Linked Visualisations via Galois Dependencies

ROLY PERERA*, The Alan Turing Institute, UK
MINH NGUYEN, University of Bristol, UK
TOMAS PETRICEK†, University of Kent, UK
MENG WANG, University of Bristol, UK

We present new language-based dynamic analysis techniques for linking visualisations and other structured outputs to data in a fine-grained way, allowing users to explore how data attributes and visual or other output elements are related by selecting (focusing on) substructures of interest. Our approach builds on bidirectional program slicing techiques based on Galois connections, which provide desirable round-tripping properties. Unlike the prior work, our approach allows selections to be negated, equipping the bidirectional analysis with a De Morgan dual which can be used to link different outputs generated from the same input. This offers a principled language-based foundation for a popular view coordination feature called *brushing and linking* where selections in one chart automatically select corresponding elements in another related chart.

CCS Concepts: • **Theory of computation → Program semantics**.

Additional Key Words and Phrases: Galois connections; data provenance

ACM Reference Format:
Roly Perera, Minh Nguyen, Tomas Petricek, and Meng Wang. 2022. Linked Visualisations via Galois Dependencies. *Proc. ACM Program. Lang.* 6, POPL, Article 7 (January 2022), 29 pages. https://doi.org/10.1145/3498668

1 INTRODUCTION

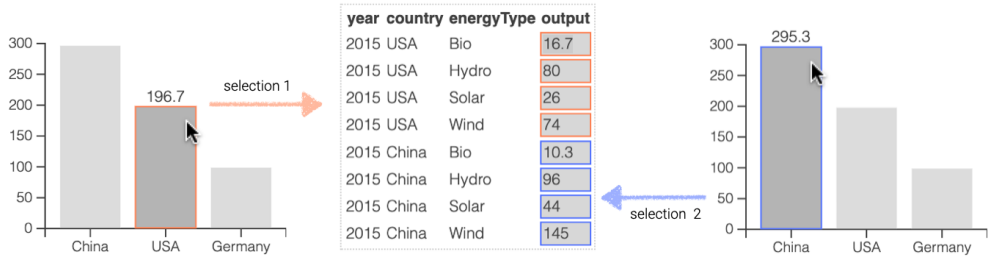
Techniques for dynamic dependency analysis have been fruitful, with applications ranging from information-flow security [Sabelfeld and Myers 2003] and optimisation [Kildall 1973] to debugging and program comprehension [De Lucia et al. 1996; Weiser 1981]. There are, however, few methods suitable for fine-grained analysis of richly structured outputs, such as data visualisations and multidimensional arrays. Dataflow analyses [Reps et al. 1995] tend to focus on analysing variables rather than parts of structured values. Where-provenance [Buneman et al. 2001] and related data provenance techniques are fine-grained, but are specific to relational query languages. Taint tracking [Newsome and Song 2005] is also fine-grained, but works forwards from input to output. For many applications, it would be useful to be able to focus on a particular part of a structured output, and have an analysis isolate the input data pertinent only to that substructure.

This is a need that increasingly arises outside of traditional programming. Journalists and data scientists use programs to compute charts and other visual summaries from data, charts which must be interpreted by colleagues, policy makers and lay readers alike. Interpreting a chart correctly

*Also with University of Bristol.
†Also with The Alan Turing Institute.

Authors' addresses: Roly Perera, The Alan Turing Institute, London, UK, rperera@turing.ac.uk; Minh Nguyen, min.nguyen@bristol.ac.uk, University of Bristol, Bristol, UK; Tomas Petricek, University of Kent, Canterbury, UK, tpetricek@kent.ac.uk; Meng Wang, meng.wang@bristol.ac.uk, University of Bristol, Bristol, UK.

2475-1421/2022/1-ART7
https://doi.org/10.1145/3498668




```
let totalFor c rows =
   sum [ row.output | row ⟵ rows, row.country == c ];
let data2015 = [ row | row ⟵ data, row.year == 2015 ];
    countryData = [ { x: c, y: totalFor c data2015 }
                  | c ⟵ ["China", "USA", "Germany"] ]
in BarChart { caption: "Total output by country", data: countryData }
```

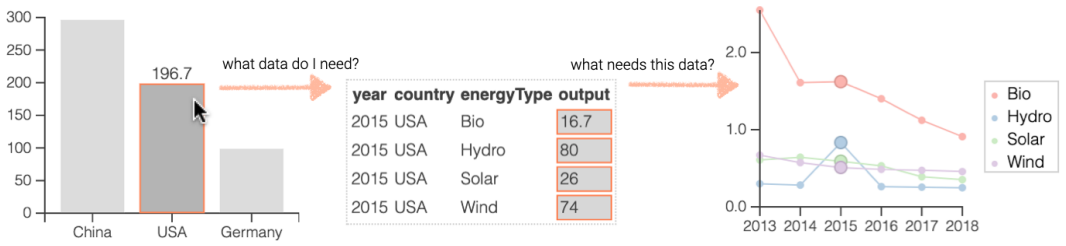
Fig. 1. Fine-grained linking of outputs to inputs, focusing on data for USA (left) and China (right).

means understanding what the components of the visualisation actually *represent*, i.e. the mapping between data and visual elements. But this is a hard task, requiring time and expertise, even with access to the data and source code used to create the visualisation. It is easy for innocent (but devastating) mistakes such as transposing two columns of data to go unnoticed [Miller 2006]. Since visualisations are simply cases of programs that transform structured inputs (data tables) into structured outputs (charts and other graphics), general-purpose language-based techiques for fine-grained dependency tracking should be able to help with this, by making it possible to reveal these relationships automatically to an interested user.

1.1 Linking Structured Outputs to Structured Inputs

First, interpreting a chart would be much easier if the user were able to explore the relationship between the various parts of the chart and the underlying data interactively, discovering the relevant relationships on a need-to-know basis. For example, selecting a particular bar in a bar chart could highlight the relevant data in a table, perhaps showing only the relevant rows, as illustrated in Figure 1. We could certainly do more and say something about the nature of the relationship (summation, in this case), but even just revealing the relevant data puts a reader in a much better position to fact-check or confirm their own understanding of what they are looking at. (The figure shows how selecting the bar for the USA should highlight different data than selecting the bar for China.) Indeed, this is useful enough that visualisation designers sometimes create "data-linked" artefacts like these by hand, such as Nadieh Bremer's award-winning visualisation of population density growth in Asian cities [Bremer and Ranzijn 2015], at the cost of significant programming effort. Libraries such as Altair [VanderPlas et al. 2018] alleviate some of this work, but require data transformations to be specified using a limited set of combinators provided (and understood) by the library.

What we would like to do is allow data scientists to author analyses and visualisations using an expressive functional language like the one shown in Figure 1, and obtain data linking automatically for the generated artefact, as a baked-in transparency feature. At the core of this is a program analysis problem: we want to be able to focus on a particular visual attribute — say the value of `y` in the record `{x: "USA", y: 196.7}` passed to `BarChart` in the example above — and perform some kind of backwards analysis to determine the relevant inputs, in this case the value of `output` in four




```
let series type country = [
   { x: year, y: row.output }
   | year ⟵ [2013..2018], row ⟵ data,
     row.year == year, row.energyType == type, row.country == country
] in LineChart {
   caption: "Output of USA relative to China",
   plots: [
      LinePlot { name: type, data: plot }
      | type ⟵ ["Bio", "Hydro", "Solar", "Wind"],
        let plot = zipWith (fun p1 p2 ⟶ { x: p1.x, y: p1.y / p2.y })
                           (series type "USA") (series type "China")
   ]
}
```

Fig. 2. Linking visualisations via common data dependencies

of the records that appear in the data source. Framing this as a program analysis problem not only provides a path to automation, but also invites interesting questions that a hand-crafted solution is unlikely to properly address. For example, does the union of two output selections depend on the union of their respective dependencies? Do dependencies "round-trip", in that they identify sufficient resources to reconstruct the selected output? Are they minimal? These questions are important to establishing trust, and a language-based approach offers a chance to address them.

1.2 Linking Structured Outputs to Other Structured Outputs

Second, authors often present distinct but related aspects of data in separate charts. In this situation a reader should be able to focus on (select) a visual element in one chart or other structured output and automatically see elements of a different chart which were computed using related inputs. For example in Figure 2 below, selecting the bar on the left should automatically highlight all the related visual elements on the right. This is a well-recognised use case called *brushing and linking* [Becker and Cleveland 1987], which is supported by geospatial applications like GeoDa [Anselin et al. 2006] and charting libraries like Plotly, but tends to be baked into specific views, or require programmer effort and therefore anticipation in advance by the chart designer. Moreover these applications and libraries provide no direct access to the common data which explains why elements are related.

Again, we would like to enable a more automated (and ubiquitous) version of brushing and linking, without imposing a burden on the programmer. They should be able to express visualisations and other data transformations using standard functional programming features such as those shown in Figure 2, and have brushing and linking enabled automatically between computed artefacts which depend on common data. At the core of this requirement is a variant of our original program analysis problem: we want to select part of the output and perform a backwards analysis to identify the required inputs, as before, but then also perform a forwards analysis to identify the dependent

parts of the other output. In Figure 2, these consist of the value of `y` for the record passed to `LinePlot` where `x` has the value `2015`, for each `LinePlot` in the list passed to `LineChart`. Moreover, we would also like the brushing and linking feature to be able to provide a concise view of the data that explain why the two selections are linked. Note, however, that the intuition behind the forwards analysis here is not the same as the one we appealed to in the context of round-tripping: there the (hypothetical) question was whether the selected data was *sufficient* to reconstruct the selected output, whereas to identify related items in another view, we must determine those parts for which the selected data is *necessary*. As before, a language-based approach offers the prospect of addressing these sorts of question in a robust way.

1.3 Contributions

To make progress towards these challenges, we present a bidirectional analysis which tracks fine-grained data dependencies between input and output selections, with round-tripping properties characterised by Galois connections. Selections have a complement, which we use to adapt the analysis to compute fine-grained dependencies between two outputs which depend on common inputs. Recent program slicing techniques [Perera et al. 2012, 2016; Ricciotti et al. 2017] allow the user to focus on the output by "erasing" parts deemed to be irrelevant; the erased parts, called *holes*, are propagated backwards by a backwards analysis which identifies parts of the program and input which are no longer needed. Although these approaches also enjoy useful round-tripping properties characterised by Galois connections, they only allow focusing on *prefixes* (the portion of the output or program that remains after the irrelevant parts have been erased), a notion which is not closed under complement. Our specific contributions are as follows:

- a new bidirectional dynamic dependency analysis which operates on selections of arbitrary parts of data values, for a core calculus with lists, records and mutual recursion, and a proof that the analysis is a Galois connection (§ 3);
- a second bidirectional dependency analysis, derived from the first by De Morgan duality, which is also a Galois connection and which can be composed with the first analysis to link outputs to outputs, with an extended example based on matrix convolution (§ 4);
- a richer surface language called Fluid[1], implemented in PureScript, with familiar functional programming features such as piecewise definitions and list comprehensions, and a further Galois connection linking selections between the core and surface languages (§ 5).

Proofs and other supplementary materials can be found at https://arxiv.org/abs/2109.00445.

2 CORE LANGUAGE

The core calculus which provides the setting for the rest of the paper is a mostly standard call-by-value functional language with datatypes and records. The main unusual feature is the use of *eliminators*, a trie-like construct that provides a uniform syntax and semantics for pattern-matching; this allows us to assume that incomplete or overlapping patterns and other syntactic considerations have been dealt in the surface language. (In § 5 we show how familiar pattern-matching features like case expressions and piecewise function definitions easily desugar into eliminators.) We give a big-step environment-based semantics, which is easier for the backward and forward dependency analyses in § 3, and introduce a compact (term-like) representation of derivation trees in the operational semantics, called *traces*, which we will use to define the analyses over a fixed execution. Mutual recursion requires some care for the backwards analysis, so we also treat that as a core language feature.

[1]See https://github.com/explorable-viz/fluid/. To generate the figures in this paper, check out tag `v0.4.2` and follow the instructions in artifact-evaluation.md.

	Type			***Continuation type***	
$A, B ::=$	`Bool`	Booleans	$K ::=$	A	term
	`Int`	integers		$A \rightarrowtail K$	eliminator
	`Rec` $(\overrightarrow{x\colon A})$	records		***Continuation***	
	`List` A	lists	$\kappa ::=$	e	term
	$A \to B$	functions		σ	eliminator
$\Gamma, \Delta ::=$	$\overrightarrow{x\colon A}$	typing context		***Eliminator***	
	Term		$\sigma, \tau ::=$	$x\colon \kappa$	variable
$e ::=$	`true` | `false`	Boolean		$\{\texttt{true}\colon \kappa, \texttt{false}\colon \kappa'\}$	Boolean
	n	integer		$\{(\overrightarrow{x})\colon \kappa\}$	record
	x	variable		$\{\texttt{[]}\colon \kappa, (\texttt{:})\colon \sigma\}$	list
	$\phi(\overrightarrow{e})$	primitive application		***Value***	
	$e\, e'$	application	$u, v ::=$	`true` | `false`	Boolean
	`[]` | u `:` v	list		n	integer
	$(\overrightarrow{x\colon e})$	record		`[]` | u `:` v	list
	$e.x$	record projection		$(\overrightarrow{x\colon v})$	record
	$\lambda\sigma$	anonymous function		$\texttt{cl}(\rho, h, \sigma)$	closure
	`let` h `in` e	recursive let	$\rho ::=$	$\overrightarrow{x\colon v}$	environment
$h ::=$	$\overrightarrow{x\colon \sigma}$	recursive functions			

Fig. 3. Syntax of core language

2.1 Syntax and Typing

Although our implementation is untyped, types help describe the structure of the core language. Figure 3 introduces the types A, B which include `Bool`, `Int` and function types $A \to B$, but also lists `List` A and records `Rec` $(\overrightarrow{x\colon A})$ which exemplify the two kinds of structured data which are of interest: recursive datatypes with varying structure, and tabular data with a fixed shape. As usual the notation $x\colon A$ denotes the binding of x to A (understood formally as a pair); $\overrightarrow{x\colon A}$ denotes the sequence of bindings that results from zipping same-length sequences $\overrightarrow{x}$ and $\overrightarrow{A}$. In a record type `Rec` $(\overrightarrow{x\colon A})$ the field names in $\overrightarrow{x}$ are required to be unique.

