Datafun: a Functional Datalog

Michael Arntzenius   Neelakantan R. Krishnaswami
University of Birmingham (UK)
{daekharel, neelakantan.krishnaswami}@gmail.com

Abstract

Datalog may be considered either an unusually powerful query language or a carefully limited logic programming language. Datalog is declarative, expressive, and optimizable, and has been applied successfully in a wide variety of problem domains. However, most use-cases require extending Datalog in an application-specific manner. In this paper we define Datafun, an analogue of Datalog supporting higher-order functional programming. The key idea is to *track monotonicity with types*.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

Keywords Prolog, Datalog, logic programming, functional programming, domain-specific languages, type theory, denotational semantics, operational semantics, adjoint logic

1. Introduction

The phrase "declarative programming" is as popular as it is ambiguous, with seemingly hundreds of disparate senses in which it is used. However, two of those usages stand out for popularity: both *functional* and *logic* programming languages are generally deemed declarative languages. Despite this shared epithet, the logic and functional programming traditions have largely evolved independently of one another (with a few honorable exceptions such as Mercury (Somogyi et al. 1994), Curry (Antoy and Hanus 2010) and Kanren (Friedman et al. 2005)). This could be seen as an occasion for sorrow, but we prefer to view it as an opportunity: as functional language designers, we can look to logic languages to discover new ideas to steal.

A Prolog program can be understood as a collection of logical axioms formulated as Horn clauses (i.e., first-order formulas of the form $\forall \vec{x}.\ P_1 \wedge \ldots \wedge P_n \rightarrow Q$, where P_i and Q are atomic formulas). Execution of a Prolog program can be understood as running a proof search algorithm on these clauses to figure out whether a particular formula is derivable or not.

In other words, functional and logic programming languages embody the Curry-Howard correspondence in two different ways. In a functional language, types are propositions, terms are proofs, and program evaluation corresponds to proof normalization. On the other hand, for logic programming languages, *terms* are propositions, and program evaluation corresponds to *proof search*.

Since proof search is in general undecidable, designers of logic programming languages must be careful both about the kinds of formulas they admit as programs, and about the proof search algorithm they implement. Prolog offers a very expressive language — full Horn clauses — and so faces an undecidable proof search problem. Therefore, Prolog specifies its proof search strategy: depth-first goal-directed/top-down search. This lets Prolog programmers reason about the behaviour of their programs; however, it also means many logically natural programs fail to terminate. Notoriously, transitive closure calculations are much less elegant in Prolog than one might hope, since their most natural specification is best computed with a bottom-up (aka "forwards chaining") proof search strategy.

This view of Prolog suggests other possible design choices, such as restricting the logical language so as to make proof search decidable. One of the oldest such variants is Datalog (Gallaire and Minker 1978), a subset of Prolog satisfying three restrictions:

1. Programs must be *constructor-free*: only atomic terms and variables are permitted to appear as arguments to predicates. This ensures that deduction will not introduce terms that do not occur in the source of the program.
2. Clauses are *range-restricted*: all variables in the consequent (head) of a clause must also occur positively in its premises (body).
3. Programs are limited to *stratified negation*: the negation of a predicate may be used in a definition only if it has already been fully defined. That is, within the recursive definition of a predicate, it cannot be used in negated form.

These restrictions make Datalog Turing-*in*complete: all queries are decidable. As functional programmers are well aware, though, there is power in restraint: for example, in a total functional language, the compiler may switch between strict and lazy evaluation at will. Similarly, in Datalog decidability means that implementations are free to use forwards chaining, and so can easily support queries (like reachability and transitive closure) which are difficult to implement in ordinary Prolog.

