Cognacy Queries over Dependence Graphs for Transparent Visualisations

Joe Bond[1], Cristina David[1], Minh Nguyen[1], Dominic Orchard[2,3], and Roly Perera[1,3(✉)]

[1] University of Bristol, Bristol, UK
{j.bond,cristina.david,min.nguyen}@bristol.ac.uk
[2] University of Kent, Canterbury, UK
d.a.orchard@kent.ac.uk
[3] University of Cambridge, Cambridge, UK
roly.perera@cl.cam.ac.uk

Abstract. Charts, figures, and text derived from data play an important role in decision making. But making sense of or fact-checking outputs means understanding how they relate to the underlying data. Even for experts with access to the source code and data sets, this poses a significant challenge. We introduce a new program analysis framework (A supporting artifact is available at https://zenodo.org/records/14637654 [5].) which supports interactive exploration of fine-grained IO relationships directly through computed outputs, using dynamic dependence graphs. This framework enables a novel notion in data provenance which we call *linked inputs*, a relation of mutual relevance or cognacy which arises between inputs that contribute to common features of the output. We give a procedure for computing linked inputs over a dependence graph, and show how the presented in this paper is faster on most examples than an implementation based on execution traces.

1 Introduction: Towards Transparent Research Outputs

Whether formulating national policy or making day-to-day decisions about our own lives, we increasingly rely on the charts, figures and text created by scientists and journalists. Interpreting these visual and textual summaries is essential to making informed decisions. However, most of the artifacts we encounter are *opaque*, inasmuch as they are unable to reveal anything about how they relate to the data they were derived from. Whilst one could in principle try to use the source code and data to reverse engineer some of these relationships, this requires substantial expertise, as well as valuable time spent away from the "comprehension context" in which the output in question was encountered. These difficulties are compounded when the information presented uses multiple data sources, such as medical meta-analyses [13], ensemble models in climate science [25], or queries that span multiple database tables [36]. Perhaps more often than we would like, we end up taking things on trust.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-91118-7_6

V. Vafeiadis (Ed.): ESOP 2025, LNCS 15694, pp. 144–171, 2025.
https://doi.org/10.1007/978-3-031-91118-7_6

With traditional print media, a "disconnect" between outputs and underlying data is unavoidable. For digital media, other options are open to us. One way to improve things is to engineer visual artifacts to be more "self-explanatory", revealing the relationship to the underlying data to an interested user. Consider the histogram in Figure 1 showing urban population growth in Asia over a decade [6]. A reader might have many questions about what this chart represents – in other words, how visual elements map to underlying data. Do the points represent individual cities? What does the colour scheme indicate? Which of the points represent large cities or small cities? Legends and other accompanying text can help, but some ambiguities inevitably remain. The purpose of a visual summary after all is to present the big picture at the expense of certain detail.

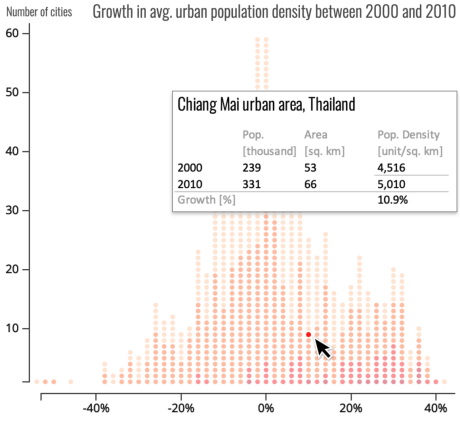

Fig. 1: A hand-crafted transparent visualisation due to Bremer and Ranzijn [6]

Bremer and Ranzijn's [6] approach to this problem is an interactive feature that allows a user to explore some of these provenance-related questions *in situ*, i.e. directly from the chart. Selecting an individual point shows the user a view of the data the point was calculated from. In the figure the highlighted point represents Chiang Mai, and shows that the plotted value of 10.9% was derived from an increase in population density from 4,416 to 5,010 people per sq. km. Features like these are valuable as comprehension aids, but are laborious to implement by hand and require the author to anticipate the queries a user might have. For these reasons they also tend not to generalise: for example Bremer and Ranzijn's [6] visualisation only allows the user to select one point at a time.

1.1 Data transparency as PL infrastructure

Hand-crafted efforts like Bremer and Ranzijn's [6] are labour-intensive because they involve manually embedding metadata about the relationship between outputs and inputs into the same program. When the visualisation or analysis logic changes, the relationship between inputs and outputs also change, and the metadata must be manually updated. A less costly and more robust approach is to treat data provenance as a language infrastructure problem, baking lineage or provenance metadata directly into outputs so that provenance queries can be supported automatically. For example, for an in-memory database engine, Psallidas and Wu [32] describe how to "backward trace" from output selections to input selections, and then "forward trace" to find related output selections in other views, to support a popular feature from data visualisation called *linked brushing*.

Perera et al. [30] implemented a similar system for a general-purpose functional programming language, using execution traces for bidirectional queries.

The advantage of shifting the burden of implementing transparency features onto the language runtime is that the author of the content can concentrate purely on data visualisation, and as the infrastructure improves, the benefits are inherited automatically by end users, at no additional cost to the author. For example if formal guarantees (perhaps that data selections are in some sense minimal and sufficient) are provided, then those can be proved once for the infrastructure rather than for each bespoke implementation.

In this paper, we propose a new bidirectional analysis framework for a general-purpose programming language, which supports fine-grained *in situ* provenance queries, an end-user feature we call *data transparency*. In contrast to prior work, we use *dynamic dependence graphs* [17,2] to implement the analyses, which enables our approach to be fast enough for interactive use and also language-independent, by separating queries over the graph from the problem of deriving the dependence graph for a particular program.

1.2 Contributions and Roadmap

Our specific contributions are as follows:

- § 3 defines a core calculus for data-transparent outputs, where parts of inputs and computed values are assigned unique labels called *addresses*, and programs have an operational semantics that pairs every result with a dynamic dependence graph capturing fine-grained IO relationships between input addresses and output addresses.
- § 4 presents a new formal framework for bidirectional provenance queries over dependence graphs, formalising two operators over such graphs, ∇ (*demands*) and $\triangle$ (*demanded by*). We show these to be *conjugate* in the sense of Jonsson and Tarski [22], and give procedures to compute ∇ and $\triangle$. We show how a novel cognacy operator $\nabla\triangle$ called *linked inputs*, relating inputs when they contribute to common parts of the output, can be obtained by composing ∇ and $\triangle$, and how a dual *linked outputs* operator $\triangle\nabla$, supporting the "linked visualisations" of prior work [32,30], is obtained by transposing the two operators.
- § 5 shows that our implementation performs better than one based on traces and bidirectional interpreters, the primary alternative implementation technique. Using usability metrics from Nielsen [26], we find that 81% of the queries we tested execute at a speed that appears instantaneous to a user, compared to 25% using an implementation based on traces. We also compare overhead of building a trace vs. building a dependence graph for a given program.

Although Psallidas and Wu [32] support fast queries for linked visualisations in a database setting, and Perera et al. [30] support linked outputs in a general-purpose languages, ours is (to the best of our knowledge) the first implementation for a general-purpose language which is fast enough for interactive use. § 6

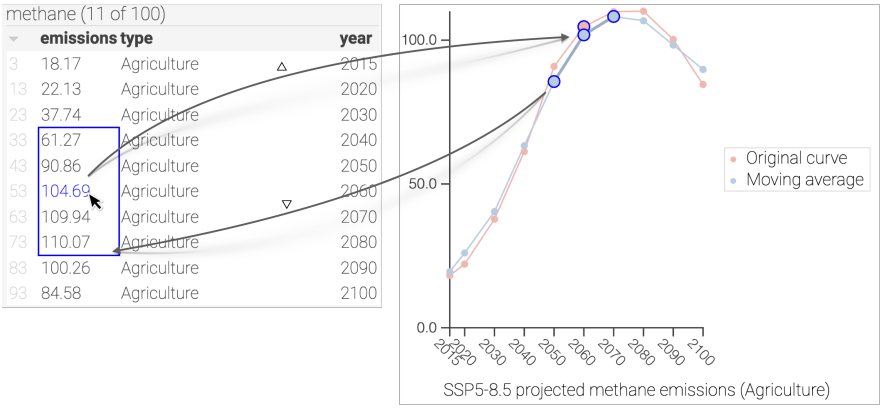

Fig. 2: Data transparency: *demanded by* (△) and *demands* (▽) operators link inputs that share common output dependencies.

discusses other related work in database provenance, program slicing and data visualisation.

§ 7 wraps up with a discussion of some limitations. In particular, the sort of transparency considered in this paper is purely extensional, and falls short of providing full explanations of how output parts are related to input parts. We discuss more intensional forms of transparency in § 7 and propose some other ways in which the present system could be improved.

We implement our approach in a pure functional programming language called Fluid. The author of a visualisation expresses their chart as a pure function of the inputs, using a set of built-in data types for common visualisations; a d3.js front end automatically enriches the rendered outputs with support for interactive selection and the data transparency queries introduced in the next section. Our implementation is available at https://github.com/explorable-viz/fluid.

2 Overview: Fine-Grained Interactive Provenance

In this section, we introduce the main interactive data provenance features that wish to support, using two Fluid examples to illustrate. The line chart in Figure 2 shows projected methane emissions from agricultural sources under a global warming scenario called RCP8.5, with source code in Figure 3; the scatter plot and stacked bar chart in Figure 4 show changing non-renewable energy outputs and capacities for various countries, with source code in Figure 5.