The terms e of the language are defined in Figure 3. These include Boolean constants `true` and `false`, integers n, variables x, and applications $e\, e'$. Primitives are not first-class; the expression $\phi(\overrightarrow{e})$ is the fully saturated application of ϕ to a sequence of arguments. (First-class and infix primitives are provided by desugarings in § 5). We also provide list constructors nil `[]` and cons e `:` e', record construction $(\overrightarrow{x\colon e})$ and record projection $e.x$. The final two term forms, anonymous functions $\lambda\sigma$ and recursive let-bindings `let` h `in` e where h is of the form $\overrightarrow{x\colon \sigma}$, are explained below after we introduce the pattern-matching construct σ (*eliminator*). The typing rules for terms are given in Figure 4, and are intended only to help a reader understand the language; therefore the rules are simple and do not include features such as polymorphism. The main typing rules of interest are the ones which involve eliminators.

2.2 Eliminators

Eliminators σ, τ are also defined in Figure 3, and are essentially generalised tries [Connelly and Morris 1995; Hinze 2000] extended with variable binding. An eliminator specifies how to match an initial part of a value and select a continuation κ for further execution; κ may be a term e, or another eliminator σ. The Boolean eliminator $\{\texttt{true}\colon \kappa, \texttt{false}\colon \kappa'\}$ selects either κ or κ' depending on whether a Boolean value is `true` or `false`. The record eliminator $\{(\overrightarrow{x})\colon \kappa\}$ matches a record with

$\boxed{\Gamma \vdash e : A}$ *e has type A under Γ*

$$\frac{}{\Gamma \vdash x : A}\ x : A \in \Gamma \qquad \frac{}{\Gamma \vdash n : \texttt{Int}} \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}}$$

$$\frac{\Gamma \vdash e_i : A_i \quad (\forall i \le |\vec{x}|)}{\Gamma \vdash (\overrightarrow{x\colon e}) : \texttt{Rec}\ (\overrightarrow{x\colon A})} \qquad \frac{\Gamma \vdash e : \texttt{Rec}\ (\overrightarrow{x\colon A})}{\Gamma \vdash e.x_i : A_i}\ i \le |\vec{x}| \qquad \frac{\Gamma \vdash \sigma : A \rightarrowtail B}{\Gamma \vdash \lambda\sigma : A \to B}$$

$$\frac{\Gamma \vdash e_i : \texttt{Int} \quad (\forall i \le j)}{\Gamma \vdash \phi(\vec{e}) : \texttt{Int}}\ \phi \in \mathbb{Z}^j \to \mathbb{Z} \qquad \frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A}{\Gamma \vdash e\,e' : B} \qquad \frac{}{\Gamma \vdash [] : \texttt{List}\ A}$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash e' : \texttt{List}\ A}{\Gamma \vdash (e : e') : \texttt{List}\ A} \qquad \frac{\Gamma \vdash h : \Delta \qquad \Gamma \cdot \Delta \vdash e : A}{\Gamma \vdash \texttt{let}\ h\ \texttt{in}\ e : A}$$

$\boxed{\Gamma \vdash \sigma : A \rightarrowtail K}$ *σ has type $A \rightarrowtail K$ under Γ*

$$\frac{\Gamma \cdot x\colon A \vdash \kappa : K}{\Gamma \vdash (x\colon \kappa) : A \rightarrowtail K} \qquad \frac{\Gamma \vdash \kappa : K \qquad \Gamma \vdash \kappa' : K}{\Gamma \vdash \{\texttt{true}\colon \kappa, \texttt{false}\colon \kappa'\} : \texttt{Bool} \rightarrowtail K} \qquad \frac{\Gamma \vdash \kappa : K}{\Gamma \vdash \{():\kappa\} : \texttt{Rec}\ () \rightarrowtail K}$$

$$\frac{\Gamma \vdash \{(\vec{x}):\sigma\} : \texttt{Rec}\ (\overrightarrow{x\colon A}) \rightarrowtail B \rightarrowtail K}{\Gamma \vdash \{(\vec{x} \cdot y):\sigma\} : \texttt{Rec}\ (\overrightarrow{x\colon A} \cdot y\colon B) \rightarrowtail K} \qquad \frac{\Gamma \vdash \kappa : K \qquad \Gamma \vdash \sigma : A \rightarrowtail \texttt{List}\ A \rightarrowtail K}{\Gamma \vdash \{[]\colon \kappa, (\texttt{:}):\sigma\} : \texttt{List}\ A \rightarrowtail B}$$

$\boxed{\vdash v : A}$ *v has type A*

$$\frac{}{\vdash n : \texttt{Int}} \qquad \frac{}{\vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\vdash \texttt{false} : \texttt{Bool}} \qquad \frac{\vdash v_i : A_i \quad (\forall i \le |\vec{x}|)}{\vdash (\overrightarrow{x\colon v}) : \texttt{Rec}\ (\overrightarrow{x\colon A})} \qquad \frac{}{\vdash [] : \texttt{List}\ A}$$

$$\frac{\vdash u : A \qquad \vdash v : \texttt{List}\ A}{\vdash (u : v) : \texttt{List}\ A} \qquad \frac{\vdash \rho : \Gamma \qquad \Gamma \vdash h : \Delta \qquad \Gamma \cdot \Delta \vdash \sigma : A \rightarrowtail B}{\vdash \texttt{cl}(\rho, h, \sigma) : A \to B}$$

$\boxed{\vdash \rho : \Gamma}$ *ρ has type Γ*

$$\frac{\vdash v_i : A_i \quad (\forall i \le |\vec{x}|)}{\vdash \overrightarrow{x\colon v} : \overrightarrow{x\colon A}}$$

$\boxed{\Gamma \vdash h : \Delta}$ *h has type Δ under Γ*

$$\frac{\Gamma \cdot \Delta \vdash \sigma_i : A_i \rightarrowtail B_i \quad (\forall i \le |\vec{x}|)}{\Gamma \vdash \overrightarrow{x\colon \sigma} : \Delta}\ \Delta = \overrightarrow{x\colon A \to B}$$

Fig. 4. Typing rules for core language

fields $\vec{x}$ and then selects κ with the variables $\vec{x}$ bound to the components of the record. The list eliminator $\{[]\colon \kappa, (\texttt{:})\colon \sigma\}$ selects κ if the list is empty and otherwise defers to another eliminator σ which specifies how the head and tail of the list are to be matched. Finally, the variable eliminator $x\colon \kappa$ extends the usual notion of trie, matching any value, and selecting κ with x bound to that value. Eliminators resemble the "case trees" commonly used as an intermediate form when compiling languages with pattern-matching [Graf et al. 2020], and can serve as an elaboration target for more user-oriented features such as the piecewise definitions described in § 5.

The use of nested eliminators to match sub-values will become clearer if we consider the typing judgement $\Gamma \vdash \sigma : A \rightarrowtail K$ given in Figure 4. Eliminators always have a function-like type; the judgement form should be read as a four-place relation, with $\rightarrowtail$ being part of the notation. (The definition delegates to an auxiliary judgement $\Gamma \vdash \kappa : K$ which we define to be the union of the $\Gamma \vdash e : A$ and $\Gamma \vdash \sigma : A \rightarrowtail K$ relations.) The typing rule for variable eliminators reveals

the connection between eliminators and functions: it converts a continuation κ which can be assigned type K under the assumption that x is of type A into an eliminator of type $A \rightarrowtail K$. The typing rule for Boolean eliminators says that to make an eliminator of type $\texttt{Bool} \rightarrowtail K$, we simply need continuations κ and κ' of type K. The rule for the empty record states that to make an eliminator of type $\texttt{Rec ()} \rightarrowtail K$, we simply need a continuation κ of type K. The rule for non-empty records allows us to transform a "curried" eliminator of type $\texttt{Rec } (\overrightarrow{x\colon A}) \rightarrowtail B \rightarrowtail K$ into one of type $\texttt{Rec } (\overrightarrow{x\colon A} \cdot y\colon B) \rightarrowtail K$, analogous to the isomorphism between $A \to B \to C$ and $A \times B \to C$ [Hinze 2000]. (Formalising eliminators precisely requires nested datatypes [Bird and Meertens 1998] and polymorphic recursion, but these details need not concern us here.)

The typing rule for list eliminators $\{\texttt{[]}\colon \kappa, (\texttt{:})\colon \sigma\}$ combines some of the flavour of record and Boolean eliminators. To make an eliminator of type $\texttt{List } A \rightarrowtail K$, we need a continuation of type K for the empty case, and for the non-empty case, an eliminator of type $A \rightarrowtail \texttt{List } A \rightarrowtail K$ which will be used to process the head and tail.

2.2.1 Functions as Eliminators. We can now revisit the term forms $\lambda\sigma$ and $\texttt{let } h \texttt{ in } e$. If σ is an eliminator of type $A \rightarrowtail B$, then $\lambda\sigma$ is an anonymous function of type $A \to B$. If h is of the form $\overrightarrow{x\colon \sigma}$, then $\texttt{let } h \texttt{ in } e$ introduces a sequence of mutually recursive functions which are in scope in e. The typing rule for $\texttt{let } h \texttt{ in } e$ uses an auxiliary typing judgement $\Gamma \vdash h : \Delta$ which assigns to every x in Δ the function type $A \to B$ if the σ to which x is bound in h has the eliminator type $A \rightarrowtail B$.

2.2.2 Values. Values v, u, and environments ρ are also defined in Figure 3, and are standard for call-by-value. (Environments are more convenient than substitution for tracking variable usage.) To support mutual recursion, the closure form $\texttt{cl}(\rho, h, \sigma)$ captures the (possibly empty) sequence h of functions with which the function was mutually defined, in addition to the ambient environment ρ. For the typing judgements $\vdash \rho : \Gamma$ and $\vdash v : A$ for environments and values (Figure 4), only the closure case is worth noting, which delegates to the typing rules for recursive definitions and eliminators.

2.2.3 Evaluation. Figure 6 gives the operational semantics of the core language. In § 3 we will define forward and backward analyses over a single execution; in anticipation of that use case, we treat the operational semantics as an inductive data type, following the "proved transitions" approach adopted by Boudol and Castellani [1989] for reversible CCS. The inhabitants of this data type are derivation trees explaining how a result was computed, and the analyses will be defined by structural recursion over these trees. Expressed in terms of inference rules, these trees can become quite cumbersome, so we introduce an equivalent but more term-like syntax for them, called a *trace* (Figure 5), similar to the approach taken by Perera et al. [2016] for π-calculus.

Trace

$T, U ::=$	x	variable
	$\texttt{true} \mid \texttt{false}$	Boolean
	n	integer
	$(\overrightarrow{x\colon T})$	record
	$T_{\overrightarrow{x\colon v}}.y$	record projection
	$\texttt{[]} \mid T : U$	list
	$\lambda\sigma$	anonymous function
	$T\ U \blacktriangleright w\colon T'$	application
	$\phi(\overrightarrow{U_n})$	primitive application
	$\texttt{let } h \texttt{ in } T$	recursive let

Match

$w ::=$	x	variable
	$\texttt{true} \mid \texttt{false}$	Boolean
	$(\overrightarrow{x\colon w})$	record
	$\texttt{[]} \mid w : w'$	list

Fig. 5. Syntax of traces and matches

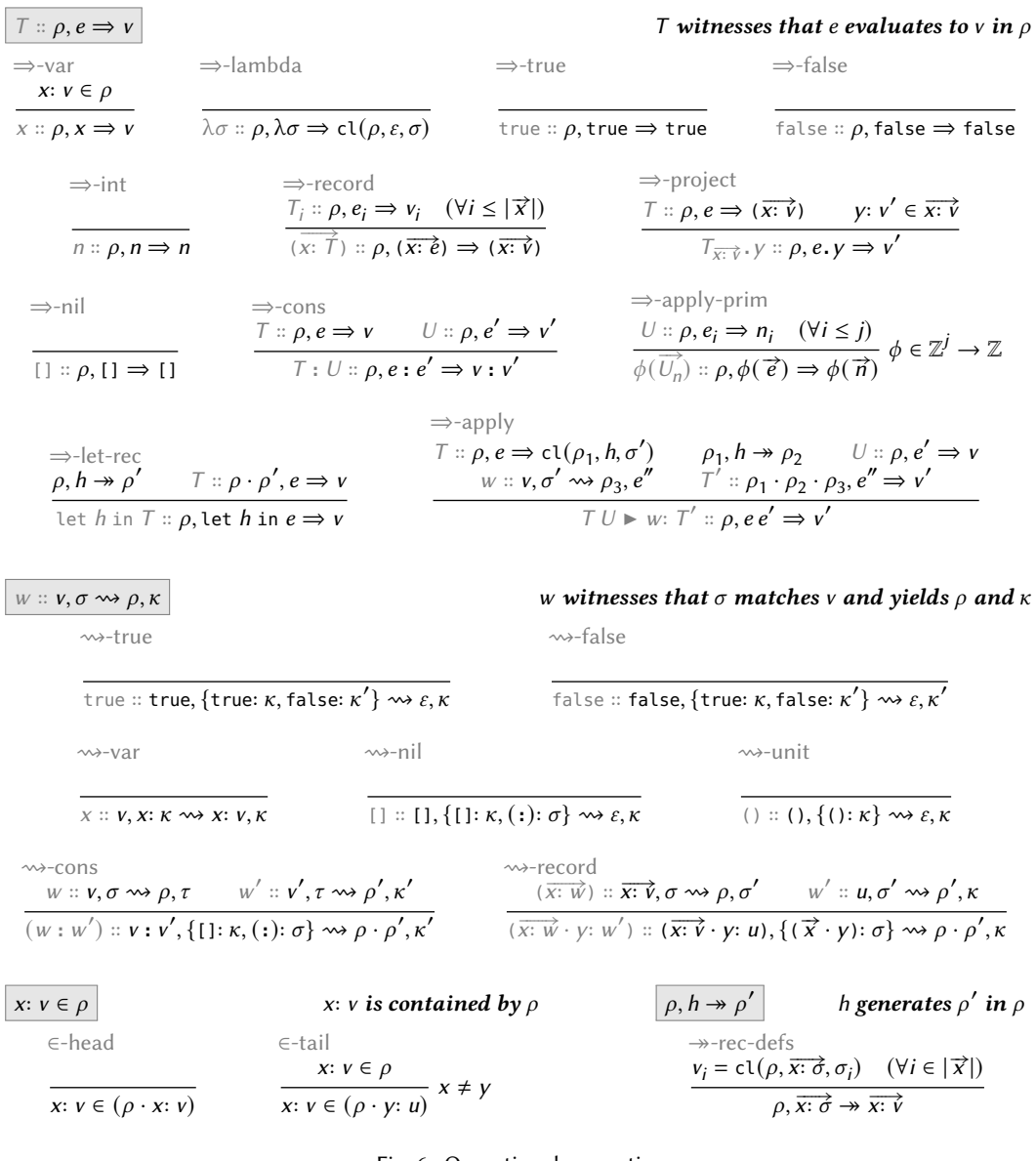


Fig. 6. Operational semantics

The judgement $T :: \rho, e \Rightarrow v$ defined at the top of Figure 6 states that term e under environment ρ evaluates to value v, and that T is a proof term that witness that fact. (In the figure, the traces appear in grey, to reinforce the idea that they are not part of the definition of $\Rightarrow$ but rather a notation for its inhabitants.) The rules for Booleans, integers and lists are standard and have unsurprising trace forms. For variables, we give an explicit inductive definition of the environment lookup relation $\in$ at the bottom of the figure, again so that later we can perform analysis over a proof that an environment contains a binding. The lambda rule is standard except that we specify ε for the sequence of definitions being simultaneously defined, since a lambda is not recursive. For record

construction, the trace form contains a subtrace T_i for each field, and for record projection, which also uses the lookup relation $\in$, the trace form $T_{\overrightarrow{x\colon v}}.y$ records both the record $\overrightarrow{x\colon v}$ and the field name y that was selected.