Over the last decade or so, this freedom has been put to good use, with Datalog appearing at the heart of a a wild variety of applications in both research and industry. For example, Whaley and Lam (Whaley et al. 2005; Whaley 2007) implemented pointer analysis algorithms in Datalog, and found that they could reduce their analyses from thousands of lines of C code to *tens* of lines of Datalog code, while retaining competitive performance. Semmle has developed the .QL language (de Moor et al. 2007; Schäfer and de Moor 2010) based on Datalog for analysing source code (which was used to analyze the code for NASA's Curiosity Mars rover), and LogicBlox has developed the LogiQL (Aref et al. 2015) language for business analytics. The Boom project at Berkeley has developed the Bloom language for distributed programming (Alvaro et al. 2011), and the Datomic cloud database (Hickey et al.) uses Datalog

(embedded in Clojure) as its query language. Microsoft's SecPAL language (Becker et al. 2010) uses Datalog as the foundation of its decentralised authorization specification language.

In all of these cases, the use of Datalog permits giving specifications and implementations which are dramatically shorter and clearer than alternatives implemented in more conventional languages. However, while all of these applications are built on a foundation of Datalog, they all also extend it significantly. For example, it is impossible even to implement arithmetic in Datalog, since adding 2 and 3 produces 5, which is a new term not equal to either 2 or 3! As a result, even though Datalog has a very clean semantics, its metatheory needs to be re-established once again for each application-specific extension to it.

As a result, it would be very desirable to understand what makes Datalog tick, so that we can embed it into a more expressive language *without* sacrificing the properties that make it so powerful within its domain. In this way, extensions can become "a small matter of programming", without having to do a custom redesign of the language for each application.

In this paper, we present Datafun, a typed functional language which permits programming in the style of Datalog, while still supporting the full power of higher-order functional programming.

Contributions

- We describe Datafun, a typed language capturing the expressive power of Datalog and extending it to support higher-order functional programming. Datafun's key feature is to *track monotonicity with types*. This permits us to use typing to analyze fixed point computations in a way ensuring their termination.
- We present examples illustrating the expressive power of Datafun, including relational-algebra-style operations, transitive closure, CYK parsing, and dataflow analysis. Some of these examples are familiar from Datalog, but many of them go well beyond what can be expressed in it, illustrating the benefits of our approach.
- We identify the semantic structures underpinning Datalog, and use this to give a denotational semantics for Datafun in terms of a pair of adjunctions between Set, Poset, SemiLat.
- We have a prototype implementation of Datafun in Racket, which has been used to implement all of the examples in this paper, and is available at `https://github.com/rntz/datafun/`.

2. Datafun, informally

We give the core syntax of Datafun in Figure 1. Datafun is a simply-typed λ-calculus extended in four major ways:

1. We add a type of finite sets, $\{A\}$.
2. We add a type of *monotone functions*, $A \xrightarrow{+} B$. Consequently Datafun has two flavors of variable: ordinary variables, which we call *discrete*, and *monotone* variables. We write discrete variables in script and monotone variables in **bold**.

   In order for "monotone" to have meaning, our types are implicitly partially ordered:

   - Booleans 2 are ordered $\mathsf{false} < \mathsf{true}$.
   - Natural numbers $\mathbb{N}$ have the usual order: $0 < 1 < 2 < \ldots$.
   - We have no particular use-case for comparing strings str in this paper, so we order them discretely; $a \le b$ iff $a = b$.
   - Pairs and functions are ordered pointwise:
     - $(a, x) \le (b, y)$ iff $a \le b \wedge x \le y$
     - $f \le g$ iff $\forall x.\, f\, x \le g\, x$
   - Sum types are ordered disjointly: $\mathsf{in}_i\ a \le \mathsf{in}_i\ b$ iff $a \le b$, but $\mathsf{in}_1\ a$ and $\mathsf{in}_2\ b$ are never comparable.
   - Sets are ordered by inclusion: $a \le b$ iff $a \subseteq b$.
3. We add a term ($\mathsf{fix}\ \mathbf{x}\ \mathsf{is}\ e$) denoting the least fixed point of the monotone function ($\lambda\mathbf{x}.\, e$). This is computed (modulo optimizations) by iteration, starting from the smallest value of the desired type and halting once a fixed point is found. This strategy constrains the types of fix terms in several ways:
   - The type must have a smallest value. We enforce this using semilattice types (see item 4, below).
   - The type must support equality tests, to determine when a fixed point has been reached. We call a type supporting equality tests an *eqtype*.
   - To ensure termination, the type must have finite height.[1] We conservatively approximate this property by limiting fix to finite types.