The key idea of a transparent visualisation is that the original data sources are kept around and can be viewed (when a user so requests) alongside the visualisation, and the user is then able to interact with both the data and the view to

```
let nthPad n xs = nth (min (max n 0) (length xs − 1)) xs;
let movingAvg ys window =
      [ sum [ nthPad n ys | n ← is ] / (1 + 2 * window)
      | i ← [ 0 .. length ys − 1 ],
        let is = [ i − window .. i + window ] ];
let movingAvg' rs window =
      zipWith (fun x y → {x: x, y: y})
         (map (fun r → r.x) rs)
         (movingAvg (map (fun r → r.y) rs) window);
let points =
      [ { x: r.year, y: r.emissions } | r ← methane, r.type == "Agriculture" ]
in LineChart {
   tickLabels: { x: Rotated, y: Default },
   size: { width: 330, height: 285 },
   caption: "SSP5−8.5 projected methane emissions (Agriculture)",
   plots: [
      LinePlot { name: "Original curve", points: points },
      LinePlot { name: "Moving average", points: movingAvg' points 1 }
   ] }
```

Fig. 3: Moving average source code

explore how they are related. Crucially the author of the visualisation does not to have to implement any of these interactions themselves; they need only write the visualisation code and the language runtime and rendering infrastructure provides the interactions.

1. Fine-grained linking of the data to the view. In Figure 2, the user has chosen to reveal the underlying data set, which is shown on the left; only rows relevant to the chart are visible, with the other rows hidden automatically. This clarifies that only agricultural data is relevant, confirming the (informal) claim in the caption. The emissions values shown do contribute to the chart in some way, and the user can investigate this by interacting with individual entries. For example, moving their mouse over the number 104.69 highlights *one* point in the projected emissions curve, and *three* points in the other curve, which plots the moving average (see Figure 3) of the projected emissions. The provenance analysis which underpins this we write as $\triangle$ ("demanded by"), and here this tells us that 104.69 was needed to compute either the x or y coordinate (in this case just the y coordinate) of the four highlighted points.

2. Linked inputs. We want to support fine-grained IO queries that run in the other direction too, via another provenance analysis written $\triangledown$ ("demands"). In Figure 2 the $\triangledown$ analysis is initiated automatically on the output of $\triangle$; this reveals that calculating the y coordinates of the points highlighted on the right required not only the 104.69 we started with, but 4 additional emissions values (blue border on the left). These additional inputs we refer to as the *related inputs* of the original input selection; they are the other inputs demanded by any output that demands our starting input, and are computed using the composite operator $\triangledown\triangle$. Given that the initial input selection here contributes to 3 points of the moving average, the "window" of related inputs in this case is 5 wide, comprising all the data points needed to account for the selection on the right. Related inputs is

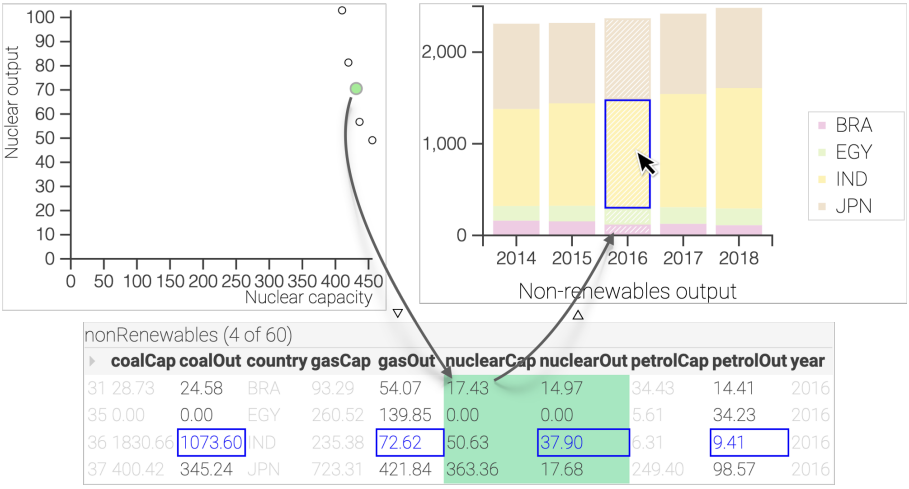

Fig. 4: Data transparency: *demands* (▽) and *demanded by* (△) operators link outputs that share common input dependencies.

a *cognacy* relation: two inputs are related if they have a common ancestor in a *dependence graph* that captures how inputs are demanded by outputs.

3. Fine-grained linking of the view to the data. We would also like to support cognacy queries that start from the output rather than the input; this is the basis of a feature called *linked brushing* (or *brushing and linking*) in data visualisation [4,7]. Here we would like to provide linked brushing in a way that is transparent to the user. In Figure 4, the user has again revealed the underlying data set, this time opting to see only rows with active data selections, rather than only rows with data that are used by any part of the output. They then express interest in one of the points in the scatter plot. By clicking on it, rather than just mousing over it, they create what we call a *persistent* selection (shown in green). The demands analysis ▽ reveals (also in green) the inputs needed to compute both the x and y coordinates of the selected point; this shows that the 2016 data for 4 countries was used, although it does not reveal how.

4. Linked outputs. Now that the inputs demanded by the output selection are determined, the △ analysis runs automatically on the output of ▽; this reveals that those inputs were also needed for 4 of the bar segments (highlighted with cross-hatching) in the bar chart, namely those for 2016. This is standard linked brushing, and is implemented using the composite operator △▽, which we call *related outputs*. Related outputs is a "co-cognacy" relation, relating two outputs whenever they have a common descendant, rather than common ancestor, in the dependence graph. Here the user can take advantage of the fact that a persistent selection remains active even when the mouse moves off the original

item of interest; this allows them to investigate further. Moving their mouse over the yellow IND segment of the 2016 bar (highlighted with blue border) initiates an orthogonal ▽ query which shows (also with various blue borders) the data needed to compute the yellow segment. Overlaid on the persistent selection, this reveals that the 37.90 in the nuclearOut column is needed to compute *both* the IND bar segment *and* the selected scatter plot point, explaining why the two output selections are linked.

```
let countries = ["BRA", "EGY", "IND", "JPN"];
let totalFor year country =
   let [ row ] =
      [ row | row ← nonRenewables, row.year == year, row.country == country ]
   in row.nuclearOut + row.gasOut + row.coalOut + row.petrolOut;
let stack year =
      [ { y: country, z: totalFor year country } | country ← countries ];
let yearData year =
      [ row | row ← nonRenewables,
              row.year == year, row.country `elem` countries ]
in MultiView {
   barChart: BarChart {
      caption: "Non−renewables output",
      size: { width: 275, height: 185 },
      stackedBars:
         [ { x: numToStr year, bars: stack year } | year ← [2014..2018] ]
   },
   scatterPlot: ScatterPlot {
      caption: "",
      points:
        [ { x: sum [ row.nuclearCap | row ← rows ],
            y: sum [ row.nuclearOut | row ← rows ] }
        | year ← [2014..2018], let rows = yearData year ],
      xlabel: "Nuclear capacity",
      ylabel: "Nuclear output"
   }
}
```

Fig. 5: Non-renewable energy source code

3 A Core Calculus for Transparent Outputs

We start by introducing a calculus for transparent outputs, which is the core of Fluid. The core language supports deep pattern-matching and mutual recursion. Fluid also provides a surface syntax that desugars into the core, providing piecewise function definitions, list comprehensions and other conveniences, as shown in Figures 3 and 5, which would otherwise complicate the core.

The term syntax of the core language (§ 3.1) is uncontroversial: it is pure and untyped, and provides datatypes, records and matrices (although we omit a presentation of matrices here). The syntax of values is less standard: each part of a value has a unique *address* α which serves to identify that part of the value in a dependence graph. We give a big-step operational semantics that evaluates a term to both a value and a *dynamic dependence graph* for that value (§ 3.2), capturing how parts of the value depend on parts of the input.

Expression

$e ::=$	x	variable
	n	integer
	let $x = e$ in e'	let
	$\{\overrightarrow{x : e}\}$	record
	$e.x$	record projection
	$c(\overrightarrow{e})$	constructor
	$e\, e'$	application
	$f(\overrightarrow{e})$	foreign application
	$\lambda\sigma$	function
	let ρ in e	recursive let

Continuation

$\kappa ::=$	e	expression
	σ	eliminator

Eliminator

$\sigma ::=$	$x \mapsto \kappa$	variable
	$\{\overrightarrow{x}\} \mapsto \kappa$	record
	$\{\overrightarrow{c \mapsto \kappa}\}$	constructor

Value

$v ::=$	$\boldsymbol{v}_\alpha$	
$\boldsymbol{v} ::=$	n	integer
	$\{\overrightarrow{x : v}\}$	record
	$c(\overrightarrow{v})$	constructor
	$\mathrm{cl}(\gamma, \rho, \sigma)$	closure

Environment

$\gamma ::= \{\overrightarrow{x : v}\}$

Recursive definitions

$\rho ::= \{\overrightarrow{x : \sigma}\}$

Fig. 6: Syntax of the core language, including values labeled with addresses

Some notation. We write to $\overrightarrow{x}$ denote a finite sequence of elements $x_1, .., x_n$, with ϵ as the empty sequence. Concatenation of sequences is written $\overrightarrow{x} +\!\!+ \overrightarrow{x'}$; we also write $x \cdot \overrightarrow{x}$ for cons (prepend) and $\overrightarrow{x} \cdot x$ for snoc (append). We write $\{\overrightarrow{k : x}\}$ to denote a finite map, i.e. a set of pairs $k_1 : x_1, .., k_n : x_n$ where keys k_i are pairwise unique. If X and Y are sets, we write $X \uplus Y$ to mean $X \cup Y$ where X and Y are disjoint, and also $x \cdot X$ or $X \cdot x$ to mean $X \uplus \{x\}$.

3.1 Syntax

Terms Figure 6 defines expressions e of the language, which include variables x, integer constants n, let-bindings let $x = e$ in e', record construction $\{\overrightarrow{x : e}\}$, record projection $e.x$ and (saturated) constructor expressions $c(\overrightarrow{e})$, where c ranges over data constructors. The language is parameterised by a finite map Σ from constructors c to arities $\Sigma(c) \in \mathbb{N}$. Function application has two forms: the usual $e\, e'$ and (saturated) *foreign function* application $f(\overrightarrow{e})$ (described below). Lastly, expressions include anonymous functions $\lambda\sigma$, where σ is a pattern-matching construct called an *eliminator* (§ 3.1), and sets of mutually recursive functions let ρ in e, where ρ is a finite map $\overrightarrow{x : \sigma}$ from names to eliminators.