The rule for (mutually) recursive functions `let` h `in` e, where h is a sequence $\overrightarrow{x\colon \sigma}$ of function definitions, makes use of the auxiliary relation $\rho, h \twoheadrightarrow \rho'$ at the bottom of Figure 6 which turns h into an environment ρ' binding each function name x_i to a closure $\mathtt{cl}(\rho, h, \sigma_i)$ capturing ρ and a copy of h. For primitive applications, the trace records the values of the arguments which were passed to the operation ϕ. The rule for application $e\,e'$ is slightly non-standard, because it must deal with both mutual recursion and pattern-matching. First we unpack the recursive definitions h from the closure $\mathtt{cl}(\rho_1, h, \sigma)$ computed by e, and again use the auxiliary relation $\twoheadrightarrow$ to promote this into an environment ρ_2 of closures. We then use the relation $\rightsquigarrow$ explained below to match v against the eliminator σ, obtaining the branch e'' of the function to be executed and parameter bindings ρ_3. In addition to subtraces T and U for the function and argument, the application trace $T\ U \blacktriangleright w\colon T'$ also has subtraces w for the pattern-match and T' for the selected branch.

2.2.4 Pattern Matching. The judgement $w :: v, \sigma \rightsquigarrow \rho, \kappa$ also defined in Figure 6 states that eliminator σ can match v and produce environment ρ and continuation κ, with ρ containing the variable bindings that arose during the match. *Matches* w are a compact notation for proof terms for the $\rightsquigarrow$ relation, analogous to traces for the $\Rightarrow$ relation, and again appear in grey in the figure.

Variable eliminators $x\colon \kappa$ match any value, returning the singleton environment $x\colon v$ and continuation κ. Boolean eliminators match any Boolean value, returning the appropriate branch and empty environment ε. List eliminators $\{\mathtt{[]}\colon \kappa, (\mathtt{:})\colon \sigma\}$ match any list. The nil case is analogous to the handling of Booleans; the cons case depends on the fact that the nested eliminator σ for the cons branch has the curried type $A \rightarrowtail \mathtt{List}\ A \to K$. First, we recursively match the head v of type A using σ, obtaining bindings ρ and eliminator $\tau : \mathtt{List}\ A \rightarrowtail K$ as the continuation. Then the tail v' is matched using τ to yield additional bindings ρ' and final continuation κ' of type K. As a simple example, which omits the proof terms w, consider the following pattern-match:

$$\rightsquigarrow\text{-cons}\ \dfrac{\rightsquigarrow\text{-var}\ \dfrac{}{5, \mathtt{x}\colon \mathtt{xs}\colon e_2 \rightsquigarrow \mathtt{x}\colon 5, \mathtt{xs}\colon e_2} \qquad \rightsquigarrow\text{-var}\ \dfrac{}{\mathtt{6 : []}, \mathtt{xs}\colon e_2 \rightsquigarrow \mathtt{xs}\colon (\mathtt{6 : []}), e_2}}{\mathtt{5 : 6 : []}, \{\mathtt{[]}\colon e_1, (\mathtt{:})\colon \mathtt{x}\colon \mathtt{xs}\colon e_2\} \rightsquigarrow (\mathtt{x}\colon 5) \cdot (\mathtt{xs}\colon 6), e_2}$$

Here the eliminator $\{\mathtt{[]}\colon e_1, (\mathtt{:})\colon \mathtt{x}\colon \mathtt{xs}\colon e_2\}$ is used to match `5 : 6 : []`. The `[]` case is disregarded; the `(:)` case is used to retrieve a variable eliminator $\mathtt{x}\colon \mathtt{xs}\colon e_2$, which is used to match the head `5`. This produces the binding $\mathtt{x}\colon 5$ and a further variable eliminator $\mathtt{xs}\colon e_2$ as the continuation, which is used to match the tail. This produces the additional binding $\mathtt{xs}\colon (\mathtt{6 : []})$ and the expression e_2 as the continuation. To see how this might generalise to a nested pattern, consider how one could replace the inner variable eliminator $\mathtt{xs}\colon e_2$ by another list eliminator.

Record matching is similar: the empty record case resembles the nil case, and the non-empty case relies on the nested eliminator having curried type $\mathtt{Rec}\ (\overrightarrow{x\colon A}) \rightarrowtail B \rightarrowtail K$. The initial part $\overrightarrow{x\colon v}$ of the record is matched using σ, returning another eliminator σ' of type $B \rightarrowtail K$. Then the last field $y\colon u$ is matched using σ' to yield final continuation κ of type K.

3 A BIDIRECTIONAL DYNAMIC DEPENDENCY ANALYSIS

We now extend the core language from § 2 with a bidirectional mechanism for tracking data dependencies. § 3.1 establishes a way of selecting (parts of) values, such as the height of a bar in a bar chart. § 3.2 defines a forward analysis function $\nearrow_T$ which specifies how selections on programs and environments (collectively: *input selections*) become selections on outputs; selections represent *availability*, with the computed output selection indicating the data available to the downstream

computation. § 3.3 defines a backward dependency function $\searrow_T$ specifying how output selections are mapped back to inputs; then selections represent demands, with the computed input selection identifying the data needed from the upstream computation. Both functions are monotonic. This will become important in § 3.4, where we show that $\searrow_T$ and $\nearrow_T$ form a *Galois connection*, establishing the round-tripping properties alluded to in § 1.1.

3.1 Lattices of Selections

Our approach to representing selections is shown in Figure 7. The basic idea is to parameterise the type Val of values by a (bounded) lattice $\mathcal{A}$ of *selection states* α. We add selection states to Booleans, integers, records and lists; while it would present no complications to equip closures with selection states too, for present purposes we are only interested in dependencies between first-order data, so closures are not (directly) selectable. Closures do however have selectable parts, and moreover capture the current *argument availability*, explained in § 3.2.2 below, which is also a selection state α. We parameterise the type Term of terms similarly, allowing us to trace data dependencies back to expressions that appear in the source code, but only add selection states to the term constructors corresponding to selectable values. We return to this in § 5.

	Terms selections			***Value selections***	
$e \in \mathsf{Term}\,\mathcal{A} ::=$	...		$u, v \in \mathsf{Val}\,\mathcal{A} ::=$	$\square$	hole
	$\square$	hole		$\texttt{true}_\alpha \mid \texttt{false}_\alpha$	Boolean
	$\texttt{true}_\alpha \mid \texttt{false}_\alpha$	Boolean		n_α	integer
	n_α	integer		$(\overrightarrow{x\colon v})_\alpha$	record
	$(\overrightarrow{x\colon e})_\alpha$	record		$[]_\alpha \mid u :_\alpha v$	list
	$[]_\alpha \mid e :_\alpha e'$	list		$\mathtt{cl}(\rho, h, \alpha, \sigma)$	closure
$\alpha, \beta \in \mathcal{A}$		selection state			

Fig. 7. Selection states, term selections and value selections

The top and bottom elements $\top$ and $\bot$ of $\mathcal{A}$ represent fully selected and fully unselected; the meet and join operations $\sqcap$ and $\sqcup$, which have $\top$ and $\bot$ as their respective units, are used to combine selection information. In Figure 1, the `data` field of `BarChart` expects a list of records with fields `x` and `y`, mapping strings representing categorical data to floats determining the height of the corresponding bar; the record computed for China is $(\texttt{x}\colon \texttt{"China"} \cdot \texttt{y}\colon \texttt{295.3})$. The two-point lattice $2 \stackrel{\text{def}}{=} \langle \{\mathrm{tt}, \mathrm{ff}\}, \mathrm{tt}, \mathrm{ff}, \wedge, \vee \rangle$ can be used to represent the selection of the field `y` within this record as $(\texttt{x}\colon \texttt{"China"}_{\mathrm{ff}} \cdot \texttt{y}\colon \texttt{295.3}_{\mathrm{tt}})_{\mathrm{ff}}$, indicating that the number `295.3` is selected, but that neither the string `"China"`, nor the record itself, is selected. Because lattices are closed under component-wise products, we sometimes write $(\alpha, \beta) \sqsubseteq (\alpha', \beta')$ to mean that $\alpha \sqsubseteq \alpha'$ and $\beta \sqsubseteq \beta'$. This also suggests more interesting lattices of selections, such as vectors of Booleans to represent multiple selections simultaneously, which might be visualised using different colours (as in Figure 1).

3.1.1 Selections of a Value. The analyses which follow will be defined with respect to a fixed computation, and so we will often need to talk about the selections of a given value. To make this notion precise, consider that the raw (selection-free) syntax described in § 2 can be recovered from a term selection via an erasure operation $\lfloor \cdot \rfloor : \mathsf{Val}\,\mathcal{A} \to \mathsf{Val}\,1$ which forgets the selection information, where 1 is the trivial one-point lattice. We refer to $\lfloor v \rfloor$ as the *shape* of v. Allowing $\mathbf{u}, \mathbf{v}$ from now on to range over raw values, and reserving u, v for value selections, we can then define:

Definition 3.1 (Selections of $\mathbf{v}$*).* Define $\mathsf{Sel}_{\mathbf{v}}\,\mathcal{A}$ to be the set of all values $v \in \mathsf{Val}\,\mathcal{A}$ with shape $\mathbf{v}$, i.e. that erase to $\mathbf{v}$.

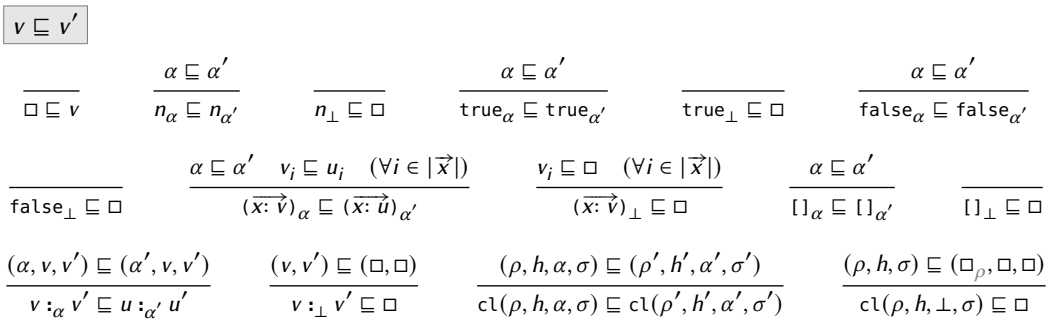


Fig. 8. Preorder on value selections

Since its elements have a fixed shape, the pointwise comparison of any $v, v' \in \mathrm{Sel}_{\mathbf{v}}\,\mathcal{A}$ using the partial order $\sqsubseteq$ of $\mathcal{A}$ is well defined, as is the pointwise application (zip) of a binary operation [Gibbons 2017]. It should therefore be clear that if $\mathcal{A}$ is a lattice, then $\mathrm{Sel}_{\mathbf{v}}\,\mathcal{A}$ is also a lattice, with $\top_{\mathbf{v}}$, $\perp_{\mathbf{v}}$, $\sqcap_{\mathbf{v}}$, and $\sqcup_{\mathbf{v}}$ defined pointwise. For example, if u and u' have the same shape and v and v' have the same shape, the join of the lists $(u :_\alpha v)$ and $(u' :_{\alpha'} v')$ is defined and equal to $(u \sqcup u') :_{\alpha \sqcup \alpha'} (v \sqcup v')$. Similarly, the top element of $\mathrm{Sel}_{\mathbf{v}}\,\mathcal{A}$ is the selection of $\mathbf{v}$ which has $\top$ at every selection position. (We omit the $\mathbf{v}$ indices from these lattice operations if it is clear which lattice is being referred to.) The notion of the "selections" of $\mathbf{v}$ extends to the other syntactic forms.

3.1.2 Environment Selections and Hole Equivalence. The notion of the "selections" of $\mathbf{v}$ also extends (pointwise) to environments, so that $\mathrm{Sel}_{\boldsymbol{\rho}}\,\mathcal{A}$ means the set of environment selections ρ' of shape $\boldsymbol{\rho}$, where the variables in ρ' are bound to selections of the corresponding variables in $\boldsymbol{\rho}$. One challenge arises from the pointwise use of $\sqcup$ to combine environment selections. Since environments contain other environments recursively, via closures, a naive implementation of environment join is a very expensive operation. One solution is to employ an efficient representation of values which are fully unselected, which is often the case during the backward analysis.

We therefore augment the set of value selections $\mathrm{Val}\,\mathcal{A}$ with a distinguished element *hole*, written $\square$, which is an alternative notation for $\perp_{\mathbf{v}}$ for any $\mathbf{v}$, i.e. the selection of shape $\mathbf{v}$ which has $\perp$ at every selection position, and generalise this idea to terms and eliminators. The equivalence of $\square$ to any such bottom element is established explicitly by the preorder order defined (for values) in Figure 8: the first rule always allows $\square$ on the left-hand side of $\sqsubseteq$, and other rules allow $\square$ on the right-hand side of $\sqsubseteq$ as long as all the selections that appear on the left-hand side are $\perp$. (The rules for terms e and eliminators σ are analogous and are omitted.) If we write $\doteq$ for the equivalence relation induced by $\sqsubseteq$ on values selections, which we call *hole-equivalence*, it should be clear that $\square \sqcup v \doteq v$ and $\square \sqcap v \doteq \square$. This means the join of two selections v, v' of $\mathbf{v}$ can be implemented efficiently, whenever one selection is $\square$, by simply discarding $\square$ and returning the other selection without further processing.

Definition 3.2 (Hole equivalence). Define $\doteq$ as the intersection of $\sqsubseteq$ and $\sqsupseteq$.

Because $\square$ is equivalent to $\perp_{\mathbf{v}}$ for any $\mathbf{v}$, all such bottom elements are hole-equivalent. For example, the value selection $\square :_\top \square$ is hole-equivalent to $5_\perp :_\top \square$, but also to $6_\perp :_\top [\,]_\perp$, and so the last two selections, even though they have different shapes, are hole-equivalent by transitivity. In practice we only use the hole ordering to compare selections with the same shape.