   In summary, fix may only be used at *finite semilattice eqtypes*.
4. Generalizing the empty set $\emptyset$ and union $\cup$, we identify a subset of types that have a *least element* ε and a *least upper bound* operator $\vee$. We call these *semilattice types*[2], and denote them by the metavariables L, M.

   Semilattice types serve two purposes. First, as already mentioned, they guarantee the presence of a least element, needed to compute fix terms.

   Second, they provide a natural eliminator for sets. Given $e_1 : \{A\}$, we write $\bigvee(x \in e_1)\ e_2$ for the least upper bound, over all elements $x \in e_1$, of e_2, provided e_2 has some semilattice type L.

A, B types	$::=$	$2 \mid \mathbb{N} \mid \mathsf{str} \mid \{A\} \mid A + B \mid A \times B$ $A \to B \mid A \xrightarrow{+} B$
L, M semilattice types	$::=$	$2 \mid \mathbb{N} \mid \{A\} \mid L \times M \mid A \to L \mid A \xrightarrow{+} L$
$\underset{eq}{A}, \underset{eq}{B}$ eqtypes	$::=$	$2 \mid \mathbb{N} \mid \mathsf{str} \mid \{\underset{eq}{A}\} \mid \underset{eq}{A} + \underset{eq}{B} \mid \underset{eq}{A} \times \underset{eq}{B}$
$\underset{fin}{A}, \underset{fin}{B}$ finite eqtypes	$::=$	$2 \mid \{\underset{fin}{A}\} \mid \underset{fin}{A} + \underset{fin}{B} \mid \underset{fin}{A} \times \underset{fin}{B}$
Δ	$::=$	$\cdot \mid \Delta, x : A$
Γ contexts	$::=$	$\cdot \mid \Gamma, \mathbf{x} : A$
e terms	$::=$	$x \mid \mathbf{x} \mid \lambda x.\, e \mid \lambda\mathbf{x}.\, e \mid e\ e$ $(e, e) \mid \pi_1\ e \mid \pi_2\ e \mid \mathsf{in}_1\ e \mid \mathsf{in}_2\ e$ $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{in}_1\ x \to e;\ \mathsf{in}_2\ x \to e$ $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{in}_1\ \mathbf{x} \to e;\ \mathsf{in}_2\ \mathbf{x} \to e$ $\mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$ $\{e\} \mid \varepsilon \mid e \vee e \mid \bigvee(x \in e)\ e$ $\mathsf{fix}\ \mathbf{x}\ \mathsf{is}\ e \mid \mathsf{fix}\ \mathbf{x} \le e\ \mathsf{is}\ e$

Figure 1. Syntax of core Datafun

[1] The height of a poset is the cardinality of its largest chain (totally-ordered subset).

[2] Technically, the partial orderings on these types form *join-semilattices with a least element*. For brevity's sake, we call these structures simply "semilattices."

terms	e	$::=$	$\ldots \mid \{\vec{e}\} \mid \{e \mid \mathcal{L}\} \mid \bigvee(\mathcal{L})\ e$ $\mathcal{C}\ \vec{e} \mid \mathsf{case}\ e\ \mathsf{of}\ [p \to e]^*$
patterns	p	$::=$	$_ \mid x \mid !e \mid (p, p) \mid \mathsf{true} \mid \mathsf{false} \mid \mathcal{C}\ \vec{p}$
constructors	$\mathcal{C}$		are abstract identifiers
loops	$\mathcal{L}$	$::=$	$\mathcal{L}, \mathcal{L} \mid p \in e \mid e$

$$
\begin{array}{rcl}
\{\} & \xrightarrow{\text{expand}} & \varepsilon \\
\{e, \vec{e_i}\} & \xrightarrow{\text{expand}} & \{e\} \vee \{\vec{e_i}\} \\
\{e \mid \mathcal{L}\} & \xrightarrow{\text{expand}} & \bigvee(\mathcal{L})\ \{e\} \\
\bigvee(\mathcal{L}_1, \mathcal{L}_2)\ e & \xrightarrow{\text{expand}} & \bigvee(\mathcal{L}_1)\ \bigvee(\mathcal{L}_2)\ e \\
\bigvee(p \in e_1)\ e_2 & \xrightarrow{\text{expand}} & \bigvee(x \in e_1)\ \mathsf{case}\ x\ \mathsf{of}\ p \to e_2;\ _ \to \varepsilon \\
\bigvee(e_1)\ e_2 & \xrightarrow{\text{expand}} & \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ \varepsilon
\end{array}
$$