The language is also parameterised by a finite map Φ from variables f to arities $\Phi(f) \in \mathbb{N}$ of the foreign function they denote. For the graph semantics in § 3.2 below, every foreign function name f is required to provide an interpretation $\hat{f}$ that, for a sequence of arguments $\overrightarrow{v}$ and dependence graph G, returns the result of applying the foreign function f to $\overrightarrow{v}$ plus a dependence graph G' which extends G with information about how the result depends on $\overrightarrow{v}$.

Continuations and eliminators A *continuation* κ is a term e or an eliminator σ, describing how an execution proceeds after a value is matched. *Eliminators* are a deep pattern-matching construct based on tries [21,31]; for a language with rich structured values like records and data types, they make for a cleaner presentation than single shallow elimination forms for each type. Piecewise function definitions in the surface language desugar into eliminators.

An eliminator specifies how values of a particular shape are matched, and for a given value, determines how any pattern variables get bound and the continuation κ which will be executed under those bindings. A variable eliminator $x \mapsto \kappa$ says how to match any value (as variable x) and continue as κ. A record eliminator $\{\vec{x}\} \mapsto \kappa$ says how to match a record with fields $\vec{x}$, and provides a continuation κ for sequentially matching the values of those fields. Lastly, a constructor eliminator $\{\overrightarrow{c \mapsto \kappa}\}$ provides a branch $c \mapsto \kappa$ for each constructor c in $\vec{c}$, where each κ specifies how any arguments to c will be matched. (We assume that any constructors matched by an eliminator all belong to the same data type, but this is not enforced in the core language.)

Addresses, values and environments We define addressed *values* v mutually inductively with *raw values*. A raw value is simply a value without an associated address; a value decorates a raw value with an address α. Addresses are allocated during evaluation so that new partial values have fresh addresses, which can then be used as vertices in a dependence graph.

Raw values $\boldsymbol{v}$ include integers n, m; records $\{\overrightarrow{x \mapsto v}\}$; and constructor values $c(\vec{v})$. Raw values also include closures $\mathrm{cl}(\gamma, \rho, \sigma)$ where σ is the function body, γ the captured environment, and to support mutual recursion, ρ is the (possibly empty) set of named functions with which σ was mutually defined. *Environments* are finite maps from variables to values. Because foreign functions in the core language are not first-class and calls $f(\vec{e})$ are saturated, i.e. $|\vec{e}| = \Phi(f)$, the surface language provides a top-level environment which maps every foreign function name f of arity n to the closure $\mathrm{cl}(\varnothing, \varnothing, x_1 \mapsto .. \mapsto x_n \mapsto f(\vec{x}))_\alpha$, with α fresh, emulating first-class foreign functions.

3.2 Operational Semantics

We now give a big-step operational semantics for the core language, which evaluates a term to a value paired with a *dynamic dependence graph* for that value.

Dynamic Dependence Graphs A *dynamic dependence graph* [2] (hereafter *dependence graph*) is a directed acyclic graph $G = (V, E)$ with a set V of *vertices* and a set $E \subseteq V \times V$ of edges. When convenient we write $\mathsf{V}(G)$ for V and $\mathsf{E}(G)$ for E. In the dependence graph for a particular program, vertices $\alpha, \beta \in V$ are addresses associated to values (either supplied to the program as inputs or produced during evaluation) and edges $(\alpha, \beta) \in E$ indicate that, in the evaluation of that program, the value associated to β *depends on* (is *demanded by*, in the terminology of § 2) the value associated to α. Such values may be sub-terms of

a larger value. This diverges somewhat from traditional approaches to dynamic dependence graphs in only considering one type of edge (data dependency) rather than separate data and control dependencies; moreover our edges point in the direction of dependent vertices, whereas in the literature the other direction is somewhat more common.

During evaluation, when a fresh vertex α is allocated for a constructed value, the dependence graph G is extended by a graph fragment specifying that α depends on a set V of preexisting vertices, given by the following notation:

Definition 1 (In-star notation). *Write* $\{V \mapsto \alpha\}$ *as shorthand for the star graph* $(\{\alpha\} \uplus V, V \times \{\alpha\})$.

Pattern matching Evaluation relies on pattern matching, which is defined by a separate judgement form, computing a vertex set. Figure 7 defines the pattern-matching judgement $\overrightarrow{v}, \kappa \rightsquigarrow \gamma, e, V$. Rather than matching a single value v, the judgement matches a "stack" of values $\overrightarrow{v}$ against a continuation κ, returning the selected branch e, an environment γ providing bindings for the free variables of e, and the set V of addresses found in the matched portions of $\overrightarrow{v}$.

A continuation which is just an expression e matches only the empty stack of values ($\rightsquigarrow$-done), in which case pattern-matching is complete and e is the selected branch. Other rules require an *eliminator* σ as the continuation and a non-empty stack $v \cdot \overrightarrow{v}$; any relevant subvalues of v are unpacked and pushed onto the tail $\overrightarrow{v}$ and then recursively matched using the continuation κ selected from σ. A variable eliminator $x \mapsto \kappa$ pops v off the stack, using κ to recurse ($\rightsquigarrow$-var); no part of v is consumed so the addresses V consumed by the recursive match are returned unmodified. A record eliminator $\{\overrightarrow{y}\} \mapsto \kappa$ matches a record of the form $\{\overrightarrow{x : v}\}_\alpha$ as long as the variables in $\overrightarrow{y}$ are also fields in $\overrightarrow{x}$, augmenting V with the address α associated with the record ($\rightsquigarrow$-record); the premise $\{\overrightarrow{y : u}\} \subseteq \{\overrightarrow{x : v}\}$ projects out the corresponding values of $\overrightarrow{u}$ from $\overrightarrow{v}$. Lastly, a constructor eliminator $(c \mapsto \kappa)$ matches any constructor value of the form $c(\overrightarrow{v})_\alpha$, augmenting V with the address α associated with the constructor ($\rightsquigarrow$-constr).

$\boxed{\overrightarrow{v}, \kappa \rightsquigarrow \gamma, e, V}$

$\rightsquigarrow$-done
$$\frac{}{\epsilon, e \rightsquigarrow \varnothing, e, \varnothing}$$

$\rightsquigarrow$-constr
$$\frac{\overrightarrow{v} +\!\!+ \overrightarrow{v}', \kappa \rightsquigarrow \gamma, e, V}{c(\overrightarrow{v})_\alpha \cdot \overrightarrow{v}', (c \mapsto \kappa) \cdot \{\overrightarrow{c \mapsto \kappa}\} \rightsquigarrow \gamma, e, \alpha \cdot V} \; \Sigma(c) = |\overrightarrow{v}|$$

$\rightsquigarrow$-var
$$\frac{\overrightarrow{v}, \kappa \rightsquigarrow \gamma, e, V}{v \cdot \overrightarrow{v}, x \mapsto \kappa \rightsquigarrow \gamma \cdot (x : v), e, V}$$

$\rightsquigarrow$-record
$$\frac{\{\overrightarrow{y : u}\} \subseteq \{\overrightarrow{x : v}\} \qquad \overrightarrow{u} +\!\!+ \overrightarrow{v}', \kappa \rightsquigarrow \gamma, e, V}{\{\overrightarrow{x : v}\}_\alpha \cdot \overrightarrow{v}', \{\overrightarrow{y}\} \mapsto \kappa \rightsquigarrow \gamma, e, \alpha \cdot V}$$

Fig. 7: Pattern matching

$\boxed{\gamma, e, V, G \Rightarrow v, G'}$

$$\Rightarrow\text{-var}\quad \frac{}{\gamma \cdot (x : v), x, V, G \Rightarrow v, G} \qquad \Rightarrow\text{-int}\quad \frac{\alpha \notin \mathsf{V}(G)}{\gamma, n, V, G \Rightarrow n_\alpha, G \cup \{V \mapsto \alpha\}}$$

$$\Rightarrow\text{-function}\quad \frac{\alpha \notin \mathsf{V}(G)}{\gamma, \lambda\sigma, V, G \Rightarrow \mathrm{cl}(\gamma, \varnothing, \sigma)_\alpha, G \cup \{V \mapsto \alpha\}}$$

$$\Rightarrow\text{-record}\quad \frac{\gamma, \overrightarrow{e}, V, G \Rightarrow \overrightarrow{v}, G' \qquad \alpha \notin \mathsf{V}(G')}{\gamma, \{\overrightarrow{x : e}\}, V, G \Rightarrow \{\overrightarrow{x : v}\}_\alpha, G' \cup \{V \mapsto \alpha\}} \qquad \Rightarrow\text{-project}\quad \frac{\gamma, e, V, G \Rightarrow \{\overrightarrow{x : v} \cdot (y : u)\}_\alpha, G'}{\gamma, e.y, V, G \Rightarrow u, G'}$$

$$\Rightarrow\text{-constr}\quad \frac{\gamma, \overrightarrow{e}, V, G \Rightarrow \overrightarrow{v}, G' \qquad \alpha \notin \mathsf{V}(G')}{\gamma, c(\overrightarrow{e}), V, G \Rightarrow c(\overrightarrow{v})_\alpha, G' \cup \{V \mapsto \alpha\}}\ \Sigma(c) = |\overrightarrow{e}|$$

$$\Rightarrow\text{-foreign-app}\quad \frac{\gamma, \overrightarrow{e}, V, G_1 \Rightarrow \overrightarrow{v}, G_2 \qquad \hat{f}(\overrightarrow{v}, G_2) = (u, G_3)}{\gamma, f(\overrightarrow{e}), V, G_1 \Rightarrow u, G_3}\ \Phi(f) = |\overrightarrow{e}|$$