3.2 Forward Data Dependency

We now define the core bidirectional data dependency analyses for a fixed computation $T :: \boldsymbol{\rho}, \mathbf{e} \Rightarrow \mathbf{v}$, where T is a trace. In practice one would obtain T by first evaluating $\mathbf{e}$ in $\boldsymbol{\rho}$, and then run multiple forward or backward analyses over T with appropriate lattices. We start with the forward dependency function $\nearrow_T$ which "replays" evaluation, turning input availability into output availability, with T guiding the analysis whenever holes in the input selection would mean the analysis would otherwise get stuck. $\nearrow_T$ uses the auxiliary function $\pitchfork_w$ for forward-analysing a pattern-match; we explain this first, as it introduces the key idea of a selection as identifying the data available to a downstream computation.

3.2.1 Forward Match. Figure 9 defines a family of *forward-match* functions $\pitchfork_w$ of type $\mathrm{Sel}_{\mathbf{v},\sigma}\,\mathcal{A} \to (\mathrm{Sel}_{\boldsymbol{\rho},\boldsymbol{\kappa}}\,\mathcal{A}) \times \mathcal{A}$ for any $w :: \mathbf{v}, \sigma \rightsquigarrow \boldsymbol{\rho}, \boldsymbol{\kappa}$. (The definition is presented in a relational style for readability, but should be understood as a total function defined by structural recursion on w, which appears in grey to emphasise the connection to Figure 6.) Forward match replays the match witnessed by w, turning availability $(v, \sigma) \in \mathrm{Sel}_{\mathbf{v},\sigma}\,\mathcal{A}$ on the matched value and eliminator into availability $(\rho, \kappa) \in \mathrm{Sel}_{\boldsymbol{\rho},\boldsymbol{\kappa}}\,\mathcal{A}$ on the variable bindings and continuation yielded by the match.

$\pitchfork_w$ also returns the *meet* of the selection states associated with the part of $\mathbf{v}$ which was matched by $\boldsymbol{\sigma}$. We call this the *argument availability*, since it represents the availability of the matched part of a function argument. In the variable case, the empty part of $\mathbf{v}$ was matched and so the argument availability in this context is simply $\top$, the unit for $\sqcap$. In the Boolean case, the argument availability is simply the α on $\texttt{true}_\alpha$ or $\texttt{false}_\alpha$; the empty list and empty record cases are similar. In the cons case, we return the meet of the α on the cons node itself with the availabilities β and β' computed for v and v'. Non-empty records are similar, but to process the initial part of the record, we supply the neutral selection state $\top$ on the subrecord in order to use the definition recursively. (Note that these subrecords exist only as intermediate artefacts of the interpreter.)

One might hope to be able to dispense with the match witness w and simply define $\pitchfork$ by case analysis on v and σ. However, it is then unclear how to proceed in the event that either v or σ is a hole. In particular, it is not clear how to obtain the $\boldsymbol{\rho}$ associated with the original pattern-match in order to produce an environment selection $\rho' \in \mathrm{Sel}_{\boldsymbol{\rho}}\,\mathcal{A}$. If $\pitchfork$ is defined with respect to a known w, this can be achieved via additional rules $\pitchfork$-hole-v and $\pitchfork$-hole-σ that define the behaviour at hole to be the same as the behaviour at any $\doteq$-equivalent value in $\mathrm{Sel}_{\mathbf{v}}\,\mathcal{A}$ or $\mathrm{Sel}_{\sigma}\,\mathcal{A}$.

Operationally, these hole rules can be interpreted as "expanding" the holes in v or σ, in a shape-preserving way, until another rule of the definition applies. Recall the pattern-matching example from § 2.2.4. This pattern-match has the witness $\texttt{x : xs}$, recording that the list $\texttt{5 : 6 : []}$ was matched to the depth of a single cons. Suppose we wish to forward-analyse over the pattern-match using $\square$ to represent the selection on the matched list. The information in the match witness allows us to expand $\square$ to $\square :_\bot \square$ and then use the $\pitchfork$-cons rule to derive the following forward-match:

$$\pitchfork\text{-hole-}v\ \dfrac{\pitchfork\text{-cons}\ \dfrac{\pitchfork\text{-var}\ \dfrac{}{\square, \texttt{x: xs:}\ e_2\ \pitchfork_{\texttt{x}}\ \texttt{x:}\ \square, \texttt{xs:}\ e_2, \top} \qquad \pitchfork\text{-var}\ \dfrac{}{\square, \texttt{xs:}\ e_2\ \pitchfork_{\texttt{xs}}\ \texttt{xs:}\ \square, e_2, \top}}{\square :_\bot \square, \{\texttt{[]:}\ e_1, \texttt{(:): x: xs:}\ e_2\}\ \pitchfork_{\texttt{x : xs}}\ (\texttt{x:}\ \square) \cdot (\texttt{xs:}\ \square), e_2, \bot}}{\square, \{\texttt{[]:}\ e_1, \texttt{(:): x: xs:}\ e_2\}\ \pitchfork_{\texttt{x : xs}}\ (\texttt{x:}\ \square) \cdot (\texttt{xs:}\ \square), e_2, \bot}$$

Lemma 3.3 below implies that an implementation is free to replace any term by a hole-equivalent one of the same shape, with the result of $\pitchfork_w$ being unique up to $\doteq$. This justifies the strategy of expanding holes just enough for a non-hole rule to apply; there will be exactly one such rule, corresponding to the execution path originally taken, and although there may be multiple possible

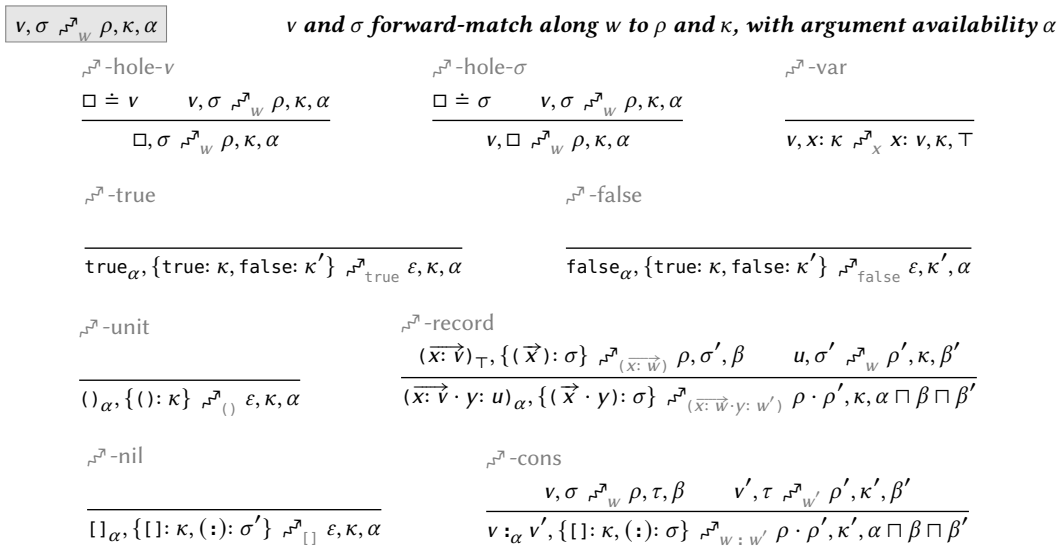


Fig. 9. Forward match

expansions, they will produce hole-equivalent results. This also explains why it is reasonable to think of $\nearrow_w$ not just as a relation, but as a function.

Lemma 3.3 (Monotonicity of $\nearrow_w$). *Suppose* $w :: \mathbf{v}, \boldsymbol{\sigma} \rightsquigarrow \boldsymbol{\rho}, \boldsymbol{\kappa}$, *with* $v, \sigma \nearrow_w \rho, \kappa, \alpha$ *and* $v', \sigma' \nearrow_w \rho', \kappa', \alpha'$. *If* $(v, \sigma) \sqsubseteq (v', \sigma)$ *then* $(\rho, \kappa, \alpha) \sqsubseteq (\rho', \kappa', \alpha')$.

The forward-match function $\nearrow_w$ is a key component of the forward evaluation function $\nearrow_T$ defined in § 3.2.2 below. When forward-analysing a function call, the argument is forward-matched using $\nearrow_w$, and the resulting argument availability α used to upper-bound the availability of any partial values constructed by that function, establishing a forward link from resources consumed and resources produced. Since the dynamic context of a function call extends over multiple evaluation steps, $\nearrow_T$ is threaded with an additional input α which tracks the active argument availability; at the outermost level, before there are any active function calls, this has the value $\top$.

3.2.2 Forward Evaluation. Figure 10 defines a family of *forward-evaluation* functions $\nearrow_T$ of type $(\mathsf{Sel}_{\boldsymbol{\rho},\mathbf{e}}\,\mathcal{A}) \times \mathcal{A} \to \mathsf{Sel}_{\mathbf{v}}\,\mathcal{A}$ for any $T :: \boldsymbol{\rho}, \mathbf{e} \Rightarrow \mathbf{v}$. (Like forward match, forward evaluation is presented in a relational style, but should be read as a total function defined by structural recursion on T.) Forward evaluation replays T, taking a selection $(\rho, e) \in \mathsf{Sel}_{\boldsymbol{\rho},\mathbf{e}}\,\mathcal{A}$ identifying the available parts of the environment and program, and an $\alpha \in \mathcal{A}$ representing the argument availability for the dynamically innermost function call, and returning a selection $v \in \mathsf{Sel}_{\mathbf{v}}\,\mathcal{A}$ identifying the outputs that can be produced using only the available resources. The rules resemble those for the evaluation relation $\Rightarrow$. The general pattern is that each rule takes the active argument availability α, combines it (using $\sqcap$) with any availability supplied on the expression form consumed at that step, and uses the result as the availability of any partial values constructed at that step. The argument availability α is passed down unchanged to any subcomputations, except in the case of function application.

Function application. In the application case, the rule must determine a new argument availability for the function body, because the function context is changing. First, we unpacks the β stored in the closure, representing the argument availability which was active when the closure was constructed. Then we determine an additional selection state β', representing the availability of

$\boxed{\rho, e, \alpha \nearrow_T v}$ *ρ and e, with argument availability α, forward-evaluate along T to v*

$$\frac{\square \doteq e \qquad \rho, e, \alpha \nearrow_T v}{\rho, \square, \alpha \nearrow_T v}\ \nearrow\text{-hole} \qquad \frac{x\colon v \in \rho}{\rho, x, \alpha \nearrow_x v}\ \nearrow\text{-var} \qquad \frac{}{\rho, \lambda\sigma, \alpha \nearrow_{\lambda\sigma'} \mathtt{cl}(\rho, \varepsilon, \alpha, \sigma)}\ \nearrow\text{-lambda}$$

$$\frac{}{\rho, \mathtt{true}_{\alpha'}, \alpha \nearrow_{\mathtt{true}} \mathtt{true}_{\alpha\sqcap\alpha'}}\ \nearrow\text{-true} \qquad \frac{}{\rho, \mathtt{false}_{\alpha'}, \alpha \nearrow_{\mathtt{false}} \mathtt{false}_{\alpha\sqcap\alpha'}}\ \nearrow\text{-false} \qquad \frac{}{\rho, n_{\alpha'}, \alpha \nearrow_n n_{\alpha\sqcap\alpha'}}\ \nearrow\text{-int}$$

$$\frac{\rho, e_i, \alpha \nearrow_{T_i} v_i \quad (\forall i \le |\overrightarrow{x}|)}{\rho, (\overrightarrow{x\colon e})_{\alpha'}, \alpha \nearrow_{(\overrightarrow{x\colon T})} (\overrightarrow{x\colon v})_{\alpha\sqcap\alpha'}}\ \nearrow\text{-record} \qquad \frac{\rho, e, \alpha \nearrow_T \doteq (\overrightarrow{x\colon u})_\beta \qquad y\colon v' \in \overrightarrow{x\colon u}}{\rho, e.y, \alpha \nearrow_{T_{\overrightarrow{x\colon v}.y}} v'}\ \nearrow\text{-project} \qquad \frac{}{\rho, []_{\alpha'}, \alpha \nearrow_{[]} []_{\alpha\sqcap\alpha'}}\ \nearrow\text{-nil}$$

$$\frac{\rho, e, \alpha \nearrow_T v \qquad \rho, e', \alpha \nearrow_U v'}{\rho, e :_{\alpha'} e', \alpha \nearrow_{T:U} v :_{\alpha\sqcap\alpha'} v'}\ \nearrow\text{-cons} \qquad \frac{\rho, e_i, \alpha \nearrow_{U_i} \doteq n_{i\beta_i} \quad (\forall i \le |\overrightarrow{n}|)}{\rho, \phi(\overrightarrow{e}), \alpha \nearrow_{\phi(\overrightarrow{U_n})} \phi(\overrightarrow{n})_{\alpha'}}\ \phi_{\overrightarrow{n}*}(\overrightarrow{\beta}) = \alpha'\ \nearrow\text{-apply-prim}$$

$$\frac{\rho, e, \alpha \nearrow_T \doteq \mathtt{cl}(\rho_1, h, \beta, \sigma) \qquad \rho_1, h, \beta \nearrow \rho_2 \qquad \rho, e', \alpha \nearrow_U v \qquad v, \sigma \overset{\nearrow}{\sim}_w \rho_3, e'', \beta' \qquad \rho_1 \cdot \rho_2 \cdot \rho_3, e'', \beta \sqcap \beta' \nearrow_{T'} v'}{\rho, e\, e', \alpha \nearrow_{T\, U \blacktriangleright w\colon T'} v'}\ \nearrow\text{-apply} \qquad \frac{\rho, h', \alpha \nearrow \rho' \qquad \rho \cdot \rho', e, \alpha \nearrow_T v}{\rho, \mathtt{let}\ h'\ \mathtt{in}\ e, \alpha \nearrow_{\mathtt{let}\ h\ \mathtt{in}\ T} v}\ \nearrow\text{-let-rec}$$

$\boxed{\rho, h, \alpha \nearrow \rho'}$ *h forward-generates to ρ' in ρ and α*

$$\frac{v_i = \mathtt{cl}(\rho, \overrightarrow{x\colon \sigma}, \alpha, \sigma_i) \quad (\forall i \in |\overrightarrow{x}|)}{\rho, \overrightarrow{x\colon \sigma}, \alpha \nearrow \overrightarrow{x\colon v}}\ \nearrow\text{-rec-defs}$$

Fig. 10. Forward evaluation

the matched part of the current argument, by forward-matching v with the eliminator σ from the closure. These are combined using $\sqcap$ to represent the conjoined availability of all arguments that were pattern-matched in order to execute the function body, and the result $\beta \sqcap \beta'$ used to forward-evaluate the function body. The auxiliary function $\nearrow_{\rho,h}\colon (\mathrm{Sel}_{\boldsymbol{\rho},\mathbf{h}}\,\mathcal{A}) \times \mathcal{A} \to \mathrm{Sel}_{\boldsymbol{\rho}'}\,\mathcal{A}$ for any $\boldsymbol{\rho}, \mathbf{h} \twoheadrightarrow \boldsymbol{\rho}'$ is given at the bottom of Figure 10 and resembles $\twoheadrightarrow$, but captures the active argument availability into each closure.