Figure 2. Syntax sugar

$$
\begin{array}{lcl}
\mathsf{case}\ e_1\ \mathsf{of}\ \ x \to e_3;\ \ _ \to e_4 & \xrightarrow{\text{expand}} & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_3 \\
\mathsf{case}\ e_1\ \mathsf{of}\ \ !e_2 \to e_3;\ \ _ \to e_4 & \xrightarrow{\text{expand}} & \mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4 \\
\mathsf{case}\ e_1\ \mathsf{of}\ \ _ \to e_3;\ \ _ \to e_4 & \xrightarrow{\text{expand}} & e_3 \\
\mathsf{case}\ e_1\ \mathsf{of}\ \ (p, p') \to e_3;\ \ _ \to e_4 & \xrightarrow{\text{expand}} & \mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ \mathsf{case}\ x\ \mathsf{of}\ \ p \to (\mathsf{case}\ y\ \mathsf{of}\ p' \to e_3;\ _ \to e_4);\ \ _ \to e_4 \\
\mathsf{case}\ e_1\ \mathsf{of}\ \ \mathsf{in}_1\ p \to e_3;\ \ _ \to e_4 & \xrightarrow{\text{expand}} & \mathsf{case}\ e_1\ \mathsf{of}\ \ \mathsf{in}_1\ x \longrightarrow (\mathsf{case}\ x\ \mathsf{of}\ p \to e_3;\ _ \to e_4);\ \ \mathsf{in}_2\ y \longrightarrow e_4 \\
\mathsf{case}\ e_1\ \mathsf{of}\ \ \mathsf{in}_2\ p \to e_3;\ \ _ \to e_4 & \xrightarrow{\text{expand}} & \mathsf{case}\ e_1\ \mathsf{of}\ \ \mathsf{in}_1\ x \longrightarrow e_4;\ \ \mathsf{in}_2\ y \longrightarrow (\mathsf{case}\ y\ \mathsf{of}\ p \to e_3;\ _ \to e_4)
\end{array}
$$

Figure 3. Pattern matching expansion

If e_2 is a set, for example, this provides the set type's monadic "bind" operation. For example, $\bigvee(x \in \{1, 2, 3\})\ \{10 \cdot x, x^2\}$ denotes the set $\{1, 4, 9, 10, 20, 30\}$.

3. Examples

For purposes of these examples, we use a simple Haskell-like syntax for top-level type and function definitions. We also permit ourselves infix notation, let-binding, n-ary tuples, n-ary sum types with named constructors, and a restricted form of pattern-matching (including non-linear patterns), and additional syntax sugar given

$$
\begin{array}{rcl}
\neg & : & 2 \to 2 \\
= & : & \underset{eq}{A} \to \underset{eq}{A} \to 2 \\
\le & : & \underset{eq}{A} \to \underset{eq}{A} \xrightarrow{+} 2 \\
\mathsf{range} & : & \mathbb{N} \to \mathbb{N} \xrightarrow{+} \{\mathbb{N}\} \\
\mathsf{length} & : & \mathsf{str} \to \mathbb{N} \\
\mathsf{substring} & : & \mathsf{str} \to \mathbb{N} \to \mathbb{N} \to \mathsf{str}
\end{array}
$$

Figure 4. Primitive functions and their type schemes

in Figures 2 and 3. These figures also show how to expand the sugar into our core language. Full expansion for case-analysis is complicated, so we include only the fragment for expressions of the form $\mathsf{case}\ e_1\ \mathsf{of}\ p \to e_2;\ _ \to e_3$, as that is all we use in this paper.

