can be arranged in a special type of graph structure, called a *designation graph*. Formally, a designation graph is a triplet $T = \langle V, \lambda, \delta\rangle$ where V is a set of vertices, $\delta \subseteq V^2$ is a connectivity relation that forms a directed acyclic graph (DAG), and $\lambda : V \to L$ is a labeling function, which assigns to each vertex in V some label. A vertex label can either be the special symbols $\wedge$ ("and"), $\vee$ ("or"), or a tuple $(\overline{d}, o) \in \mathfrak{D}^* \times \mathfrak{U}$. We shall denote by $\mathfrak{T}$ the set of all designation graphs.

Figure 2 shows an example of a simple designation graph. By convention, such graphs will be read top-down, which eliminates the need for arrows on directed edges. In this example, one can see that the designation $\langle\downarrow^1, f\rangle$ ("the first output pin of function f") is linked through an "and" node to two other designations: one refers to the first input pin of f, while the other is itself an "or" node. The left-hand child refers to the first element of the second input pin of f, while the second refers to the first element of the third input pin of f.

This graph structure addresses one of the issues mentioned in Section 1, namely the fact that explainability must account for the presence of multiple alternate explanations. In the example, the derivation graph could be interpreted as asserting that the (first) output value of f can be explained by the value of its first input argument, combined with *either* the first element of its second input argument, or the first element of its third input argument. For example, suppose f is defined as $f(x,y,z) = (x - y[1]) \cdot (x - z[1])$. Let $x = 3$, and y and z be two lists whose first element is 3. In such a case, f produces the value 0; however, since both operands of the multiplication are null, the value of x combined with either $y[1]$ or $z[1]$ suffices to produce the null value. Note how this designation graph depends on the actual values of x, y and z; this explanation is not valid for all possible inputs.

Given a function $f \in \mathbb{F}$ of input arity m and output arity n, a *derivation operator* is a function $\Delta_f : \mathfrak{D}^* \times \mathfrak{U}^m \times \mathfrak{U}^n \to \mathfrak{T}$. It takes as input a triplet $(\overline{d}, \overline{u}, \overline{u'})$ where $\overline{d}$ is a designator whose head is $\downarrow^i$ for $i \in [1,n]$, $\overline{u}$ is an m-vector containing the input values of f, and $\overline{u'}$ is an n-vector containing the output values of f. We require that $\Delta_f(\overline{d}, \overline{u}, \overline{u'})$ produces a designation graph $\langle V, \lambda, \delta\rangle \in \mathfrak{T}$ such that all leaves $v' \in V$ have a labeling of the form $(\overline{d'}, f)$, where the tail of $\overline{d'}$ is $\uparrow_i$ for some $i \in [1,m]$. In

other words, a part of the function's output is linked to parts of its inputs.

The advantage of modeling lineage in such a way is threefold. First, if Δ_f is defined for any triplet $(\overline{d}, \overline{u}, \overline{u'}) \in \mathfrak{D}^* \times \mathfrak{U}^m \times \mathfrak{U}^n$, a designation graph can easily be obtained for any function circuit. By construction, all the leaves of a designation graph have designators pointing to one of the function's input i. Since that function f is part of a circuit, this input is connected to another function g's output j; therefore, it suffices to replace f with g and $\downarrow^i$ with $\uparrow_j$, and evaluate the derivation operator again from this new node. The process eventually stops and produces a finite, global designation graph whose endpoints are the circuit's output at one end, and parts of the circuit's inputs at the other.

The second advantage is that, through the use of designators, one can ask for provenance relationships about specific *parts* of a function's output. For example, if f is a function that returns a list of elements, a user is not restricted to asking provenance for the whole list ($\downarrow^1$), but may, using a compound designator such as $[n] \circ \downarrow^1$, ask for the provenance relationship of the n-th element inside that list. Similarly, this relationship may point not only to the whole of a function's input, but to a *part* of that input.

The third advantage is that this framework can accommodate different definitions of Δ, and therefore different provenance relationships. The generic technique to produce designation graphs remains unchanged; the process differs only in the specific instance of derivation operator Δ that is being used to generate a graph.

5 Revisiting the Example

To illustrate this last point, let us revisit the function circuit given as example in Figure 1, and the input file shown in the left part of Figure 3. For the sake of the example, let us assume that the derivation operator Δ is defined such that in the default case, the output pin $\downarrow^i$ for a function f is associated to each of its input pins $\uparrow_j$ through an "and" node. Moreover, we informally define a particular behavior for Δ for a few functions:

- The "/," function links each element of the output array to the subsequence of the input string corresponding to that element.
- The $[n]$ function links its output to the n-th element of its input list.
- The G function computes the logical conjunction of all the Boolean values in its input list; if this conjunction is false, $\downarrow^1$ is linked through an "or" node to $[i] \circ \uparrow_i$, for all positions i in the input list that contain the value $\perp$.
- The α function applies another function f to each of the elements of an input list; hence the i-th element of the output list is linked to the designation graph that f produces on the i-th element of the input list.
- The application of the window function $\mathsf{W}^n(f)$ results in a list where the i-th output element is linked to the designation graph of f, when given the corresponding window of events from the input list.

Notice how, in the case of "/," and G, the derivation graph produced depends on the actual contents of the input list or string. Also note that, in the case of α and W, the derivation graph for a given element of the output list they produce relies on the derivation graph of the specific sub-evaluation of the underlying function f that produced this element.