Primitive application. Since primitive operations are opaque, their input-output dependencies cannot be derived from their execution, but must be supplied by the primitive operation itself. More specifically, every primitive $\phi \in \mathtt{Int}^i \to \mathtt{Int}$ is required to provide a forward-dependency function $\phi_{\overrightarrow{n}*} : \mathcal{A}^i \to \mathcal{A}$ for every $\overrightarrow{n} \in \mathtt{Int}^i$ which specifies how to turn an input selection $\overrightarrow{\alpha} \in \mathcal{A}^i$ for $\overrightarrow{n}$ into an output selection α' on $\phi(\overrightarrow{n})$. There is one such function per possible input $\overrightarrow{n}$ so that the dynamic dependencies for that specific call can depend on the values passed to the operation. For example, in our implementation, the dependency function for multiplication establishes (for non-zero n) that both $n * 0$ and $0 * n$ depend only on 0. However, primitives are free to implement forward-dependency however they choose, with the caveat that § 3.3.2 will also require ϕ to provide

a backward-dependency function for any input $\vec{n}$, and § 3.4 will require these to be related in a certain way for the consistency of the whole system to be guaranteed.

Other rules. The remaining rules follow the general pattern. Variable lookup disregards α, simply preserving the selection on the value extracted from the environment. The lambda rule captures α in the closure along with the environment; the letrec rule passes α on to $\nearrow$ so it can be captured by recursive closures as well. Record projection is more interesting, disregarding not only the argument availability α but also the availability β of the record itself. This is because containers are considered to be independent of the values they contain: here, v_i has its own internal availability which is preserved by projection, but there is no implied dependency of the field on the record from which it was projected. Record construction also reflects this principle, preserving the field selections unchanged into the resulting record selection. But since this rule also constructs a partial value — the record itself — it must specify an availability on that output. The availability is set to $\alpha \sqcap \alpha'$, reflecting the dependency of the constructed container on both the constructing expression and the active argument match. The rules for nil, cons, integers and Booleans are similar, since they also construct values.

Hole cases. Environments have no special $\Box$ form. However, a hole rule is needed to allow forward evaluation to continue in the event that e is $\Box$; this is essential because subsequent steps may result in non-$\Box$ outputs (for example by extracting non-$\Box$ values from ρ). The rule is similar to the hole rules for $\nearrow_w$ and again can be understood operationally as using the information in T to expand $\Box$ sufficiently for another rule to apply, with a result which is unique up to $\doteq$. In addition, application and record projection must accommodate the case where the selection on the closure or record being eliminated is represented by $\Box$. In these rules $\nearrow_T \doteq$ is used to denote the relational composition of $\nearrow_T$ and $\doteq$.

Lemma 3.4 (Monotonicity of $\nearrow_T$). *Suppose* $T :: \boldsymbol{\rho}, \mathbf{e} \Rightarrow \mathbf{v}$ *with* $\rho, e, \alpha \nearrow_T v$ *and* $\rho', e', \alpha' \nearrow_T v'$. *If* $(\rho, e, \alpha) \sqsubseteq (\rho', e', \alpha')$ *then* $v \sqsubseteq v'$.

3.3 Backward Data Dependency

The backward dependency function $\searrow_T$ "rewinds" evaluation, turning output demand into input demand, with T guiding the analysis backward. We start with the auxiliary function $\searrow_w$ which is used for backward-analysing a pattern-match.

3.3.1 Backward Match. Figure 11 defines a family of *backward-match* functions $\searrow_w$ of type $(\mathrm{Sel}_{\boldsymbol{\rho},\boldsymbol{\kappa}}\, \mathcal{A}) \times \mathcal{A} \rightarrow \mathrm{Sel}_{\mathbf{v},\boldsymbol{\sigma}}\, \mathcal{A}$ for any $w :: \mathbf{v}, \boldsymbol{\sigma} \rightsquigarrow \boldsymbol{\rho}, \boldsymbol{\kappa}$. Backward-match rewinds the match witnessed by w, turning demand on the environment and continuation into demand on the value and eliminator that were originally matched. The additional input α represents the downstream demand placed on any resources that were constructed in the context of this match; $\searrow_w$ transfers this to the matched portion of $\mathbf{v}$, establishing a backwards link from resources produced to resources consumed in a given function context. We call α the *argument demand* since it represents the demand to be pushed backwards onto the matched part of a function argument.

In the variable case, the empty part of $\mathbf{v}$ was matched, so α is disregarded. The rule need only ensure that the demand v in the singleton environment $x\colon v$ is propagated backward. If a Boolean constant was matched, α becomes the demand on that constant, and κ, capturing the demand on the continuation, is used to construct the demand on the original eliminator, with $\Box$ used to represent the absence of demand on the non-taken branch. (This use of $\Box$ explains why matches w need only retain information about taken branches.) The nil case is similar.

For a cons match $w : w'$, we split the environment into ρ and ρ', using the fact that there is a unique well-typed decomposition. We then backward-match w and w' recursively to obtain v and v', representing the demand on the head and tail of the list. These are combined into the demand on

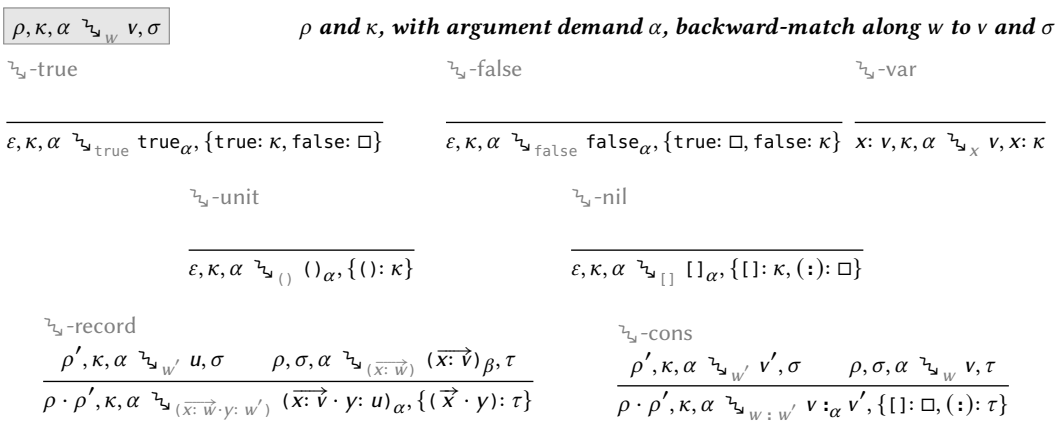


Fig. 11. Backward match

the entire list, using α as the demand on the root cons node. The eliminator selection σ represents the demand on the interim eliminator used to match the tail, and τ the demand on the eliminator used to match the head; these are then combined into a demand on the eliminator used to match the whole list, with $\square$ again used to represent the absence of demand on the nil branch. Records are similar, except that there is only a single branch. The selection state β computed for the initial part of the record is an artefact of processing records recursively, and is disregarded.

Lemma 3.5 (Monotonicity of $\searrow_w$). *Suppose* $w :: \mathbf{v}, \boldsymbol{\sigma} \rightsquigarrow \boldsymbol{\rho}, \boldsymbol{\kappa}$, *with* $\rho, \kappa, \alpha \searrow_w v, \sigma$ *and* $\rho', \kappa', \alpha' \searrow_w v', \sigma'$. *If* $(\rho, \kappa, \alpha) \sqsubseteq (\rho', \kappa', \alpha')$ *then* $(v, \sigma) \sqsubseteq (v', \sigma')$.

3.3.2 Backward Evaluation. Figure 12 defines a family of *backward-evaluation* functions $\searrow_T$ of type $\mathrm{Sel}_{\mathbf{v}}\,\mathcal{A} \to (\mathrm{Sel}_{\boldsymbol{\rho},\mathbf{e}}\,\mathcal{A}) \times \mathcal{A}$ for any $T :: \boldsymbol{\rho}, \mathbf{e} \Rightarrow \mathbf{v}$. Backward evaluation rewinds T, using the output selection $v \in \mathrm{Sel}_{\mathbf{v}}\,\mathcal{A}$ to determine an input selection $(\rho, e) \in \mathrm{Sel}_{\boldsymbol{\rho},\mathbf{e}}\,\mathcal{A}$ and an argument demand $\alpha \in \mathcal{A}$ which will eventually be pushed back onto the argument of the dynamically innermost function call. (At the outermost level, where there are no active function calls, the argument demand is discarded.) The rules resemble those of the evaluation relation $\Rightarrow$ with inputs and outputs flipped. The general pattern is that each backward rule takes the join of the demand attached to any partial values constructed at that step, and the argument demand associated with any subcomputations, and passes it upwards as the new argument demand. The output environment is constructed similarly, by joining the demand flowing back through the environment copies used to evaluate subcomputations. Demand is also attached to the source expression when it is the expression form responsible for the construction of a demanded value.

Function application. The application rule is where the argument demand is used and the function context changes, so we start here. The rule essentially runs the forward evaluation rule in reverse, using the trace T' to backward-evaluate the function body. The argument demand β associated with T' is the join of the demand on any resources constructed directly by that function invocation, and is transferred to the matched part of the function argument by the backward-match function $\searrow_w$. The argument demand passed upwards into the enclosing function context is $\alpha \sqcup \alpha'$, representing the resources needed along T and U. The auxiliary function $\searrow_{\rho,h} : \mathrm{Sel}_{\boldsymbol{\rho}'}\,\mathcal{A} \to (\mathrm{Sel}_{\boldsymbol{\rho},\mathbf{h}}\,\mathcal{A}) \times \mathcal{A}$ for any $\boldsymbol{\rho}, \mathbf{h} \twoheadrightarrow \boldsymbol{\rho}'$ defined at the bottom of Figure 12 is used to turn ρ_2, capturing the demand flowing back through any recursive uses of the function and any others with which it was mutually defined,

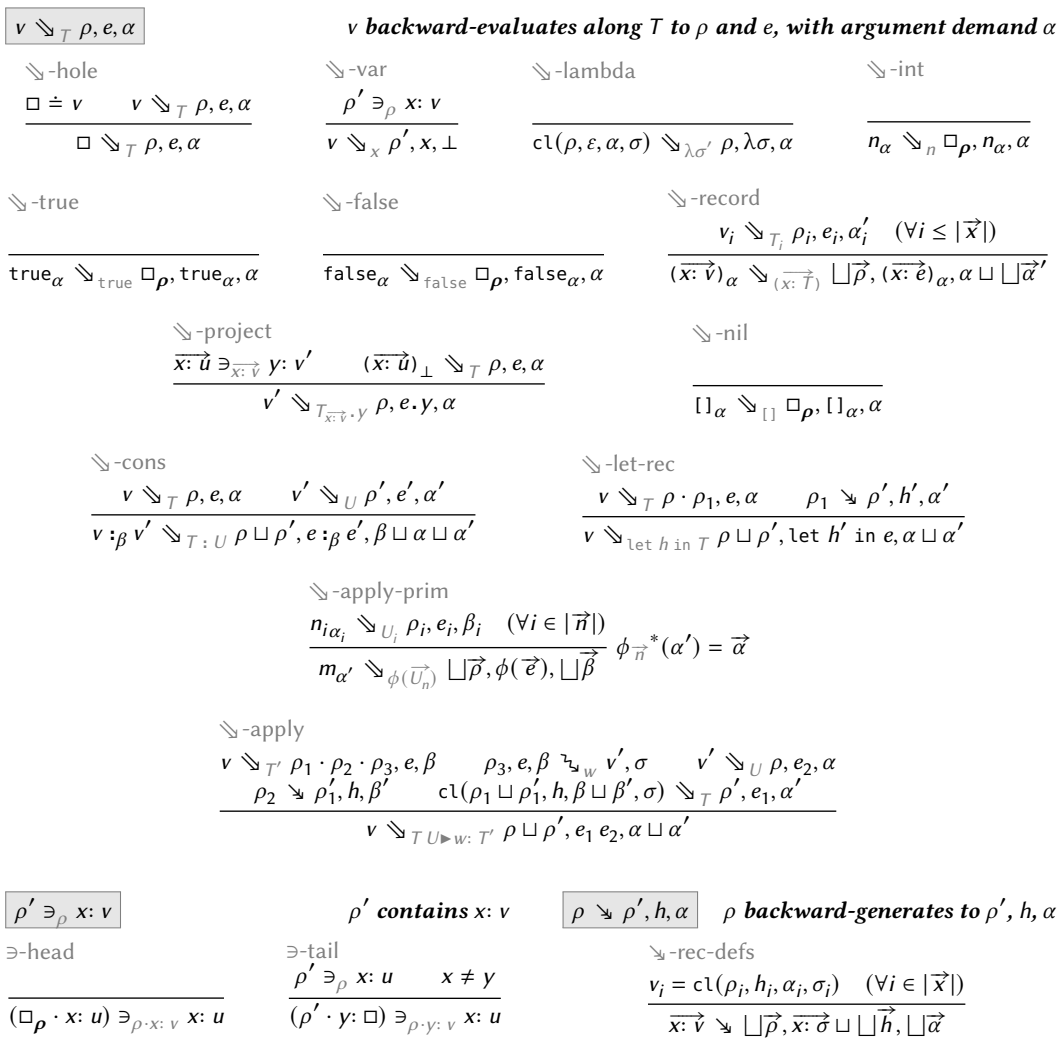


Fig. 12. Backward evaluation

into information that can be merged back into the demand on the closure. The function $\searrow_{\rho,h}$ is also used in the letrec rule, which otherwise follows the general pattern described above.

Primitive application. Each primitive operation $\phi : \texttt{Int}^i \to \texttt{Int}$ must provide a backward-dependency function $\phi_{\vec{n}}{}^* : \mathcal{A} \to \mathcal{A}^i$ for every $\vec{n} \in \texttt{Int}^i$ which specifies how to turn the output selection α' on $\phi(\vec{n})$ into an input selection $\vec{a} \in \mathcal{A}^i$ on $\vec{n}$. The rule for primitive application uses this information to pair each argument n_i with its demand α_i and then backwards-evaluate the argument. The argument demand passed upward is the join of those arising from these subcomputations, and is unrelated to the execution of the primitive itself, similar to a function application. Here $\bigsqcup\vec{\beta}$ means the fold of $\sqcup$ (with unit $\bot$) over the sequence of selection states $\beta_1 \cdot .. \cdot \beta'_{|\vec{x}|}$. Environment demands $\vec{\rho} = \rho_1 \cdot .. \cdot \rho_{|\vec{n}|}$ are joined (pointwise) in a similar fashion.