$$\Rightarrow\text{-let}\quad \frac{\gamma, e, V, G_1 \Rightarrow v, G_2 \qquad \gamma \cdot (x : v), e', V, G_2 \Rightarrow v', G_3}{\gamma, \mathrm{let}\ x = e\ \mathrm{in}\ e', V, G_1 \Rightarrow v', G_3}$$

$$\Rightarrow\text{-let-rec}\quad \frac{\gamma, \rho, V, G_1 \rightarrowtail \gamma', G_2 \qquad \gamma +\!\!+ \gamma', e, V, G_2 \Rightarrow v, G_3}{\gamma, \mathrm{let}\ \rho\ \mathrm{in}\ e, V, G_1 \Rightarrow v, G_3}$$

$$\Rightarrow\text{-app}\quad \frac{\begin{array}{c}\gamma, e, V, G_1 \Rightarrow \mathrm{cl}(\gamma_1, \rho, \sigma)_\alpha, G_2 \qquad \gamma_1, \rho, \{\alpha\}, G_2 \rightarrowtail \gamma_2, G_3 \qquad \gamma, e', V, G_3 \Rightarrow v', G_4 \\ v', \sigma \rightsquigarrow \gamma_3, e'', V' \qquad \gamma_1 +\!\!+ \gamma_2 +\!\!+ \gamma_3, e'', V' \cup \{\alpha\}, G_4 \Rightarrow u, G_5\end{array}}{\gamma, e\, e', V, G_1 \Rightarrow u, G_5}$$

$\boxed{\gamma, \overrightarrow{e}, V, G \Rightarrow \overrightarrow{v}, G'}$

$$\frac{\gamma, e_i, V, G_i \Rightarrow v_i, G_{i+1} \qquad (\forall i \le n)}{\gamma, \overrightarrow{e}, V, G_1 \Rightarrow \overrightarrow{v}, G_{n+1}}\ n = |\overrightarrow{e}|$$

$\boxed{\gamma, \rho, V, G \rightarrowtail \gamma', G'}$

$$\frac{\begin{array}{c}\gamma'(x_i) = \mathrm{cl}(\gamma, \rho, \rho(x_i))_{\alpha_i} \\ \alpha_i \notin \mathsf{dom}(G_i) \qquad G_{i+1} = G_i \cup \{V \mapsto \alpha_i\} \quad (\forall i \le n)\end{array}}{\gamma, \rho, V, G_1 \rightarrowtail \gamma', G_{n+1}}\ n = |\overrightarrow{x}|$$

Fig. 8: Operational semantics with dependence graph

Evaluation Figure 8 defines a big-step evaluation relation $\gamma, e, V, G \Rightarrow v, G'$ stating that term e, under an environment γ, vertex set V and dependence graph G, evaluates to a value v and extended dependence graph G'. The vertex set V records the (partial) input values consumed by the current active function call providing the dynamic context in which e is being evaluated; V is initially empty and changes whenever a function application is evaluated.

The evaluation rule for variables is fairly standard; the dependence graph is returned unmodified, because no new addresses are allocated as a result of simply looking up a variable. The rule for record projections $e.y$ is similar: if e evaluates to a record of the form $\{\overrightarrow{x : v} \cdot (y : u)\}_\alpha$, then $e.y$ evaluates to the value u of field y, discarding the address α of the record.

Introduction rules, such as for integers, functions, records and constructors, follow the pattern of assigning a fresh address for the (partial) value being constructed, and then extending G with a set of dependency edges from the vertices in V to α. For example, the integer rule evaluates an expression n to its value form n_α; the $\alpha \notin \mathsf{V}(G)$ constraint ensures that α is fresh. The dependence graph is then extended with $\{V \mapsto \alpha\}$, indicating that n_α depended on all matched partial inputs in V. Likewise, the rule for an anonymous function $\lambda\sigma$ constructs the closure $\mathrm{cl}(\gamma, \varnothing, \sigma)_\alpha$ with fresh address α, capturing the current environment γ and using $\varnothing$ as the set of mutually recursive definitions associated with the function, and again establishing dependency edges from V to α_i.

Rules which involve recursively evaluating subterms thread the graph under construction through the evaluation of the subterms. For example, the auxiliary evaluation relation $\gamma, \overrightarrow{e}, V \Rightarrow \overrightarrow{v}, G$ (bottom of Figure 8), evaluates each e_i in a sequence of terms $\overrightarrow{e}$ to a value v_i and dependence graph G_{i+1}, which is used as the input graph for evaluating e_{i+1}. We make use of this judgement in the rules for records $\{\overrightarrow{x : e}\}$, constructors $c(\overrightarrow{e})$, and foreign applications $f(\overrightarrow{e})$; in the last rule, $\hat{f}$ is the foreign implementation that evaluates an application of f to a sequence of values $\overrightarrow{v}$ and dependence graph G to a result v and extended dependence graph G' (§ 3.1).

The rules for let $x = e$ in e' and application $e\,e'$ may involve mutual recursion, and rely on the auxiliary relation $\gamma, \rho, V, G \rightarrowtail \gamma', G'$ defined at the bottom of Figure 8. This judgement takes a set ρ of recursive definitions, vertex set V and dependence graph G, and returns an environment γ' of closures derived from ρ and extended dependence graph G'. Each function definition $\rho(x_i) \in \rho$ generates a new closure $(\gamma, \rho, \rho(x_i))_{\alpha_i}$ capturing γ and ρ, with fresh address α_i, and extends the dependence graph with a set of edges from V to α_i.

The evaluation rule for let ρ in e is similar to the rule for regular let-bindings, except for using $\rightarrowtail$ to build a set of closures γ' which extends γ. Finally, the rule for $e\,e'$ is notable because this is where the active function context changes and the ambient V is discarded. We compute the closure $\mathrm{cl}(\gamma_1, \rho, \sigma)_\alpha$ from e and the argument v' from e', and then use σ to match v'. If pattern-matching returns selected branch e'', with vertex set V' representing the consumed part of v', then e'' is evaluated and becomes the result of the application, with $V' \cup \{\alpha\}$ serving as set of (partial) inputs associated with the new active function context.

Example Figure 9 shows part of the dependence graph for the moving average example in Figures 2 and 3. The graph shows the calculation of the first 3 points in the moving average plot from entries in the *emissions* table. This is a simplified version of the graph, since in practice they get very large; for example, we omit closures, list cells and the calculation of the divisors. Also note that the node labels here are merely illustrative: the graph only stores value dependencies, and in particular the in-neighbours of a vertex are unordered.

4 Cognacy Queries Over Dependence Graphs

We now turn to cognacy and "co-cognacy" queries over dependence graphs, expressed in terms of the ∇ and Δ operators introduced informally in § 2. Boolean algebras (Definition 2) are used to represent selections; we then define ∇ and Δ and their De Morgan duals, first for an arbitrary relation (§ 4.1) and then for the *IO relation* of a dependence graph (§ 4.2). Then we give procedures for computing ∇ and Δ and their duals over a given dependence graph, via an intermediate graph slice (§ 4.3).

Definition 2 (Boolean algebra). *A* Boolean algebra *(or* Boolean lattice*) A is a 6-tuple* $(A, \wedge, \vee, \bot, \top, \neg)$ *with carrier A, distinguished elements* $\bot$ *(bottom) and* $\top$ *(top), commutative operations* $\wedge$ *(meet) and* $\vee$ *(join) with* $\top$ *and* $\bot$ *as respective units, and unary operation* $\neg$ *(negate) satisfying* $x \vee \neg x = \top$ *and* $x \wedge \neg x = \bot$*. The operations* $\wedge$ *and* $\vee$ *distribute over each other and induce a partial order* $\leq$ *over* X*.*

An input (or output) selection, where X is the set of all addresses that occur in the input (or output) of the program, is represented by the *powerset* Boolean algebra $\mathbb{P}(X)$ whose carrier is the set of subsets $X' \subseteq X$ ordered by inclusion $\subseteq$. In this case bottom is the empty set $\varnothing$, top is the whole set X, meet and join are given by $\cap$ and $\cup$, and negation by relative complement $\setminus$, so that $\neg X' = X \setminus X'$.

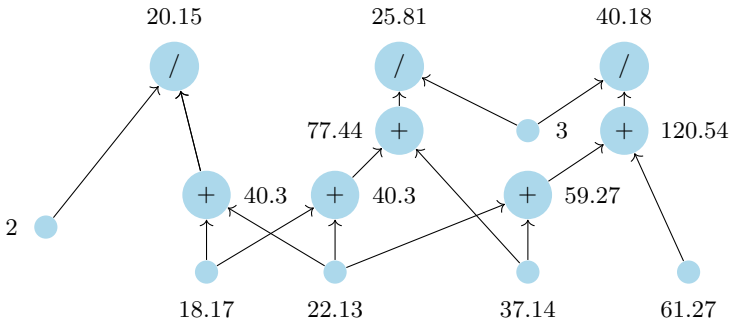

Fig. 9: Portion of dependence graph computed for moving averages example

4.1 $\triangledown$ and $\triangle$ and their duals

The $\triangledown$ and $\triangle$ operators sketched earlier are simply the image and preimage functions for a particular dependency relation R. A mnemonic device may help with reading the symbols; if R is a dependency relation oriented with outputs at the top and inputs at the bottom, then *demanded by* $\triangle$ can be read as an arrow pointing from inputs to outputs, and *demands* $\triangledown$ as an arrow pointing from outputs to inputs.