All of these conveniences are supported (with slightly different concrete syntax) in our implementation.

For clarity, we set the names of top-level variables in sans-serif; discrete variables in `script` or *italic* (for long variable names); and monotone variables in **bold**. Although Datafun as presented does not have polymorphism, we give our examples their most general possible type schemes.

3.1 Filtering, mapping, and cross products

Armed with the syntactic sugar given in Figure 2, basic set operations such as map, filter, and cross-product are easy first examples:

$$
\begin{array}{l}
\mathsf{map}\ :\ (A \to B) \to \{A\} \xrightarrow{+} \{B\} \\
\mathsf{map}\ f\ \mathbf{A} = \{f\ x \mid x \in \mathbf{A}\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{filter}\ :\ (A \to 2) \xrightarrow{+} \{A\} \xrightarrow{+} \{A\} \\
\mathsf{filter}\ \mathbf{f}\ \mathbf{A} = \{x \mid x \in \mathbf{A}, \mathbf{f}\ x\}
\end{array}
$$

$$
\begin{array}{l}
(\times)\ :\ \{A\} \xrightarrow{+} \{B\} \xrightarrow{+} \{A \times B\} \\
\mathbf{A} \times \mathbf{B} = \{(a, b) \mid a \in \mathbf{A}, b \in \mathbf{B}\}
\end{array}
$$

Worth noting here are the subtleties of monotonicity typing. For example, map is not monotone in its function argument, while filter is. Recalling that sets are ordered by inclusion, this is straightforward enough — observe, for example, that:

$$
\begin{array}{rcl}
\mathsf{map}\ (\le 0)\ \{0, 1\} & \not\subseteq & \mathsf{map}\ (\le 1)\ \{0, 1\} \\
\mathsf{filter}\ (\le 0)\ \{0, 1\} & \subseteq & \mathsf{filter}\ (\le 1)\ \{0, 1\}
\end{array}
$$

However, it is perhaps unclear how Datafun's type system "knows" filter is monotone in $\mathbf{f}$. In brief, Datafun knows that application $(\mathbf{f}\ x)$ is monotone in the function, and moreover, testing a boolean guard $(\mathbf{f}\ x)$ in a set-comprehension such as $\{x \mid x \in \mathbf{A}, \mathbf{f}\ x\}$ is monotone in the guard expression. The full explanation is in Section 4.

3.2 Equality, membership, and intersection

So long as the type of a set's elements supports equality, we can test whether the set contains a value x as follows:

$$
\begin{array}{l}
(\in?)\ :\ \underset{eq}{A} \to \{\underset{eq}{A}\} \xrightarrow{+} 2 \\
x \in?\ \mathbf{A} = \bigvee(y \in \mathbf{A})\ x = y
\end{array}
$$

The expression $\bigvee(y \in \mathbf{A})\ x = y$ takes the least upper bound, at boolean type, for every $y \in \mathbf{A}$, of the value of $x = y$. Since booleans are ordered false < true, "least upper bound" is simply logical disjunction!

Similarly, we can define set intersection by testing for equality:

$$
\begin{array}{l}
(\cap)\ :\ \{\underset{eq}{A}\} \xrightarrow{+} \{\underset{eq}{A}\} \xrightarrow{+} \{\underset{eq}{A}\} \\
\mathbf{A} \cap \mathbf{B} = \{x \mid x \in \mathbf{A}, y \in \mathbf{B}, x = y\}
\end{array}
$$

However, explicitly binding multiple variables only to test for equality can become tedious, so we support a form of *equality patterns*. The grammar of patterns includes the form $!e$, which means the term at that position equals the value of e. So we can indicate that a pattern must have an equal value at different positions by binding the first one to a variable x, and then marking later positions with a $!x$. Now, the intersection can be written as:

$$(\cap) : \{\underset{eq}{A}\} \xrightarrow{+} \{\underset{eq}{A}\} \xrightarrow{+} \{\underset{eq}{A}\}$$
$$\mathbf{A} \cap \mathbf{B} = \{x \mid x \in \mathbf{A}, !x \in \mathbf{B}\}$$

Since $!x$ implies the use of an equality test, the condition that the set's element type support equality remains in force.