Other rules. In the variable case, no partial values were constructed during evaluation and there are no subcomputations, so the argument demand is $\bot$, the unit for $\sqcup$. The returned environment selection demands v for the variable x and $\square$ for all other variables, using the family of *backwards lookup* functions $- \ni_\rho x\colon -$ of type $\mathsf{Sel}_{\mathbf{v}}\,\mathcal{A} \to \mathsf{Sel}_{\boldsymbol{\rho}}\,\mathcal{A}$ for any $x\colon \mathbf{v} \in \boldsymbol{\rho}$ also defined in Figure 12. (The output of the function is on the left in the relational notation.) For atomic values such as integers, Booleans and nil, the argument demand is simply the demand α associated with the constructed value, which is also attached to the corresponding expression, and the environment demand has $\square$ for every variable in the original environment $\boldsymbol{\rho}$, written $\square_{\boldsymbol{\rho}}$.

For closures, the argument demand is unpacked along with the other components, preserving any internal selections on ρ and σ. Composite values such as records and cons cells follow the general pattern; thus for records, the argument demands α_i' from the subcomputations are joined with the α on the record itself to produce the argument demand passed upward. Record projection never demands the record constructor itself, but simply promotes the field demand into a record demand, using $\ni_{\overrightarrow{x\colon \mathbf{v}}}$ to demand fields other than y with $\square$.

Hole rule. The hole rule, as elsewhere, ensures that the function is defined when v is $\square$, and it is easy to show that $\searrow_T$ preserves $\sqsubseteq$, and thus $\doteq$.

Lemma 3.6 (Monotonicity of $\searrow_T$). *Suppose* $T :: \boldsymbol{\rho}, \mathbf{e} \Rightarrow \mathbf{v}$ *with* $v \searrow_T \rho, e, \alpha$ *and* $v' \searrow_T \rho', e', \alpha'$. *If* $v \sqsubseteq v'$ *then* $(\rho, e, \alpha) \sqsubseteq (\rho', e', \alpha')$.

3.4 Round-Tripping Properties of $\nearrow_T$ and $\searrow_T$

We now establish more formally the round-tripping properties, alluded at the beginning of the section, that relate $\nearrow_T$ to $\searrow_T$. For the analyses to be coherent, we expect $\nearrow_T(\searrow_T(v))$ to produce a value selection $v' \sqsupseteq v$, and $\searrow_T(\nearrow_T(\rho, e))$ to produce an input selection $(\rho', e') \sqsubseteq (\rho, e)$. Pairs of (monotonic) functions $f : X \to Y$ and $g : Y \to X$ that are related in this way are called *Galois connections*. Galois connections generalise isomorphisms: f and g are not quite mutual inverses, but are the nearest to an inverse each can get to the other. We will present a visual example of some of these round-tripping properties in § 4.2; here we establish the relevant theorems.

Definition 3.7 (Galois connection). Suppose X and Y are sets equipped with partial orders $\leq_X$ and $\leq_Y$. Then monotonic functions $f : X \to Y$ and $g : Y \to X$ form a *Galois connection* $(f, g) : X \to Y$ iff $g(f(x)) \geq_X x$ and $f(g(y)) \leq_Y y$.

Galois connections are also adjoint functors between poset categories, with left and right adjoints f and g usually called the *lower* and *upper* adjoints, because f approximates an inverse of g from below, and g an inverse of f from above. Galois connections compose component-wise, so it is useful to think of them as having a type $X \to Y$, with the direction (by convention) given by the lower adjoint. If $\gamma : X \to Y$ is a Galois connection, we will write γ^* and γ_* for the lower and upper adjoints respectively; an important property we will return to is that γ^* preserves joins and γ_* preserves meets. We now show that, for any $\mathcal{A}$, $\searrow_T$ and $\nearrow_T$ form a Galois connection (Theorem 3.11), by first establishing that the relevant auxiliary functions also form Galois connections.

Theorem 3.8 (Galois connection for pattern-matching). *Suppose* $w :: \mathbf{v}, \boldsymbol{\sigma} \rightsquigarrow \boldsymbol{\rho}, \boldsymbol{\kappa}$. *Then* $(\searrow_w, \nearrow_w) : (\mathsf{Sel}_{\boldsymbol{\rho},\boldsymbol{\kappa}}\,\mathcal{A}) \times \mathcal{A} \to \mathsf{Sel}_{\mathbf{v},\boldsymbol{\sigma}}\,\mathcal{A}$ *is a Galois connection.*

Proof. Included with supplementary materials. □

Lemma 3.9 (Galois connection for environment lookup). *Suppose* $x\colon \mathbf{v} \in \boldsymbol{\rho}$. *Then* $(- \ni_\rho x\colon, \in_\rho x\colon -) : \mathsf{Sel}_{\mathbf{v}}\,\mathcal{A} \to \mathsf{Sel}_{\boldsymbol{\rho}}\,\mathcal{A}$ *is a Galois connection.*

Proof. Included with supplementary materials. □

Theorem 3.10 (Galois connection for recursive bindings). *Suppose* $\boldsymbol{\rho}, \mathbf{h} \twoheadrightarrow \boldsymbol{\rho}'$. *Then* $(\searrow_{\rho,\mathbf{h}}, \nearrow_{\rho,\mathbf{h}}) : \mathsf{Sel}_{\boldsymbol{\rho}'}\,\mathcal{A} \to (\mathsf{Sel}_{\boldsymbol{\rho},\mathbf{h}}\,\mathcal{A}) \times \mathcal{A}$ *is a Galois connection.*

Proof. Included with supplementary materials. □

We assume (rather than prove) that the backward and forward dependency functions $\phi_{\vec{n}}{}^*$ and $\phi_{\vec{n}*}$ provided for every primitive operation $\phi : \mathtt{Int}^i \to \mathtt{Int}$ and every $\vec{n}$ of length i form a Galois connection of type $\mathcal{A} \to \mathcal{A}^i$. Under this assumption the following holds.

Theorem 3.11 (Galois connection for evaluation). *Suppose* $T :: \boldsymbol{\rho}, \mathbf{e} \Rightarrow \mathbf{v}$. *Then* $(\searrow_T, \nearrow_T) : \mathsf{Sel}_{\mathbf{v}}\,\mathcal{A} \to (\mathsf{Sel}_{\boldsymbol{\rho},\mathbf{e}}\,\mathcal{A}) \times \mathcal{A}$ *is a Galois connection.*

Proof. Included with supplementary materials. □

Establishing that $(\searrow_T, \nearrow_T)$ is an adjoint pair might seem rather weak as a correctess property: it merely ensures that the two analyses are related in a sensible way, not that they actually capture any useful information. This is a familiar problem from other approximate analyses like type systems and model checking, where properties like soundness or completeness are essential but do not by themselves guarantee utility. One could certainly define versions of $\searrow_T$ and $\nearrow_T$ that are too coarse grained to be useful, yet still satisfy Theorem 3.11. However Galois connections do at least require that every tightening or tweak to the forward analysis is paired with a corresponding adjustment to the backward analysis, and vice-versa. In § 6 we consider how other ideas from provenance and program slicing might be adapted to provide additional correctness criteria.

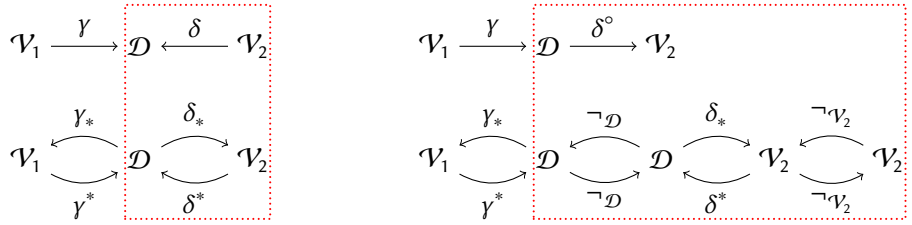
4 DE MORGAN DEPENDENCIES FOR BRUSHING AND LINKING

§ 3 addresses the first kind of question we motivated in the introduction (§ 1.1). In particular $\searrow_T$ can answer questions like: "what data is needed to compute this bar in a bar chart?", and indeed we were able to use our implementation to generate Figure 1. The second problem we set ourselves was how to link selections between *cognate* outputs, i.e. outputs computed from the same data (§ 1.2). This is called "brushing and linking" in data visualisation [Becker and Cleveland 1987], and has been extensive studied as an interaction paradigm, but with little emphasis on techniques for automation. Intuitively, the problem has a bidirectional flavour: one must consider how dependencies flow backward from a selection in one output to a selection v in the common data, and then forward from the selected data v to a corresponding selection in the other output. A natural question then is whether the analysis established in § 3 can supply the information required to support an automated solution.

An immediate problem is that the flavour of the forward dependency required here differs from that provided by the forward analysis $\nearrow_T$ defined in § 3.2. That was able to answer the question: what can we compute given only the data selected in v? But to identify the related data in another output, we must determine not what the input selection v is sufficient for, but what it is necessary for: those parts of the other output that depend on v. In fact the question can be formulated as a kind of dual: what would we *not* be able to compute if the data selected in v were *unavailable*?

4.1 De Morgan Duality

Why $\nearrow_T$ is unsuitable as a forward dependency relation for linking cognate outputs can also be understood in terms of compositionality. Suppose $\mathcal{V}_1$ and $\mathcal{V}_2$ are the lattices of selections for two views computed from a shared input source, and $\mathcal{D}$ is the lattice of selections for the shared input. Using the procedure given in § 3, we can obtain two Galois connections $\gamma : \mathcal{V}_1 \to \mathcal{D}$ and $\delta : \mathcal{V}_2 \to \mathcal{D}$ as shown in Figure 13a. (The reader can ignore Figure 13b for the moment.)



(a) γ and δ cannot be composed (b) Composing via De Morgan duality

Fig. 13. Dualising $\delta : \mathcal{V}_2 \to \mathcal{D}$ for composition with $\gamma : \mathcal{V}_1 \to \mathcal{D}$

Unfortunately, γ and δ are not composable, as their types makes clear. While the upper adjoint $\delta_* : \mathcal{D} \to \mathcal{V}_2$ has the correct type to compose with the lower adjoint $\gamma^* : \mathcal{V}_1 \to \mathcal{D}$, the result is not a Galois connection: δ_* preserves meets, whereas γ^* preserves joins. However, it turns out that if selections are closed under complement, we can derive an analysis of what is *necessary* for a given input selection from an analysis of what it is *sufficient* for. The effect is to invert δ, yielding a Galois connection δ° with a type that allows it to compose with γ. Then the composite $\delta^\circ \circ \gamma$ is a Galois connection linking $\mathcal{V}_1$ to $\mathcal{V}_2$ via $\mathcal{D}$, as shown in Figure 13b, offering a general mechanism for brushing and linking, with nice round-tripping properties. We now unpack this in more detail.

First we shift settings from the lattices used in § 3 to Boolean lattices (or Boolean algebras) $\mathcal{A} = \langle \mathcal{A}, \top, \bot, \sqcap, \sqcup, \neg \rangle$, which are lattices equipped with an involution $\neg : \mathcal{A} \to \mathcal{A}$ called *complement*. Boolean algebras satisfy complementation laws $x \sqcap \neg x = \bot$ and $x \sqcup \neg x = \top$ and De Morgan laws $\neg x \sqcap \neg y = \neg(x \sqcup y)$ and $\neg x \sqcup \neg y = \neg(x \sqcap y)$. If $\mathcal{A}$ is a Boolean algebra, then $\mathsf{Sel}_{\mathsf{v}}\, \mathcal{A}$ is also a Boolean algebra, with the Boolean operations, and in particular $\neg_{\mathsf{v}} : \mathsf{Sel}_{\mathsf{v}}\, \mathcal{A} \to \mathsf{Sel}_{\mathsf{v}}\, \mathcal{A}$, defined pointwise. An additional distinguished value selection $\blacksquare$ serves as the negation of $\square$. The two-point lattice 2 we used to illustrate § 3 is also a Boolean algebra $\langle \{\mathsf{tt}, \mathsf{ff}\}, \mathsf{tt}, \mathsf{ff}, \wedge, \vee, \neg \rangle$ with $\neg$ corresponding to logical negation.

It is an easy consequence of the complementation and De Morgan laws that any meet-preserving operation $g : \mathcal{A} \to \mathcal{B}$ on Boolean algebras has a join-preserving De Morgan dual $g^\circ : \mathcal{A} \to \mathcal{B}$ given by $\neg_{\mathcal{B}} \circ g \circ \neg_{\mathcal{A}}$, and any join-preserving operation h has a meet-preserving De Morgan dual h° defined similarly. Moreover if h is the lower adjoint of g, then g° is the lower adjoint of h°. Thus Galois connections on Boolean algebras also admit a (contravariant) notion of De Morgan duality, defined component-wise.

Definition 4.1 (De Morgan dual of a Galois connection). Suppose $\mathcal{A}$ and $\mathcal{B}$ are Boolean algebras and $\gamma : \mathcal{A} \to \mathcal{B}$ is a Galois connection (γ^*, γ_*). Define the *De Morgan dual* γ° of γ to be the Galois connection $({\gamma_*}^\circ, {\gamma^*}^\circ) : \mathcal{B} \to \mathcal{A}$.