Definition 3 (Image and Preimage Functions for a Relation). *For a relation $R \subseteq X \times Y$, define $\triangle_R : \mathbb{P}(X) \to \mathbb{P}(Y)$ and $\triangledown_R : \mathbb{P}(Y) \to \mathbb{P}(X)$ as:*

1. $\triangle_R(X') := \{y \in Y \mid \exists x \in X'.(x, y) \in R\}$ *(image)*
2. $\triangledown_R(Y') := \{x \in X \mid \exists y \in Y'.(x, y) \in R\}$ *(preimage)*

Trivially $\triangle_R = \triangledown_{R^{-1}}$. The image and preimage functions form a *conjugate pair*, in the sense of Jonsson and Tarski [22]:

Definition 4 (Conjugate Functions). *For Boolean algebras A, B, functions $f : A \to B$ and $g : B \to A$ form a conjugate pair iff:*

$$f(x) \wedge y = \bot \iff x \wedge g(y) = \bot$$

(Jonsson and Tarski [22] consider only endofunctions; here we extend the idea of conjugacy to maps between Boolean algebras.)

Lemma 1. *$\triangle_R$ and $\triangledown_R$ are conjugate.*

This should be intuitive enough: for any subsets $X' \subseteq X$ and $Y' \subseteq Y$, if the elements "on the right" to which X' is related are disjoint from Y', then there are no edges in R from X' to Y'; and in virtue of that, the elements "on the left" to which Y' is related must also be disjoint from X'.

Conjugate functions are related to another class of near-reciprocals between Boolean algebras, namely *Galois connections*; in fact every conjugate pair induces a Galois connection [24]. This relates the present setting to previous work on program slicing with Galois connections [28,29,33].

Definition 5 (Galois connection). *Suppose A, B are partial orders, then monotone functions $f : A \to B$ and $g : B \to A$ form a Galois connection iff*

$$f(x) \leq y \iff x \leq g(y)$$

Proposition 1. *Suppose functions $f : A \to B$ and $g : B \to A$ between Boolean algebras. The following statements are equivalent:*

1. *f and g form a conjugate pair*
2. *f and g° form a Galois connection*

It is also useful (both for performance reasons and as a user feature) to consider the *De Morgan duals* of $\triangledown$ and $\triangle$, which we write as $\blacktriangle$ and $\blacktriangledown$; these also form a conjugate pair.

Definition 6 (De Morgan Dual). *Suppose a function $f : A \to B$ between Boolean algebras with $\neg_A$ and $\neg_B$ the negation operators of A and B. Define the De Morgan Dual $f^\circ : A \to B$ of f as:*

$$f^\circ := \neg_B \circ f \circ \neg_A$$

Definition 7 (Dual Image and Preimage Functions for a Relation). *For relations $R \subseteq X \times Y$, define $\blacktriangle_R : \mathbb{P}(X) \to \mathbb{P}(Y)$ and $\blacktriangledown_R : \mathbb{P}(Y) \to \mathbb{P}(X)$ as:*

1. $\blacktriangle_R(X') := \{y \in Y \mid \nexists x \in X \setminus X'.(x, y) \in R\}$ *(dual image)*
2. $\blacktriangledown_R(Y') := \{x \in X \mid \nexists y \in Y \setminus Y'.(x, y) \in R\}$ *(dual preimage)*

Bearing in mind that $\neg$ for $\mathbb{P}(X)$ is just relative complement $X \setminus \cdot$, it is easy to show that these are indeed the intended De Morgan duals.

Lemma 2 (Duality of image and preimage functions).

1. $\triangle_R^\circ = \blacktriangle_R$
2. $\triangledown_R^\circ = \blacktriangledown_R$

If $\triangle_R(X')$ picks out the outputs that the elements of X' are *necessary* for, $\blacktriangle_R(X')$ picks out the outputs that the elements of X are *sufficient* for. $\blacktriangledown_R(Y')$ picks out the inputs that are needed *only* by elements of Y'.

4.2 $\triangledown_G$ and $\triangle_G$ for Dependence Graphs

Reachability in G induces a relation just between *sources* and *sinks*, which we call the *IO* relation of G. We now extend $\triangledown_R$ and $\triangle_R$ and their duals to a dependence graph G, via its IO relation.

Definition 8 (Sources and sinks). *For a graph $G = (V, E)$, write $\mathsf{S}(G)$ for the* sources *of G (i.e. those vertices with no in-edges) and $\mathsf{T}(G)$ for the* sinks *of G (those with no out-edges).*

Definition 9 (Reachability relation). *Define the* reachability relation *for a graph $G = (V, E)$ to be the reflexive transitive closure of E.*

Definition 10 (IO relation). *For any dependence graph G with reachability relation R, define the* IO *relation of G to be $R \cap (\mathsf{S}(G) \times \mathsf{T}(G))$.*

The IO relation specifies how specific inputs (sources) are demanded by specific outputs (sinks); thus we interpret $(x, y) \in R$ as "x is *demanded by* y". Clearly the IO relation of G is the converse of the IO relation of G^{-1}. The set of inputs that some outputs Y *demand*, $\triangledown_G(Y)$, is simply the subset of the inputs of G reachable from Y (by traversing the graph in its opposite direction). Conversely, the set of outputs *demanded by* some inputs X, $\triangle_G(X)$, is simply the subset of the outputs of G reachable from X.

Definition 11 ($\triangledown_G$ and $\triangle_G$ for a dependence graph). *For a graph G with IO relation R, define:*

1. $\triangledown_G := \triangledown_R : \mathbb{P}(\mathsf{T}(G)) \to \mathbb{P}(\mathsf{S}(G))$ *(demands)*
2. $\triangle_G := \triangle_R : \mathbb{P}(\mathsf{S}(G)) \to \mathbb{P}(\mathsf{T}(G))$ *(demanded by)*

4.3 Computing $\triangledown_G$, $\triangle_G$, $\blacktriangle_G$ and $\blacktriangledown_G$ for Dependence Graphs

We now show how to compute e$\triangledown_G$, $\triangle_G$, $\blacktriangle_G$ and $\blacktriangledown_G$ for a dependence graph G, via an intermediate graph slice which we then restrict to its IO relation. First some graph notation:

Definition 12 (In-edges and out-edges). *For a graph $G = (V, E)$ and vertex $\alpha \in V$, write $\mathsf{inE}_G(\alpha)$ for the in-edges of α in G and $\mathsf{outE}_G(\alpha)$ for its out-edges.*

Definition 13 (Opposite graph). *For a graph G define the* opposite *graph $G^{-1} := (V, E^{-1})$ where $\cdot^{-1}$ denotes relational converse.*

With the trace-based approaches mentioned in § 1, one can use a given algorithm to implement its De Morgan dual; for example, given a procedure for $\triangle_G$ we can compute $\blacktriangle_G$ by pre- and post-composing with negation [30]. In the dependence graph setting, we can also use a given algorithm to compute its conjugate, e.g., given a procedure for $\triangle_G$, we can compute $\triangledown_G$ simply as $\triangle_{G^{-1}}$.

For efficiency, however, we give direct procedures both for $\triangle_G$ (§ 4.3) and $\blacktriangle_G$ (§ 4.3), which can then be used to derive implementations of the other operators, as shown at the end of this section. Each procedure factors through an auxiliary operation that computes a "slice" of the original graph, i.e. a contiguous subgraph. While it is technically possible to compute the desired image/preimage of the IO relation without creating this intermediate graph, we anticipate use cases which will make use of the graph slice; these are discussed in more detail in § 7. Our implementation makes it easy to flip between G and G^{-1} so we freely make use of both in the algorithms to access out-neighbours and in-neighbours.

Direct algorithm for $\triangle_G$ (*demanded by*)

Definition 14 (demBy_G). *Figure 10a defines the family of relations demBy_G for dependence graph G.*

For a dependence graph G and inputs $X \subseteq \mathsf{S}(G)$, the judgement $X\ \mathsf{demBy}_G Y$ says that X is demanded by $Y \subseteq \mathsf{T}(G)$. The algorithm defers to an auxiliary operation demByV, which takes a (partial) graph slice $H \subseteq G$, initially containing only the original input vertices X and no edges, and which proceeds as follows. If there is a sink of H that still has outgoing edges in G, then the targets of those edges are reachable from the vertices of H, and so we must add them to H and recurse (extend). If $\mathsf{T}(H) \subseteq \mathsf{T}(G)$, then there are no unexplored nodes in G that are reachable from H, and so we terminate with the current state of H (done). We note that it is enough to consider the sinks of H, since we move every outgoing edge of a vertex from G to H all at once. When demByV is done, demBy_G returns the sinks $T(H)$ from the final graph slice H, representing the outputs that the original inputs are demanded by.

Proposition 2 (demBy_G Computes $\triangle_G$). *For any dependence graph G, any $X \subseteq \mathsf{S}(G)$ and any $Y \subseteq \mathsf{T}(G)$ we have*

$$X\ \mathsf{demBy}_G\ Y \iff \triangle_G(X) = Y$$

Proof. See the supplementary material.

$X \ \mathsf{demBy}_G \ Y$

$$\frac{(X, \varnothing), G \ \mathsf{demByV} \ H}{X \ \mathsf{demBy}_G \ \mathsf{T}(H)} \ X \subseteq \mathsf{S}(G)$$

$H, G \ \mathsf{demByV} \ H'$

$$\frac{}{H, G \ \mathsf{demByV} \ H} \ \mathsf{done} \quad \mathsf{T}(H) \subseteq \mathsf{T}(G)$$

$$\frac{H \cup E, G \setminus_{\mathsf{E}} E \ \mathsf{demByV} \ H'}{H, G \ \mathsf{demByV} \ H'} \ \mathsf{extend} \quad \alpha \in \mathsf{T}(H) \wedge E = \mathsf{outE}_G(\alpha) \neq \varnothing$$

(a) demBy_G algorithm

$X \ \mathsf{suff}_G \ U$

$$\frac{(X, \varnothing), (\varnothing, \varnothing), G \ \mathsf{suffE} \ H}{X \ \mathsf{suff}_G \ \mathsf{V}(H) \cap \mathsf{T}(G)} \ X \subseteq \mathsf{S}(G)$$

$H, P, G \ \mathsf{suffE} \ H'$

$$\frac{}{H, P, G \ \mathsf{suffE} \ H} \ \mathsf{done} \quad \mathsf{S}(G) \cap \mathsf{V}(P) \subseteq \mathsf{S}(P) \wedge \mathsf{V}(H) \subseteq \mathsf{T}(G)$$

$$\frac{H, P \cup E, G \setminus_{\mathsf{E}} E \ \mathsf{suffE} \ H'}{H, P, G \ \mathsf{suffE} \ H'} \ \mathsf{pending} \quad \alpha \in \mathsf{V}(H) \wedge E = \mathsf{outE}_G(\alpha) \neq \varnothing$$

$$\frac{H \cup E, P \setminus_{\mathsf{E}} E, G \ \mathsf{suffE} \ H'}{H, P, G \ \mathsf{suffE} \ H'} \ \mathsf{extend} \quad \alpha \in \mathsf{S}(G) \wedge E = \mathsf{inE}_P(\alpha) \neq \varnothing$$

(b) suff_G algorithm

Fig. 10: Dual analyses over a dependence graph G

Direct algorithm for $\blacktriangle_G$ (*suffices*)

Definition 15 (suff_G). *Figure 10b defines the family of relations* suff_G *for dependence graph* G.