3.3 Composition of relations

One extremely useful operator it is convenient to define using nonlinear pattern matching is composition of finite relations (that is, sets of pairs):

$$(\bullet) : \{A \times \underset{eq}{B}\} \xrightarrow{+} \{\underset{eq}{B} \times C\} \xrightarrow{+} \{A \times C\}$$
$$\mathbf{R} \bullet \mathbf{S} = \{(a, c) \mid (a, b) \in \mathbf{R}, (!b, c) \in \mathbf{S}\}$$

This already demonstrates a capability Datafun has that Datalog does not: defining operators over relations. A Datalog program defining binary predicates `r` and `s` which wished to compose those predicates would have to define a new top-level predicate:

```
r(X,Y) :- (...).
s(X,Y) :- (...).
rs(A,C) :- r(A,B), s(B,C).
```

In Datafun, we simply define $(\bullet)$ and use it inline as needed. We shall see the use of this in later examples.

3.4 Transitive closure

Consider the following Datalog program, authored perhaps by a J.R.R. Tolkien aficionado wishing to trace the geneologies of their favorite work, *The Silmarillion*:

```
parent(earendil, elrond).
parent(elrond, arwen).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Z) :- ancestor(X, Y), ancestor(Y, Z).
```

This defines a binary `parent` relation, along with its transitive closure, `ancestor`. The Datafun equivalent is:

$$\begin{aligned}
&\textbf{data}\ \mathsf{person} = \text{EÄRENDIL} \mid \text{ELROND} \mid \text{ARWEN}\\
&\mathsf{parent},\ \mathsf{ancestor} : \{\mathsf{person} \times \mathsf{person}\}\\
&\mathsf{parent} = \{(\text{EÄRENDIL}, \text{ELROND}), (\text{ELROND}, \text{ARWEN})\}\\
&\mathsf{ancestor} = \mathsf{fix}\ \mathbf{X}\ \mathsf{is}\ \mathsf{parent} \vee (\mathbf{X} \bullet \mathbf{X})
\end{aligned}$$

The type person represents the domain of our parent and ancestor relations. parent is simply a list of parent-child pairs. ancestor is where the action is at: since the Datalog predicate `ancestor` is defined recursively, ancestor is defined as a least fixed point — in this case, of the the equation

$$\mathbf{X} = \mathsf{parent} \vee (\mathbf{X} \bullet \mathbf{X})$$

Informally, we may read this as stating that a pair is in $\mathbf{X}$ if it is in either parent or the composition of $\mathbf{X}$ with itself. This requires that $\mathbf{X}$ contain the transitive closure of parent. And since we take the *least* fixed point of this equation, ancestor contains *exactly* the transitive closure of parent. Voilà!

3.4.1 Transitive closure with an upper bound

The preceding explanation glosses over a critical requirement: fix may only be used at *finite semilattice eqtypes*. ancestor has type $\{\mathsf{person} \times \mathsf{person}\}$. Does this suffice? It's certainly a semilattice, since it's a set type. Since person is effectively a sum of units, it supports equality, and sets and products of eqtypes are themselves eqtypes. Likewise, person is finite, and products and sets of finite types are themselves finite.

So we are in the clear, but in general the restriction of fix to finite types can be quite limiting. So Datafun provides a more general way to take a fixed-point: specify an *upper bound* which the fixed point may not exceed. For this we write $(\mathsf{fix}\ \mathbf{x} \leq e_\top\ \mathsf{is}\ e)$, where $e_\top$ is our upper bound.