Dualising a Galois connection flips the direction of the arrow by swapping the roles of the upper and lower adjoints. So while $\gamma : \mathcal{A} \to \mathcal{B}$ and $\delta : C \to \mathcal{B}$ are not composable, γ and $\delta^\circ : \mathcal{B} \to C$ are, and the composition is achieved by transforming δ_* from something which determines what we can compute with v into something which determines what we cannot compute without v. This offers a principled basis for an automated brushing and linking feature between cognate computations T and U. When the user selects part of the output of T, we can use $\searrow_T$ to compute the needed data v, and then use ${\nearrow_U}^\circ$ to compute the parts of the output of U that depend on v. This is the approach implemented in Fluid, and we used this to generate Figure 2 in § 1.2.

```
let zero n = const n;
    wrap n n_max = ((n - 1) `mod` n_max) + 1;
    extend n = min (max n 1);
    nth2 i j xss = nth (j - 1) (nth (i - 1) xss);

let convolve image kernel method =
    let ((m, n), (i, j)) = (dims image, dims kernel);
        (half_i, half_j) = (i `quot` 2, j `quot` 2);
        area = i * j
    in  ⟨ let weightedSum = sum [
            image!(x, y) * kernel!(i' + 1, j' + 1)
          | (i', j') ← range (0, 0) (i - 1, j - 1),
            let x = method (m' + i' - half_i) m,
            let y = method (n' + j' - half_j) n,
            x ≥ 1, x ≤ m, y ≥ 1, y ≤ n
        ] in weightedSum `quot` area
        | (m', n') in (m, n) ⟩;
```

```
let emboss = [[-2, -1, 0],
              [-1,  1, 1],
              [ 0,  1, 2]];
    filter = ⟨ nth2 i j emboss
             | (i, j) in (3, 3) ⟩;
    image' = [[15, 13,  6, 9, 16],
              [12,  5, 15, 4, 13],
              [14,  9, 20, 8,  1],
              [ 4, 10,  3, 7, 19],
              [ 3, 11, 15, 2,  9]];
    image = ⟨ nth2 i j image'
            | (i, j) in (5, 5) ⟩
in convolve image filter zero
```

Fig. 14. Matrix convolution example, with methods `zero`, `wrap` and `extend` for dealing with boundaries

4.2 Example: Matrix Convolution

We now illustrate the $(\nearrow_T^{\circ}, \searrow_T^{\circ})$ Galois connection, contrasting it with $(\searrow_T, \nearrow_T)$, using an example which computes the convolution of a 5×5 matrix with a 3×3 kernel. Convolution has an intuitive dependency structure and the values involved have an easy visual presentation, making it useful for conveying the flavour of the four distinct (but connected) dependency relations that arise in the framework. The source code for the example is given in Figure 14, and shows the `convolve` function, plus `zero`, `wrap` and `extend` which provide different methods for handling the boundaries of the input matrix. The angle-bracket notation is used to construct matrices, which were omitted from § 2. (The formal treatment is similar to records.)

Fluid was used to generate the diagrams in Figure 15, which show the four dependency relations and two of their four possible round-trips. Figure 15a shows the $(\searrow_T, \nearrow_T)$ Galois connection defined in § 3.4. In the upper figure, the user selects (in green) the output cell at position $(2, 2)$ (counting rows downwards from 1). This induces a demand (via the lower adjoint $\searrow_T$) on the input matrix `image` and the kernel `filter`, revealing (in blue) that the entire kernel was needed to compute the value 1, but only some of the input matrix. In particular the elements at $(1, 3)$ and $(3, 1)$ in `image` were not needed, because of zeros present in `filter`. If we then "round-trip" that input selection, computing the corresponding availability on the output using the upper adjoint $\nearrow_T$, the green selection grows: it turns out that the data needed to make $(2, 2)$ available are sufficient to make $(1, 1)$ available as well.

Figure 15b shows the De Morgan dual $(\nearrow_T^{\circ}, \searrow_T^{\circ})$. In the upper part of the figure, the user selects (green) kernel cell $(1, 2)$ to see the output cells that depend on it. This is computed using the De Morgan dual $\nearrow_T^{\circ}$. First we negate the input selection, marking $(1, 2)$ as unavailable, and all other inputs as available. Then we forward-analyse with $\nearrow_T$ to determine that with this data selection, we can only compute the top row of the output. (If it seems odd that we can compute even the top row, notice that the example uses the method `zero` for dealing with boundaries; `wrap` or `extend` would give a different behaviour.) Then we negate that top row selection to produce the (blue) output selection shown in the figure. These are exactly the output cells which depend on kernel cell $(1, 2)$ in the sense that they cannot be computed if that input is unavailable.

(a) Galois dependency ($\searrow_T, \nearrow_T$) (b) De Morgan dual ($\nearrow_T{}^{\circ}, \searrow_T{}^{\circ}$)

Fig. 15. Upper and lower pairs are dual; left and right pairs are adjoint

We can then round-trip this output selection using the De Morgan dual $\searrow_T{}^{\circ}$. We first negate the blue output selection (selecting the top row of the output again), and then use $\searrow_T$ to determine the needed inputs, which turn out to be the top two rows of `image`, and the top row of `filter`. Negating again produces the green output selection shown in the lower figure. Thus the backwards De Morgan dual computes the inputs that would *not* be needed if the selected outputs were not needed: more economically, the inputs that are *only* needed for the selected output. Here the round-trip reveals that if kernel cell $(1, 2)$ is unavailable, then the entire top row of the kernel might as well have been unavailable too, and similarly for the bottom 3 rows of the input.

4.3 Relationship to Galois Slicing

The De Morgan dual puts us in a better position to consider the relationship between the present system and earlier work on *Galois slicing*, a program slicing technique that has been explored for pure functional programs [Perera 2013; Perera et al. 2012], functional programs with effects [Ricciotti et al. 2017], and π-calculus [Perera et al. 2016]. We consider other related work in § 6.1.

Galois slicing operates on lattices of *slices*, which are programs (or values) where parts deemed irrelevant are replaced by a hole $\square$. (If we think of the notion of selection defined in § 3.1.1 as picking out a subset of the paths in a term, then slices resembles selections which are prefix-closed, meaning that if a given path in a term is selected, then so are all of its prefixes.) For a fixed computation, a meet-preserving *forward-slicing* function is defined which takes input slices to output slices, discarding parts which cannot be computed because the needed input is not present, plus a join-preserving *backward-slicing* function taking output slices to input slices, retaining the parts needed for the output slice. For example Figure 16a shows a computation with output `(0.4, 0.6)`, and Figure 16b gives the backward slice for output slice `(0.4, □)`. Forward and backward slicing, for a given computation, form a Galois connection, giving the analyses the nice round-tripping properties we motivated in § 3.4.

Unfortunately, the notion of slice does not lend itself to computing dependencies where the needed input or output is a proper part of a value, such as a component of a tuple. *Differential* slicing [Perera et al. 2012] improves on this by using Galois slicing to compute a pair of input slices (e, e') for a pair of output slices (v, v') where $v \sqsubseteq v'$. By monotonicity, $e \sqsubseteq e'$. This can be used to

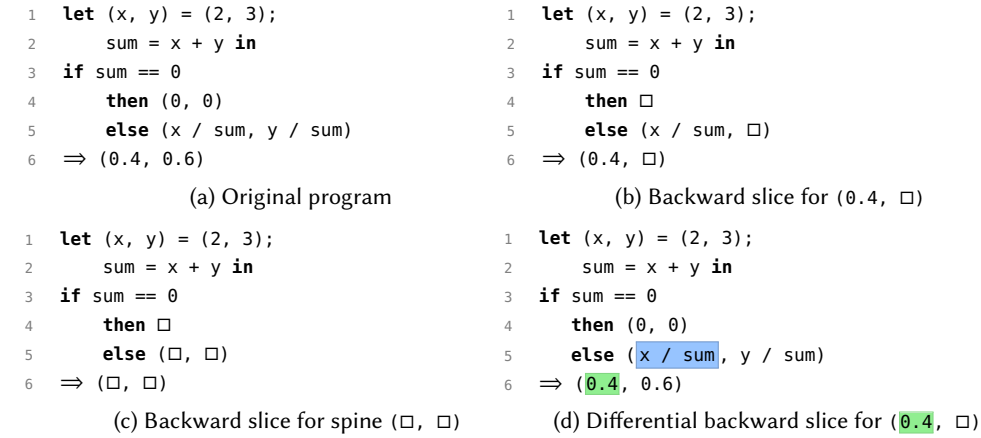


Fig. 16. Differential Galois slicing selects input (blue) needed *only* for selected output (green)

compute a (differential) slice for an arbitrary subtree, by setting v to be the "spine" of the original output up to the location of the subtree, and v' to be v with the subtree of interest plugged back in. Here we could focus on the value `0.4` in the output by computing the backward slice for `(□, □)` (Figure 16c) and then comparing it with the backward slice for `(0.4, □)`, generating a differential slice where the parts that are different are highlighted (Figure 16d). But although it supports a notion of selection which is closer to what we need, the differential slice highlights only the program parts that are needed *exclusively* by the selected output, and as such underapproximates the dependency information needed for data linking. (In fact differential slicing is similar to the De Morgan dual $\searrow_T^\circ$.) Because is this example `2` and `3` are needed to compute the spine as well (in order to decide which conditional branch to execute), they are excluded from the differential slice, whereas our backward analysis $\searrow_T$ is able to directly determine that both `2` and `3` are needed to compute `0.4`.

5 GALOIS CONNECTIONS FOR DESUGARING

Elaborating a richer surface language into a simpler core is a common pattern with well known benefits. It can, however, make it harder to express certain information to the programmer in terms of the surface language. We face this problem with the analysis in § 3, which links outputs not only to inputs, but also to expressions responsible for introducing data. We could use this information in an IDE to link structured outputs to relevant code fragments, but only if we are able to map term selections back to the surface program. We now sketch a bidirectional desugaring procedure which addresses this, and which composes with the Galois dependency analysis defined in § 3.

5.1 Surface Language Syntax

The surface language, Fluid, extends the core syntax with list notation $[s,\ldots,s']$, Haskell 98-style list comprehensions [Jones 2003], list enumerations, first-class primitives and piecewise function definitions and pattern-matching, as shown in Figure 17. Typing rules are included with the supplementary materials. We attach selection states α to surface terms s, t that desugar to core terms with selections, and let $\mathbf{s}$, $\mathbf{t}$ range over "raw" surface terms, which are isomorphic to the term selections where the type of selection states is the unit lattice 1.

Figure 18 shows how the end-to-end mapping would appear to a user. (For illustrative purposes the library function `map` and some raw data are included in the same source file.) On the left, the user

	Identifier	
$x, y ::=$	…	
	$\oplus$	operator name
	Surface term	
$s, t ::=$	…	
	$(\oplus)$	first-class operator
	$s \oplus s'$	binary application
	`let` $\overrightarrow{g}$ `in` s	recursive functions
	`if` s `then` s `else` s	if
	`match` s `as` $\overrightarrow{p = s}$	match
	`let` $p = s$ `in` s	structured let
	$[_\alpha\ s\ r$	non-empty list
	`[` s `..` s `]`	list enum
	`[` $s \mid \overrightarrow{q}\ ]_\alpha$	list comprehension
	List rest term	
$r ::=$	$]_\alpha$	end
	$,_\alpha\ s\ r$	cons

	Recursive function	
$g ::=$	$x\text{:}\ \overrightarrow{c}$	
	Clause	
$c ::=$	$\overrightarrow{p} = s$	
	Pattern	
$p ::=$	x	variable
	$(x\text{:}\ \overrightarrow{p})$	record
	`[]`	nil
	p `:` p	cons
	`[` $p\ o$	non-empty list
	List rest pattern	
$o ::=$	`]`	end
	`,` $p\ o$	cons
	Qualifier	
$q ::=$	s	guard
	`let` $p = s$	declaration
	$p \leftarrow s$	generator

Fig. 17. Syntax for surface language, with selection states

selects a cons cell (green) in the output; by backwards evaluating and then backwards desugaring, we are able to highlight the list comprehension, the cons in the second clause of `map`, and both occurrences of the constant `"Hydro"`. These last two are highlighted because the selected cons cell was constructed by eliminating a Boolean that was in turn constructed by the primitive == operator, which consumed the two strings. The user might then conjecture that the two occurrences of `"Geo"` were somehow responsible for the inclusion of the third cons cell in the output; they can confirm this by making the green selection on the right. (Highlighting == too would clearly be helpful here; we discuss this possibility in § 6.1.) The grey selection is included to contrast the cons highlighting with the data demanded by the list elements themselves, which is quite different.

```
let map f [] = [];
    map f (x : xs) = f x : map f xs;
let data = [
   { energyType: "Bio", output: 6.2 },
   { energyType: "Hydro", output: 260 },
   { energyType: "Solar", output: 19.9 },
   { energyType: "Wind", output: 91 },
   { energyType: "Geo", output: 14.4 }
];
let xs = [ row.output
   | type ⟵ ["Hydro", "Solar", "Geo"],
     row ⟵ data, row.energyType == type
] in
map (fun x ⟶ floor (x / sum xs * 100)) xs
⟹ (88 : (6 : (4 : [])))
```

```
let map f [] = [];
    map f (x : xs) = f x : map f xs;
let data = [
   { energyType: "Bio", output: 6.2 },
   { energyType: "Hydro", output: 260 },
   { energyType: "Solar", output: 19.9 },
   { energyType: "Wind", output: 91 },
   { energyType: "Geo", output: 14.4 }
];
let xs = [ row.output
   | type ⟵ ["Hydro", "Solar", "Geo"],
     row ⟵ data, row.energyType == type
] in
map (fun x ⟶ floor (x / sum xs * 100)) xs
⟹ (88 : (6 : (4 : [])))
```

Fig. 18. Source selections (blue) resulting from selecting different list cells (green)

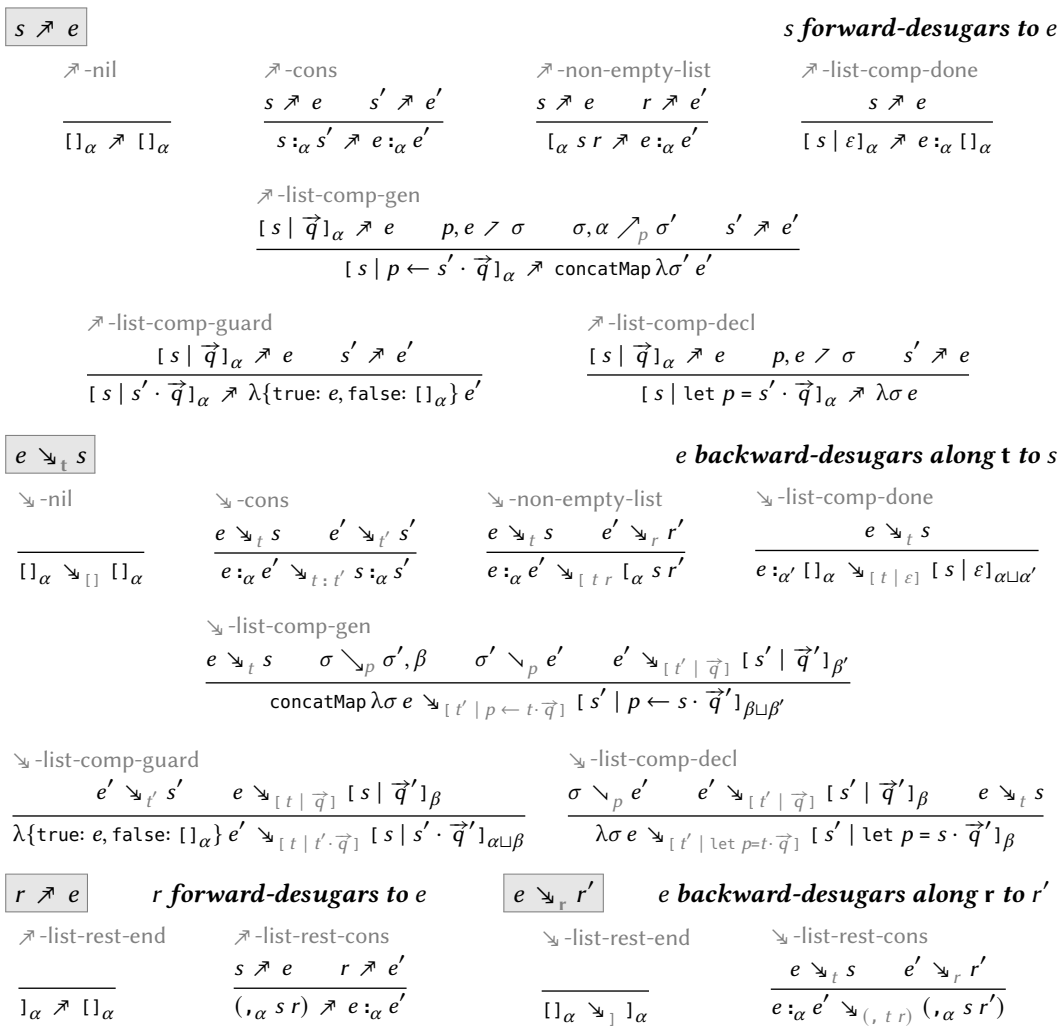


Fig. 19. Forwards and backwards desugaring (selected rules only)

5.2 Forward Desugaring

To define the forward evaluation function $\nearrow_T$ in § 3.2, we performed a regular evaluation using $\Rightarrow$ to obtain a trace T, and then defined $\nearrow_T$ by recursion over T, with T guiding the analysis in the presence of $\Box$. There are no holes in the surface language, so we can take a simpler approach, defining a single *forward desugaring* relation $\nearrow$, and then showing that for every raw surface term $\mathbf{t} \nearrow \mathbf{e}$, there is a monotonic function $\nearrow_t : \mathrm{Sel}_{\mathbf{t}}\,\mathcal{A} \to \mathrm{Sel}_{\mathbf{e}}\,\mathcal{A}$, which is simply $\nearrow$ domain-restricted to $\mathrm{Sel}_{\mathbf{t}}\,\mathcal{A}$. The full definition of $\nearrow$ is included with the supplementary materials; Figure 19 gives a representative selection of the rules.