Equipped with such a derivation operator, a designation graph can be constructed mechanically, yielding the structure shown in the right part of Figure 3. As one can easily see by examining the input file, the average of three successive values for the second element of each line is not always greater than 3, hence the global circuit returns the value false.

Examining such a graph can reveal interesting insights. The "or" node indicates the presence of three alternative explanations for this result, each involving three lines of the input file –or more precisely, three specific substrings of each input line. One can see that the file locations correspond to the numerical part of each line, and that only those lines involved in a faulty window (whose average is less than 3) are present. Some lines even seem to be "more important" than others by being involved in all possible explanations, as is the case for the leaf node designating line 5 (center); it turns out this line has a large negative value (-80) causing all the windows that contain it to fail the condition. This last observation would not be possible in a structure that does not keep multiple alternate explanations.

6 Conclusion

This paper provided the formal foundations for a lineage tracking framework, based on the concepts of function circuit, designator and derivation operator. Although many details and proofs were glossed over, we intend to significantly extend the present work with the necessary formal approach in a future publication. In addition to explainability, we intend to define existing definitions of provenance, such as *why*-provenance and *where*-provenance, using the concepts of designators provided by our framework. An implementation of these concepts in the form of an open source Java library is also under development; our prototype is already operational for a core set of functions and designators (including all those presented in this paper).[1]

Our model obviously presents some limitations. First, it is more suitable to function-oriented computations. Second, the task of defining designators falls completely to the user. Finally, the designation graph depends on the actual operator Δ_f used for each of the functions involved in the circuit. However, one can see that, given a suitable definition of Δ_f,

[1] https://github.com/liflab/petitpoucet

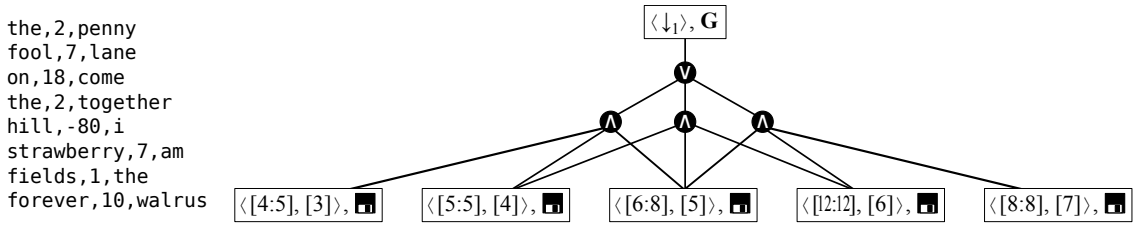

Figure 3: Left: a simple text file, used as the input function for the circuit of Figure 1. Right: a designation graph obtained for this input, using the derivation operator Δ used as an example in the paper. The graph has been compressed so as to only show the and/or relationships between the root and the leaves.

this framework can produce designation graphs matching our intuitive conception of "explanation". It is hoped that the further study of designation graphs will help shed a new light on computability theory.

References

[1] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting Lipstick on Pig: Enabling database-style workflow provenance. *PVLDB*, 5(4):346–357, 2011.

[2] 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.

[3] 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.

[4] 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.

[5] A. Gehani and D. Tariq. SPADE: support for provenance auditing in distributed environments. In P. Narasimhan and P. Triantafillou, editors, *Proc. Middleware 2012*, volume 7662 of *LNCS*, pages 101–120. Springer, 2012.

[6] Y. Gil, V. Ratnakar, J. Kim, P. A. González-Calero, P. T. Groth, J. Moody, and E. Deelman. Wings: Intelligent workflow-based design of computational experiments. *IEEE Intell. Syst.*, 26(1):62–72, 2011.

[7] S. Hallé, N. Bergeron, F. Guerin, G. L. Breton, and O. Beroual. Declarative layout constraints for testing web applications. *J. Log. Algebr. Meth. Program.*, 85(5):737–758, 2016.

[8] S. Hallé. *Event Stream Processing With BeepBeep 3: Log Crunching and Analysis Made Easy*. Presses de l'Université du Québec, 2018.

[9] M. R. Huq, A. Wombacher, and P. M. G. Apers. Inferring fine-grained data provenance in stream data processing: Reduced storage cost, high accuracy. In A. Hameurlain, S. W. Liddle, K. Schewe, and X. Zhou, editors, *Proc. DEXA 2011, Part II*, volume 6861 of *LNCS*, pages 118–127. Springer, 2011.

[10] 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.

[11] H. Park, R. Ikeda, and J. Widom. RAMP: A system for capturing and tracing provenance in MapReduce workflows. *PVLDB*, 4(12):1351–1354, 2011.

[12] B. Pérez, J. Rubio, and C. Sáenz-Adán. A systematic review of provenance systems. *Knowl. Inf. Syst.*, 57(3):495–543, 2018.

[13] 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.

[14] M. Stonebraker, J. Chen, N. Nathan, C. Paxson, and J. Wu. Tioga: Providing data management support for scientific visualization applications. In R. Agrawal, S. Baker, and D. A. Bell, editors, *Proc. VLDB 1993*, pages 25–38. Morgan Kaufmann, 1993.

[15] 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.

[16] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In W. A. Gray and P. Larson, editors, *Proc. ICDE 1997*, pages 91–102. IEEE Computer Society, 1997.