Suppose, for example, we represent our *dramatis personae* as strings (an infinite type). We may write:

$$\begin{aligned}
&\mathsf{persons} : \{\mathsf{str}\}\\
&\mathsf{persons} = \{\texttt{"eärendil"}, \texttt{"elrond"}, \texttt{"arwen"}\}\\
&\mathsf{parent}, \mathsf{ancestor} : \{\mathsf{str} \times \mathsf{str}\}\\
&\mathsf{parent} = \{(\texttt{"eärendil"}, \texttt{"elrond"}), (\texttt{"elrond"}, \texttt{"arwen"})\}\\
&\mathsf{ancestor} = \mathsf{fix}\ \mathbf{X} \leq (\mathsf{persons} \times \mathsf{persons})\ \mathsf{is}\ \mathsf{parent} \vee (\mathbf{X} \bullet \mathbf{X})
\end{aligned}$$

Instead of a person type, we have persons *set*, which we use to construct an upper bound on our fixed-point: (persons × persons), the complete binary relation. Since all strings in parent are in persons, the transitive closure of parent cannot exceed the bound.

However, this invariant is left to the programmer to check. What if a sloppy programmer should mistakenly include a person in parent not present in persons? More generally, what if the fixed point $(\mathsf{fix}\ \mathbf{x} \leq e_\top\ \mathsf{is}\ e)$ is trying to compute exceeds $e_\top$? (Or indeed, no such fixed point exists?)

In that case, the value of $(\mathsf{fix}\ \mathbf{x} \leq e_\top\ \mathsf{is}\ e)$ is *clamped* to the upper bound $e_\top$. This ensures Datafun programs terminate even in the presence of sloppy programmers, and although they may not have the value you expect, that value is at least predictable.

3.4.2 Generic transitive closure

Thus far we have only considered taking the transitive closure of a relation we have already defined. But consider: for any finite eqtype $\underset{fin}{A}$, we may write:

$$\mathsf{trans} : \{\underset{fin}{A} \times \underset{fin}{A}\} \xrightarrow{+} \{\underset{fin}{A} \times \underset{fin}{A}\}$$
$$\mathsf{trans}\ \mathbf{E} = \mathsf{fix}\ \mathbf{X}\ \mathsf{is}\ \mathbf{E} \vee (\mathbf{X} \bullet \mathbf{X})$$

Similarly, for any eqtype $\underset{eq}{A}$, we may write:

$$\mathsf{trans} : \{\underset{eq}{A}\} \xrightarrow{+} \{\underset{eq}{A} \times \underset{eq}{A}\} \xrightarrow{+} \{\underset{eq}{A} \times \underset{eq}{A}\}$$
$$\mathsf{trans}\ \mathbf{V}\ \mathbf{E} = \mathsf{fix}\ \mathbf{S} \leq (\mathbf{V} \times \mathbf{V})\ \mathsf{is}\ \mathbf{E} \vee (\mathbf{S} \bullet \mathbf{S})$$

In this way, we can abstract away from choice of underlying relation and define transitive closure generically. Using functions as a means of abstraction is of course familiar and unremarkable to functional programmers, but it is simply not possible in Datalog.

3.5 CYK parsing

Parsing can be understood logically, with a parse tree representing a proof that a certain string belongs to a language described by a context-free grammar. As a result, it is possible to formulate parsing in terms of proof search (Shieber et al. 1995). One of the simplest algorithms for parsing context free grammars is the Cocke-Younger-Kasami (CYK) algorithm for parsing with grammars in Chomsky normal form.[3] Given a grammar G, we begin by introducing a family of predicates (sometimes called *facts* or *items*) $A(i, j)$, with one A for each nonterminal, and i and j representing indices into a string. Given a word w, we write $w[i, n]$ for the n-element substring of w

[3] In Chomsky normal form, each production is of the form $A \rightarrow B \cdot C$ or $A \rightarrow \vec{a}$, with A, B, C ranging over nonterminals, and $\vec{a}$ over strings of terminals.

beginning at position i. Then, we can specify the CYK algorithm with the following two inference rules:

$$\frac{B(i, j) \qquad C(j, k) \qquad (A \to B\ C) \in G}{A(i, k)}$$

$$\frac{(A \to \vec{a}) \in G \qquad w[i, n] = \vec{a}}{A(i, i + n)}$$

Then, the predicate $A(i, j)$ means that A is derivable from the substring of w running from i to j, and so the whole word w is derivable from the start symbol S if $S(0, \mathit{length}\ w)$ is derivable.