The definition follows a similar pattern to $\nearrow_T$. At each step, we take the meet of the availability on any parts of s being consumed at that step, and use that as the availability of any parts of e being generated at that step. Thus the rules for list notation simply transfer the selection state α on the opening and closing brackets $[_\alpha$ and $]_\alpha$ to the corresponding cons and nil of the resulting list,

and those on intervening delimiters $,_\alpha$ to the corresponding cons. List comprehensions $[\, s \mid \overrightarrow{q} \,]_\alpha$ have a rule for each kind of qualifier q at the head of $\overrightarrow{q}$, plus a rule for when $\overrightarrow{q}$ is ε. The general pattern is to push the α on the comprehension itself through recursively, so it ends up on all core terms generated during its elaboration: in particular the `false` branch when q is a guard, and the singleton list when $\overrightarrow{q}$ is empty. Auxiliary relations $\nearrow$ and $\nearrow_p$ (included with the supplementary materials) transfer availability on guards and generators onto the eliminators they elaborate into.

5.3 Backward Desugaring

The backwards analysis is then defined as a family of *backward desugaring* functions $\searrow_t : \mathsf{Sel}_\mathbf{e}\, \mathcal{A} \to \mathsf{Sel}_\mathbf{t}\, \mathcal{A}$ for any $\mathbf{t} \nearrow \mathbf{e}$, with the raw surface term $\mathbf{t}$ guiding the analysis backwards. (The role of $\mathbf{t}$ in disambiguating the backwards rules should be clear if you consider that e typically matches multiple rules but only one for a given $\mathbf{t}$.) Figure 19 gives some representative rules; the full definition is included with the supplementary materials. To reverse a desugaring step, we take the join of the demand on any parts of e which were constructed at this step, and use that as the demand on the parts of s which were consumed at this step, turning demand on the core term into (minimal) demand on the surface term. Thus the effect of the list comprehension rules and auxiliary judgements is to set the demand on the comprehension itself to be the join of the demand of all the syntax generated during the elaboration of the comprehension, using auxiliary judgments $\searrow_p$ and $\searrow_p$ to transfer demand from eliminators back onto the guards and generators.

5.4 Round-Tripping Properties and Compositionality

It is easy to verify that $\nearrow_t$ and $\searrow_t$ are monotonic. Moreover they form an adjoint pair.

Theorem 5.1 (Galois connection for desugaring). *Suppose $t \nearrow e$. Then $(\searrow_t, \nearrow_t) : \mathsf{Sel}_\mathbf{e}\, \mathcal{A} \to \mathsf{Sel}_\mathbf{t}\, \mathcal{A}$ is a Galois connection.*

Proof. Included with supplementary materials. □

The $(\searrow_t, \nearrow_t)$ Galois connection readily composes with $(\searrow_t, \nearrow_t)$ to produce surface-language selections like the ones shown in Figure 18. This is useful, although somewhat monolithic. In future work we will investigate techniques for backwards desugaring at arbitrary steps in the computation, perhaps by interleaving desugaring with execution in the style of Pombrio and Krishnamurthi [2014], as well as presenting selections on intermediate values (such as lists) in the surface language, even though they were not obtained via desugaring.

6 CONCLUSION

Our research was motived by the goal of making computational outputs which are automatically able to reveal how they relate to data in a fine-grained way. A casual reader who wants to understand or fact-check a chart, or a scientist evaluating another's work, should be able to do so by interacting directly with an output. Recent work by Walny et al. [2019] suggests that developers would also benefit from such a feature while implementing visualisations, for example to check whether a quantity is represented by diameter or area in a bubble chart.

Galois connections provide an appealing setting for this problem because of their elegant round-tripping properties. However, existing dynamic analysis techniques based on Galois connections do not lend themselves to richly structured outputs like visualisations and matrices. We presented an approach that allows focusing on arbitrary substructures, which also means data selections can be inverted. This enables linking not just of outputs to data, but of outputs to other outputs, providing a mathematical basis for a widely used (but so far ad hoc) feature in data visualisation. We implemented our approach in Fluid, a realistic high-level functional programming language.

6.1 Other Related Work and Future Directions

We close by considering some limitations and opportunities in the context of other related work. Galois slicing [Perera et al. 2012, 2016; Ricciotti et al. 2017] was considered in § 4.3.

Executable slicing. Executable slices [Hall 1995] are programs with some parts removed, but which are still executable. Our approach computes data selections, not executable slices, but such a notion has obvious relevance in data science: "explaining" part of a result should (arguably) entail being able to recompute it. *Expression provenance* [Acar et al. 2012] explains how primitive values are computed using only primitive operations; however, this still omits crucial information, and does not obviously generalise to structured outputs. Work on executable slicing in term rewriting [Field and Tip 1998] could perhaps be adapted to structured data and combined with dependency tracking for higher-order data (§ 3.1).

Dynamic program analysis. Dynamic analysis techniques like dataflow analysis [Chen and Poole 1988] and taint tracking [Reps et al. 1995] tend to focus on variables, rather than parts of structured values, and lack round-tripping properties; Galois dependencies have a clear advantage here. A limitation of dynamic techniques which is shared by our approach is that they can usually only reveal *that* certain dependencies arise, not *why*, which requires analysing path conditions [Hammer et al. 2006]. In a data science setting this would clearly be valuable too, and it would be interesting to see if the benefits of the Galois framework can be extended to techniques for computing dynamic path conditions.

Brushing and linking. Brushing and linking has been extensive studied in the data visualisation community [Becker and Cleveland 1987; McDonald 1982], but although Roberts and Wright [2006] argued it should be ubiquitous, no automated method of implementation has been proposed to date. Geospatial applications like GeoDa [Anselin et al. 2006] hard-code view coordination features into specific views, and libraries like d3.js and Plotly support ad hoc linking mechanisms, with varying degrees of programmer effort required. No existing approach provides automation or round-tripping guarantees, or is able to provide data selections explaining why visual selections are linked.

Data provenance in data visualisation. A recent vision paper by Psallidas and Wu [2018] is the only work we are aware of that proposes that brushing and linking, and related view cooordination features like cross-filtering, can be understood in terms of data provenance. In a relational (query processing) setting, where the relevant notion of provenance is lineage, they propose backward-analysing to data, and then forward-analysing to another view, although again without the round-tripping features of Galois connections. Moreover theirs is primarily a concept paper, proposing a research programme, rather than solving a specific problem.

Acknowledgements. Perera and Petricek were supported by The UKRI Strategic Priorities Fund under EPSRC Grant EP/T001569/1, particularly the *Tools, Practices and Systems* theme within that grant, and by The Alan Turing Institute under EPSRC grant EP/N510129/1. Wang was supported by *Expressive High-Level Languages for Bidirectional Transformations*, EPSRC Grant EP/T008911/1.

REFERENCES

Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. 2012. A Core Calculus for Provenance. In *Proceedings of the First International Conference on Principles of Security and Trust* (Tallinn, Estonia) *(POST '12)*. Springer-Verlag, Berlin, Heidelberg, 410–429. https://doi.org/10.1007/978-3-642-28641-4_22

Luc Anselin, Ibnu Syabri, and Youngihn Kho. 2006. GeoDa: An Introduction to Spatial Data Analysis. *Geographical Analysis* 38, 1 (2006), 5–22. https://doi.org/10.1111/j.0016-7363.2005.00671.x

Richard A. Becker and William S. Cleveland. 1987. Brushing Scatterplots. *Technometrics* 29, 2 (May 1987), 127–142. https://doi.org/10.1080/00401706.1987.10488204

Richard Bird and Lambert Meertens. 1998. Nested datatypes. In *Mathematics of Program Construction*, Johan Jeuring (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–67.

Gérard Boudol and Ilaria Castellani. 1989. Permutation of transitions: An event structure semantics for CCS and SCCS. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, J.W. Bakker, W.-P. Roever, and G. Rozenberg (Eds.). Lecture Notes in Computer Science, Vol. 354. Springer, 411–427. https://doi.org/10.1007/BFb0013028

Nadieh Bremer and Marlieke Ranzijn. 2015. Urbanization in East Asia between 2000 and 2010. http://nbremer.github.io/urbanization/.

Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *Proceedings of the 8th International Conference on Database Theory (ICDT '01)*. Springer-Verlag, London, UK, 316–330.

TY Chen and PC Poole. 1988. Dynamic dataflow analysis. *Information and Software Technology* 30, 8 (1988), 497–505. https://doi.org/10.1016/0950-5849(88)90146-2

Richard H. Connelly and F. Lockwood Morris. 1995. A generalization of the trie data structure. *Mathematical Structures in Computer Science* 5, 3 (1995), 381–418. https://doi.org/10.1017/S0960129500000803

A. De Lucia, A.R. Fasolino, and M. Munro. 1996. Understanding function behaviors through program slicing. In *WPC '96. 4th Workshop on Program Comprehension*. 9–18. https://doi.org/10.1109/WPC.1996.501116

John Field and Frank Tip. 1998. Dynamic Dependence in Term Rewriting Systems and its Application to Program Slicing. *Information and Software Technology* 40, 11–12 (November/December 1998), 609–636.

Jeremy Gibbons. 2017. APLicative Programming with Naperian Functors. In *European Symposium on Programming (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). 568–583. https://doi.org/10.1007/978-3-662-54434-1_21

Sebastian Graf, Simon Peyton Jones, and Ryan G Scott. 2020. Lower your guards: a compositional pattern-match coverage checker. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–30.

Robert J. Hall. 1995. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering* 2 (1995), 33–53. https://doi.org/10.1007/BF00873408

Christian Hammer, Martin Grimme, and Jens Krinke. 2006. Dynamic path conditions in dependence graphs. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 58–67. https://doi.org/10.1145/1111542.1111552

Ralf Hinze. 2000. Generalizing generalized tries. *Journal of Functional Programming* 10, 4 (2000), 327–351. https://doi.org/10.1017/S0956796800003713

Simon L. Peyton Jones. 2003. Haskell 98: Introduction. *Journal of Functional Programming* 13, 1 (2003), 0–6.

Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL '73)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945

John Alan McDonald. 1982. *Interactive graphics for data analysis*. Ph.D. Dissertation.

Greg Miller. 2006. A Scientist's Nightmare: Software Problem Leads to Five Retractions. *Science* 314, 5807 (2006), 1856–1857. https://doi.org/10.1126/science.314.5807.1856

James Newsome and Dawn Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed Systems Security Symposium*.

Roland Perera. 2013. *Interactive Functional Programming*. Ph.D. Dissertation. University of Birmingham, Birmingham, UK. http://etheses.bham.ac.uk/4209/.

Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. 2012. Functional Programs That Explain Their Work. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* (Copenhagen, Denmark) *(ICFP '12)*. ACM, New York, NY, USA, 365–376. https://doi.org/10.1145/2364527.2364579

Roly Perera, Deepak Garg, and James Cheney. 2016. Causally Consistent Dynamic Slicing. In *Concurrency Theory, 27th International Conference, CONCUR '16 (Leibniz International Proceedings in Informatics (LIPIcs))*, Josée Desharnais and Radha Jagadeesan (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany. https://doi.org/10.4230/LIPIcs.CONCUR.2016.18

Justin Pombrio and Shriram Krishnamurthi. 2014. Resugaring: Lifting Evaluation Sequences through Syntactic Sugar. *SIGPLAN Notices* 49, 6 (Jun 2014), 361–371. https://doi.org/10.1145/2666356.2594319

Fotis Psallidas and Eugene Wu. 2018. Provenance for Interactive Visualizations. In *Workshop on Human-In-the-Loop Data Analytics (HILDA 2018)*. ACM.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

Wilmer Ricciotti, Jan Stolarek, Roly Perera, and James Cheney. 2017. Imperative Functional Programs That Explain Their Work. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 14 (2017), 28 pages. https://doi.org/10.1145/3110258

J. C. Roberts and M. A. E. Wright. 2006. Towards Ubiquitous Brushing for Information Visualization. In *Tenth International Conference on Information Visualisation (IV'06)*. 151–156. https://doi.org/10.1109/IV.2006.113

A. Sabelfeld and A. C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan 2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

Jacob VanderPlas, Brian E. Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. 2018. Altair: Interactive Statistical Visualizations for Python. *The Journal of Open Source Software* 3, 32 (2018). https://doi.org/10.21105/joss.01057

Jagoda Walny, Christian Frisson, Mieka West, Doris Kosminsky, Søren Knudsen, Sheelagh Carpendale, and Wesley Willett. 2019. Data Changes Everything: Challenges and Opportunities in Data Visualization Design Handoff. *IEEE Transactions on Visualization and Computer Graphics* PP (08 2019), 1–1. https://doi.org/10.1109/TVCG.2019.2934538

Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, California, USA) *(ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 439–449. https://doi.org/10.5555/800078.802557