Like $\mathsf{demands}_G$, the algorithm suff_G also delegates to an auxiliary operation suffE which builds a slice of G. The operation suffE takes the graph slice H under construction, a "pending" subgraph P of nodes for which we have discovered partial information, and the remaining unexplored graph G, and returns the final graph slice H'. Whenever we have a vertex in H which still has outgoing edges in G, we add those edges and their endpoints to the pending graph P (pending). If a vertex α has no more incoming edges in G, we can then move α and its incoming edges from P to H (extend), as this only happens when we have already moved every ancestor of α into H. If neither scenario is the case, then we terminate with the (potentially incomplete) graph H (done). Once suffE has terminated with H, suff_G returns just the vertices in H that are also sinks in G. If there are no such vertices, it simply means that the input data X was insufficient to compute any of the original program's outputs.

We provide an example of a run of the algorithm in Figure 11. Here, the green vertices and thick edges are in H, the orange vertices and dashed arrows are in P, and the blue vertices and normal arrows are in G. In the figure, steps 2, 3, 5 and 6 correspond to applications of the rule pending, and steps 4 and 7 correspond to applications of the rule extend. After step 7, the run is complete.

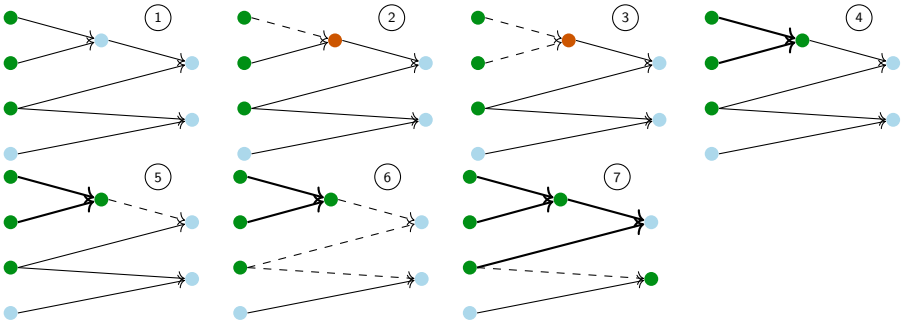

Fig. 11: Run of suff with G and H superimposed on G_0

Proposition 3 (suff_G Computes $\blacktriangle_G$). *For any dependence graph* G, *any* $X \subseteq \mathsf{S}(G)$ *and any* $Y \subseteq \mathsf{T}(G)$ *we have:*

$$X \ \mathsf{suff}_G \ Y \iff \blacktriangle_g(X) = Y$$

Proof. See the supplementary material.

Derived algorithms Propositions 2 and 3 justify treating demBy_G and suff_G as functions. Using Lemma 2 we can define a direct algorithm for $\triangledown_G$ from demBy_G by simply flipping the graph, and the same can be done to acquire an algorithm for $\blacktriangledown_G$ from suff_G. Using Lemma 2 and Definition 6 we can also define alternative algorithms for all 4 operators using the De Morgan dual construction. These are summarised in the table below.

Abstract operator	*Direct algorithm*	*Alternative algorithm*
$\triangle_G$	demBy_G	${\mathsf{suff}_G}^{\circ}$
$\triangledown_G$	$\mathsf{demBy}_{G^{-1}}$	${\mathsf{suff}_{G^{-1}}}^{\circ}$
$\blacktriangle_G$	suff_G	${\mathsf{demBy}_G}^{\circ}$
$\blacktriangledown_G$	$\mathsf{suff}_{G^{-1}}$	${\mathsf{demBy}_{G^{-1}}}^{\circ}$

In § 5 we contrast the performance of the direct and De Morgan dual implementations of $\triangle_G$, used to implement linked inputs/outputs as shown in § 1.

4.4 $\triangledown\triangle_G$ and $\triangle\triangledown_G$ for Dependence Graphs

Now we can provide a formal account of the notions of linked inputs and outputs.

Definition 16 (Linked inputs and outputs). *For a dependence graph G let:*

1. $\triangledown\triangle_G := \triangledown_G \circ \triangle_G$ *(linked inputs)*
2. $\triangle\triangledown_G := \triangle_G \circ \triangledown_G$ *(linked outputs)*

Intuitively, linked outputs and linked inputs are relations of *cognacy* (common ancestry) in G and G^{-1} respectively. For a set of inputs $X \subseteq \mathsf{S}(G)$, linked inputs asks "what other inputs are demanded by the outputs which demand X?"; it first finds all outputs $\triangle_G(X)$ that our inputs are demanded by, and then computes the inputs that those outputs demand, i.e. $\triangledown_G(\triangle_G(X'))$. Conversely for a set of outputs $Y \subseteq \mathsf{T}(G)$, linked outputs asks "what other outputs demand the inputs that Y demands"; it first finds all inputs $\triangledown_G(Y')$ that our outputs demand, and then computes the outputs that those inputs $\triangledown_G(Y)$ demand, i.e. $\triangle_G(\triangledown_G(Y))$.

5 Evaluation

We now compare two implementations of the dependency-tracking runtime of Fluid, one based on the dependence graph design described in Sections 3 and 4 and one based on the main alternative style of implementation, namely a bidirectional interpreter with two components: a forward evaluator which performs a forwards analysis and produces an execution trace, and a backward evaluator which consumes the execution trace and performs a backwards analysis [33,32,30]. Other Fluid system components were shared by the two implementations, including the parser, desugaring and visualisation layers, and libraries.

For a language implementor, the main benefit of our graph approach is that it removes the need to implement a bidirectional interpreter. Otherwise, changing the language requires modifying each direction of the interpreter in a manner

that maintains bidirectionality — a considerable effort. In our approach the graph algorithms for computing ∇_G and $\triangle_G$ (Figure 10) are language-agnostic and only have to be defined once.

For a user, our hypothesis is that our approach has better performance than trace-based techniques; to test this, we evaluated a number of benchmark programs taking examples from data analysis and data visualisation (§ 5 below). Because interactive performance plays a critical role for the user, we measured the following:

- Q1: Overhead of building a dependence graph versus building a trace;
- Q2: Performance of computing *demands* over a graph vs. a trace;
- Q3: Performance of various implementations of *demanded by*: trace-based, graph-based using `demands`, and graph-based using the dual of `suff`.

We assessed performance according to the guidelines proposed by Nielsen [26]:

- 100ms: limit for an interaction feeling *instantaneous*;
- 1000ms: limit for a user's train of thought to remain uninterrupted, although they may perceive a delay;
- 10000ms: limit to keep a user's attention on the system.

Ideally, interactive queries (Q2 and Q3) would stay as close as possible to the *instantaneous* category, and the time taken to evaluate programs (Q1) would stay within the limit for keeping a user's attention. Experiments were timed on an Intel Core i7-10850H 2.70GHz with 16Gb of RAM, using Chrome 121.0.6261.69 and JavaScript runtime V8 12.2.281.16. We used PureScript as the host language.

Choice of examples We collected some canonical examples from data visualization and analytics. First, we considered matrix convolution. Given that different kernels give rise to subtly different dependency structures, we implemented three different kernels (**edge-detect**, **emboss**, **gaussian**, 31 lines of code (LOC) each). Second, we developed a full-featured graphics library based on SVG that demonstrates the performance on bespoke visualisation code (**grouped-bar-chart**, 140 LOC, **line-chart**, 143 LOC, **stacked-bar-chart**, 136 LOC). Finally, we tested two examples that use a D3.js front end for visualisation (**stacked-bar-scatter-plot**, 26 LOC, and **bar-chart-line-chart**, 38 LOC), the same front end used in Figures 2 and 4. The code for our benchmarks is given in the supplementary material.

In the experiments, each benchmark is run 10 times, and we report the mean runtime and standard deviation (in parentheses, coloured grey). Runtimes are reported in milliseconds, to 1 decimal place. When we refer to *speedup* or *slowdown* in reference to a pair of implementations, we mean the ratio of average runtimes for that benchmark.

Q1: Overhead of graph construction vs. trace construction Table 1 compares the average time taken to evaluate a program in the graph approach

(**G-Eval**) with the time taken in the trace-based approach (**T-Eval**). The ratio of graph-based to trace-based time is shown in the third column (**Eval-Slowdown**). Since building the graph involves maintaining a heap of allocated vertices, and we also build the inverse graph at the same time, we expect a larger overhead for the graph approach. As expected, we do see a higher overhead, being from 2.5x to 15x slower than their trace-based counterparts. For both approaches, program evaluation always stays within the limit to keep a user's attention on the system (10000ms).

Table 1: Average Evaluation Time for Traces vs. Graphs (ms, 1 decimal place)

	T-Eval	**G-Eval**	**Eval-Slowdown**
edge-detect	732.4 (±44.7)	2150.1 (±109.9)	2.94
emboss	633.4 (±42.3)	1718.5 (±30.9)	2.71
gaussian	619.9 (±48.7)	1714.3 (±42.5)	2.77
bar-chart-line-chart	2336.9 (±51.9)	5775.7 (±131.6)	2.47
stacked-bar-scatter-plot	1212.9 (±46.2)	7548.9 (±476.4)	6.22
grouped-bar-chart	89.3 (±8.4)	1341.6 (±132.4)	15.03
line-chart	130.0 (±10.5)	1131.0 (±14.4)	8.70
stacked-bar-chart	55.9 (±5.6)	783.0 (±15.4)	14.00

Q2: Graph-based demands vs. trace-based demands In Table 2, columns **T-Demands** and **G-Demands** show the average times taken to compute *demands* with the trace and graph approaches, respectively. The ratio of graph-based to trace-based performance is in the last column, **Bwd-Speedup**. We compute *demands* over the graph by running the algorithm for *demBy* from Figure 10a over the opposite graph. In these examples, we keep our query selections small, generally only selecting one or two output nodes.

Overall, we observe a significant speedup in the performance of *demands* using the graph approach, from 33x speedup for the **stacked-bar-chart**, to more than 80x for **gaussian**. In particular, every graph-based *demands* query is within the "instantaneous" category, whilst all but two of the examples using the trace approach are in the "noticable delay" category. We attribute this speedup to a couple of factors. First, in the trace approach, each backwards query involves rewinding the entire execution, regardless of whether all of the trace is relevant; graph queries only traverse the relevant parts of the graph. Second, in the trace approach, backwards evaluation makes frequent use of lattice join ($\vee$) to combine slicing information from different branches of the computation; joining closure slices in particular has the potential to be expensive because closures contain environments (which in turn contain closures, and so on). The graph approach lends itself to a more imperative implementation style, where demand information accrues against vertices as the backwards analysis proceeds, with no separate join steps.

Table 2: Trace-Based vs. Graph-Based Implementations of Demands (ms, 1 d.p.)

	T-Demands	G-Demands	Bwd-Speedup
edge-detect	61.0 (±7.9)	11.9 (±0.8)	50.84
emboss	504.3 (±9.1)	8.0 (±0.7)	63.28
gaussian	511.1 (±8.3)	6.3 (±0.8)	81.38
bar-chart-line-chart	1274.0 (±24.8)	18.5 (±0.6)	68.86
stacked-bar-scatter-plot	616.6 (±35.4)	14.0 (±1.7)	44.14
grouped-bar-chart	74.4 (±6.2)	1.3 (±0.3)	55.96
line-chart	114.1 (±4.8)	1.8 (±0.2)	65.22
stacked-bar-chart	45.1 (±5.4)	1.3 (±0.4)	33.62

Q3: Demanded By Table 3 contrasts average times for the trace implementation of *demanded by* (**T-DemBy**) with the graph-based algorithm from Figure 10a (**G-DemBy**) and the graph-based approach using using the De Morgan dual of the suff algorithm from Figure 10b (**G-DemBy-Suff**). **S** is the speedup of **G-DemBy** compared to **T-DemBy**, and **S'** is the speedup of **G-DemBy-Suff** compared to **T-DemBy**. Again, because our selections are small, the dual of suff, which traverses a "complement" subgraph, potentially explores a much larger subgraph than demBy. For all benchmarks, the graph demBy algorithm in Figure 10a performs better than the other two approaches, with five benchmarks running instantaneously, as opposed to only two for the trace-based approach.

Table 3: Trace-Based vs. Graph-Based Implementations of Demanded By (ms, 1 d.p.) where S = *T-DemBy*/*G-DemBy* and S' = *T-DemBy*/*G-DemBy-Suff*

	T-DemBy	G-DemBy	S	G-DemBy-Suff	S'
edge-detect	696.9 (±14.9)	265.8 (±19.1)	2.62	664.3 (±36.3)	1.05
emboss	603.0 (±18.0)	244.9 (±16.3)	2.46	481.5 (±16.4)	1.25
gaussian	596.8 (±11.6)	224.3 (±17.4)	2.66	474.0 (±12.0)	1.26
bar-chart-line-chart	2079.0 (±97.2)	19.1 (±1.2)	108.73	2360.2 (±48.0)	0.88
stacked-bar-scatter-plot	1004.1 (±72.7)	31.3 (±1.3)	32.07	11598.9 (±778.8)	0.09
grouped-bar-chart	82.4 (±9.4)	5.0 (±0.6)	16.47	2146.5 (±158.0)	0.04
line-chart	113.4 (±4.9)	2.3 (±0.5)	50.41	900.3 (±15.3)	0.13
stacked-bar-chart	42.6 (±1.9)	3.2 (±1.0)	13.18	1271.1 (±131.7)	0.03

Discussion When comparing the graph and trace approaches, the main shortcoming of the graph approach is the increased cost of evaluating a program to a graph versus a trace (*Q1*). We propose that this overhead is worth paying; the graph is only constructed once, and evaluation time still remains within the limit for keeping the user's attention. More importantly, the fast response times for *demands* and *demanded by* compared to the trace approach (Q2 and Q3) are good enough for instantaneous queries. Also notable (for Q3), is the performance

of **G-DemBy-Suff**, which uses the dual of the suff algorithm, and which tends to perform significantly worse than **G-DemBy**. This can perhaps be explained by the fact that our selections are small: since the selection is complemented, and the portion of the graph that the algorithm traverses is in general much larger. It is also plausible that some of this disparity can be explained by suff requiring more bookkeeping than demBy.

6 Related work

6.1 Data Provenance

Provenance research has a long history, ranging from scientific workflow provenance [9,12] to more fine-grained database techniques like *where* provenance [8], with comparatively less emphasis on general-purpose languages. Dietrich et al. [14] track both value and control dependencies in recursive database queries and user-defined functions by rewriting SQL queries to provenance-tracking counterparts; Fehrenbach and Cheney [16] explore provenance for language-integrated query, extending the SQL fragment of the multi-tier language Links [11] with provenance tracking. For structured outputs like visualisations, the most closely related lines of work are Psallidas and Wu [32] for relational languages, and Perera et al. [30] for general-purpose languages. These authors all emphasise the importance of *cognacy*, i.e. common ancestry, albeit not in the context of dependence graphs, showing how to "link" outputs by tracing back from an output selection to a data selection and then forward to another output selection. This seems to be largely unstudied in the provenance literature, despite its importance in data visualisation.

6.2 Dynamic Dependence Graphs

Dynamic dependence graphs have been used extensively for program slicing [18,19], optimisation [17], incremental computation [1] and fault localisation [37]. Most of these applications are for imperative languages, where it is useful distinguish between control and data edges; because Fluid is pure, we use somewhat simpler dependence graphs with a single kind of edge. However different kinds of edge are likely to be needed for future applications (§ 7). The main contribution of our work to the dependence graph paradigm are cognacy operators which allow the identification of minimally related sets of inputs ($\nabla\triangle$) and minimally related sets of outputs ($\triangle\nabla$), in a formal setting where we show these operators to be self-conjugate and related to Galois connections.

6.3 Galois Slicing

Although we do not consider program slices here, our formal setting is closely related to bidirectional program slicing techniques based on Galois connections [28,29,33]. In these works, the forward and backward directions of the interpreter

correspond to our *sufficiency* (▲) and *demands* (▽) queries. Perera et al. [30] extended this approach to a setting where slices have complements, and showed how composing the backward analysis with the De Morgan dual of the forwards analysis can be used to implement linked brushing. As well as requiring a bidirectional interpreter, the main difficulty with this approach has been achieving interactive performance. Implementations have relied on a sequential execution trace, used in the forwards direction to "replay" the computation, propagating sufficiency information from inputs to outputs, and in the backwards direction to "rewind" the computation, propagating demand from outputs to inputs. As discussed in § 5, and in contrast to the graph approach presented here, this approach is overly sequential and unable to exploit the independence of some subcomputations that makes slicing useful in the first place.

6.4 Linked Visualisations

The connection between visual selection (such as clicking or lassoing) and data "selection" has been explored in the visualisation literature, leading to various techniques for inverting selections in visual space to obtain selections in the underlying data [27,20]. There is also a literature on linked brushing [4,23] as a way of connecting output selections via mediating data selections. Although this has long been recognised as an important visual tool, with Roberts and Wright [34] arguing it should be ubiquitous, there is relatively little work on general-purpose infrastructure to support it. For example, Reactive Vega [35] provides a rich set of interactivity features to support linked brushing, but no mechanism for automatically propagating selections through arbitrary intervening queries; Glue [3] is a powerful Python library for building linked visualisations for scientific applications, but the developer is responsible for specifying the relationships between data sets that unpin the linking. More infrastructural approaches to linked brushing [27,23,32], similar in spirit to our work, have been developed mainly for relational languages. Relational languages are a promising direction in data visualisation, but general-purpose languages continue to be widely used too, so it is important to support linked brushing in this context as well.

6.5 Transparent Research Outputs

Dragicevic et al. [15] also develop techniques for transparent research outputs, via explorable multiverse analysis, exposing *analysis parameters*, the methodological choices made in designing a data analysis. Since different methodological choices will impact the conclusions of the analysis, they expose these choices to users and allow them to switch between different choices and observe the impact on the results. Our work is potentially complementary: we are concerned with surfacing the dependencies that arise within a specific choice of analysis parameters, whereas multiverse analyses are about exploring how things change under different choices. It might be useful to package our dependency analysis as part of such a multiverse analysis, allowing the user to observe the different dependency structures that arise from different methodological decisions.

7 Conclusion and Future Work

Visualisations are an essential tool for communicating science and other data-driven claims, but can be hard to make sense of and trust. Visualisations and other outputs that are "transparent" – that can reveal to an interested reader how they are related to data – are more informative and trustworthy but are also more difficult to produce. In this paper, we introduced a novel bidirectional program analysis framework for a general-purpose programming language that makes it easier to create transparent visualisations by shifting much of the burden to the language runtime. Relative to prior work based on execution traces and bidirectional interpreters, our approach also makes life easier for the language implementor, by decomposing the system into a single evaluator that builds a dependence graph, and a pair of language-agnostic bidirectional graph algorithms. We also showed an overall performance improvement for provenance queries, at the cost of some overhead in building the dependence graph.

We identify two important directions for future work. First, the dependency relation captured by the dependence graph semantics in § 3 omits certain intuitively plausible edges, such as the *projection* edge that one might expect to exist between a record and the value of a contained field as a consequence of evaluating a field access expression $e.x$. On the other hand, recording *all* structural dependencies of this nature would significantly bloat the dependence graph with information that is only relevant in certain contexts (for example, when one is specifically concerned with where a value came from, rather than how it was computed). We would like to develop a more semantically justified dependence graph, along with a proof that the graph is "complete" in some formal sense (cf. the *dependency correctness* notion of Cheney et al. [10]), but anticipate that making this richer graph practical will require distinguishing different kinds of query (e.g. "how" vs. "where from") for use in different contexts.

Second, although the algorithms presented in § 4 compute graph slices as an interim structure, the internal nodes of the graph slices are discarded. This is adequate for the use cases in § 2, which only concern "extensional" (IO) transparency. However, intensional information would be useful too: for example, highlighting a point in the moving average in Figure 2 could show not only that 3 emissions inputs were relevant, but that those values were summed and divided by 3. This would connect to work on *executable program slicing* [18]. One challenge here is managing the size and complexity of intensional explanations; again Cheney et al. [10] offer inspiration. Their idea of *expression provenance* may be a way of presenting pared-back but still informative explanations (for example, showing the tree of primitive arithmetic operations that computed a value but omitting user-defined function calls and branches). And the user may only want intensional information for certain subcomputations, in which case graph transformations which elide internal nodes but preserve IO connectivity, such as the Y-Δ transform [38], may also have a role to play.

Acknowledgments This research received support through Schmidt Sciences, LLC via the Institute of Computing for Climate Science (Perera and Orchard).

References

1. Acar, U.A., Blelloch, G.E., Harper, R.: Adaptive functional programming. In: POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 247–259. ACM Press, New York, NY, USA (2002). https://doi.org/10.1145/503272.503296
2. Agrawal, H., Horgan, J.R.: Dynamic program slicing. In: PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation. pp. 246–256. ACM, New York, NY, USA (1990). https://doi.org/10.1145/93542.93576
3. Beaumont, C., Robitaille, T., Goodman, A., Borkin, M.: Multidimensional data exploration with glue. In: van der Walt, S., Millman, J., Huff, K. (eds.) Proceedings of the 12th Python in Science Conference. pp. 8 – 12 (2013). https://doi.org/10.25080/Majora-8b375195-002
4. Becker, R.A., Cleveland, W.S.: Brushing scatterplots. Technometrics **29**(2), 127–142 (May 1987). https://doi.org/10.1080/00401706.1987.10488204
5. Bond, J., David, C., Nguyen, M., Orchard, M., Perera, R.: Fluid. https://zenodo.org/records/14637654 (2025)
6. Bremer, N., Ranzijn, M.: Urbanization in east asia between 2000 and 2010. http://nbremer.github.io/urbanization/ (2015)
7. Buja, A., McDonald, J.A., Michalak, J., Stuetzle, W.: Interactive data visualization using focusing and linking. In: Proceedings of Visualization '91. pp. 156–163 (Oct 1991). https://doi.org/10.1109/VISUAL.1991.175794
8. Buneman, P., Khanna, S., Tan, W.C.: Why and where: A characterization of data provenance. In: Proceedings of the 8th International Conference on Database Theory. pp. 316–330. ICDT '01, Springer-Verlag, London, UK (2001)
9. Callahan, S.P., Freire, J., Santos, E., Scheidegger, C.E., Silva, C.T., Vo, H.T.: VisTrails: Visualization meets data management. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 745–747 (2006). https://doi.org/10.1145/1142473.1142574
10. Cheney, J., Ahmed, A., Acar, U.A.: Provenance as dependency analysis. Mathematical Structures in Computer Science **21**(6), 1301–1337 (2011)
11. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: web programming without tiers. In: Proceedings of the 5th International Conference on Formal methods for Components and Objects. pp. 266–296. FMCO'06, Springer-Verlag, Berlin, Heidelberg (2007)
12. Davidson, S.B., Freire, J.: Provenance and scientific workflows: challenges and opportunities. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. pp. 1345–1350. SIGMOD '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1376616.1376772
13. DerSimonian, R., Laird, N.: Meta-analysis in clinical trials. Control Clin Trials **7**(3), 177–88 (Sep 1986). https://doi.org/10.1016/0197-2456(86)90046-2
14. Dietrich, B., Müller, T., Grust, T.: Data provenance for recursive sql queries. In: Proceedings of the 14th International Workshop on the Theory and Practice of Provenance. TaPP '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3530800.3534536, https://doi.org/10.1145/3530800.3534536
15. Dragicevic, P., Jansen, Y., Sarma, A., Kay, M., Chevalier, F.: Increasing the transparency of research papers with explorable multiverse analyses. In: CHI 2019 - The ACM CHI Conference on Human Factors in Computing Systems. Glasgow, United Kingdom (May 2019). https://doi.org/10.1145/3290605.3300295

16. Fehrenbach, S., Cheney, J.: Language-integrated provenance by trace analysis. In: Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages. p. 74–84. DBPL 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3315507.3330198, https://doi.org/10.1145/3315507.3330198
17. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (jul 1987). https://doi.org/10.1145/24039.24041
18. Field, J., Tip, F.: Dynamic dependence in term rewriting systems and its application to program slicing. Information and Software Technology **40**(11–12), 609–636 (November/December 1998)
19. Hammer, C., Grimme, M., Krinke, J.: Dynamic path conditions in dependence graphs. pp. 58–67 (01 2006). https://doi.org/10.1145/1111542.1111552
20. Heer, J., Agrawala, M., Willett, W.: Generalized selection via interactive query relaxation. In: ACM Human Factors in Computing Systems (CHI). pp. 959–968 (2008)
21. Hinze, R.: Generalizing generalized tries. Journal of Functional Programming **10**(4), 327–351 (2000). https://doi.org/10.1017/S0956796800003713
22. Jonsson, B., Tarski, A.: Boolean algebras with operators. part i. American Journal of Mathematics **73**(4), 891–939 (1951), http://www.jstor.org/stable/2372123
23. Livny, M., Ramakrishnan, R., Beyer, K., Chen, G., Donjerkovic, D., Lawande, S., Myllymaki, J., Wenger, K.: Devise: integrated querying and visual exploration of large datasets. SIGMOD Rec. **26**(2), 301–312 (jun 1997). https://doi.org/10.1145/253262.253335, https://doi.org/10.1145/253262.253335
24. Menni, M., Smith, C.: Modes of adjointness. Journal of Philosophical Logic **43**(2/3), 365–391 (2014), http://www.jstor.org/stable/24564097
25. Murphy, J., Sexton, D., Barnett, D., Jones, G., Webb, M., Collins, M., Stainforth, D.: Quantification of modelling uncertainties in a large ensemble of climate change simulations. Nature **430**, 768–72 (09 2004). https://doi.org/10.1038/nature02771
26. Nielsen, J.: Usability Engineering. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1994)
27. North, C., Shneiderman, B.: Snap-together visualization: a user interface for coordinating visualizations via relational schemata. In: Proceedings of the Working Conference on Advanced Visual Interfaces. p. 128–135. AVI '00, Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/345513.345282
28. Perera, R., Acar, U.A., Cheney, J., Levy, P.B.: Functional programs that explain their work. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. pp. 365–376. ICFP '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2364527.2364579
29. Perera, R., Garg, D., Cheney, J.: Causally consistent dynamic slicing. In: Desharnais, J., Jagadeesan, R. (eds.) Concurrency Theory, 27th International Conference, CONCUR '16. Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.18
30. Perera, R., Nguyen, M., Petricek, T., Wang, M.: Linked visualisations via galois dependencies. Proc. ACM Program. Lang. **6**(POPL) (2022). https://doi.org/10.1145/3498668
31. Peyton Jones, S., Eisenberg, R., Graf, S.: Triemaps that match. Tech. rep. (July 2022), https://simon.peytonjones.org/triemaps-that-match/

32. Psallidas, F., Wu, E.: Smoke: fine-grained lineage at interactive speed. Proc. VLDB Endow. **11**(6), 719–732 (feb 2018). https://doi.org/10.14778/3199517.3199522, https://doi.org/10.14778/3199517.3199522
33. Ricciotti, W., Stolarek, J., Perera, R., Cheney, J.: Imperative functional programs that explain their work. Proceedings of the ACM on Programming Languages **1**(ICFP), 14:1–14:28 (2017). https://doi.org/10.1145/3110258
34. Roberts, J.C., Wright, M.A.E.: Towards ubiquitous brushing for information visualization. In: Tenth International Conference on Information Visualisation (IV'06). pp. 151–156 (July 2006). https://doi.org/10.1109/IV.2006.113
35. Satyanarayan, A., Moritz, D., Wongsuphasawat, K., Heer, J.: Vega-Lite: A grammar of interactive graphics. IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis) (2017)
36. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data. p. 23–34. SIGMOD '79, Association for Computing Machinery, New York, NY, USA (1979). https://doi.org/10.1145/582095.582099, https://doi.org/10.1145/582095.582099
37. Soremekun, E., Kirschner, L., Böhme, M., Zeller, A.: Locating faults with program slicing: an empirical analysis. Empirical Software Engineering **26**(3), 51 (Apr 2021). https://doi.org/10.1007/s10664-020-09931-7
38. Truemper, K.: On the delta-wye reduction for planar graphs. Journal of Graph Theory **13**(2), 141–148 (1989). https://doi.org/https://doi.org/10.1002/jgt.3190130202