In Datafun, this rule-based description of the algorithm can be transliterated almost directly into code. We begin by introducing a few basic types.

```
type sym = str
data rule = STRING str | CONCAT sym sym
type grammar = {sym × rule}
type fact = sym × ℕ × ℕ
```

The sym type is a type synonym representing nonterminal names with strings. The rule type is the type of the right-hand-sides of productions in Chomsky normal form – either a string, or a pair of nonterminals. A grammar is just a set of productions – a set of pairs of nonterminals paired with their rules. The type fact is the type representing the atomic facts derived by the CYK inference system – they are triples of the rulename, the start position, and the end position.

With these types in hand, we can write the CYK algorithm as a fixed point computation. In fact, it is convenient to break it into two pieces, by first defining the function whose fixed point we take. So we can write down the iter function, which represents one step of the fixed point iteration.

```
iter : str → grammar →⁺ {fact} →⁺ {fact}
iter text G chart =
      {(a, i, k) | (a, CONCAT b c) ∈ G,
                   (!b, i, j) ∈ chart,
                   (!c, !j, k) ∈ chart}
   ∨ {(a, i, i + length s)
      | (a, STRING s) ∈ G,
        i ∈ range 0 (n − length s),
        s = substring text i (i + length s)}
```

This function works by taking a string *text* and a grammar **G**, and then taking a set of facts **chart**, and taking a union. The first clause is a set comprehension, saying that we return (a, i, k) if (b, i, j) and (c, j, k) are in **chart** – this corresponds to applications of the first rule. The second clause corresponds to the second rule above, saying that $(a, i, i + \text{length } s)$ is a generated fact if s is a substring of *text* at position i.

We can then use iter to implement the parse function.

```
parse : str → grammar →⁺ {sym}
parse text G =
  let n = length text
      bound = {(a, i, j) | (a, _) ∈ G,
                           i ∈ range 0 n,
                           j ∈ range i n}
      chart = fix C ≤ bound is iter text G C
  in {a | (a, 0, !n) ∈ chart}
```

This function just takes the fixed point of iter – almost. Because facts are triples sym $\times \mathbb{N} \times \mathbb{N}$, sets of facts may in general grow unboundedly. To ensure termination, we construct a set **bound** to bound the sets of facts we consider in our fixed point computation, by bounding the symbols to names found in the grammar **G**, and the indices to positions of the string. Since all of these are finite, we know that the computation of **chart** as a bounded fixed point will terminate. Then, having computed the fixed point, we can check chart to see if $(a, 0, \text{length } \mathit{text})$ is derivable.

There are three things worth noting about this program. First, it is not expressible in Datalog. Because Datalog provides no way to represent a *grammar* as a piece of data (it's compound, not an atom), there is simply no way in Datalog to express a *generic* parser taking a grammar as an input. This demonstrates one of the key benefits of moving to a functional language like Datafun.

Moreover, Datalog programs must be *constructor-free*, to ensure all relations are finite. Primitives such as range and substring violate this restriction (as relations, they are infinite); it is not immediately obvious that Datalog programs extended with these primitives remain terminating. Our use of bounded fixed-points to guarantee termination is robust under such extensions; as long as all primitive functions are total, Datafun programs always terminate.

Finally, having computed a set via a fixed point, we can test whether or not an element is in that set *or not* – the ability to test for negative information after the fixed point computation completes corresponds to a use of stratified negation in Datalog.

3.6 Dataflow analysis

In this section, we show how some simple dataflow analyses can be expressed in Datafun. We begin with the types in these programs.

```
type var = str
type label = ℕ
data oper = EQ | LE | ADD | SUB | MUL | DIV
data atom = VAR var | NUM ℕ
data expr = ATOM atom | APPLY oper atom atom
data stmt = ASSIGN var expr | IF expr label label
type program = {label × stmt}
```

The basic idea is that we represent a program as a kind of control flow graph. Each node of this graph has a label, which is a natural number, and contains a statement of type stmt, which is either an assignment of an expression (of type expr) to a variable (of type var), or a conditional jump. A program is then just the set of nodes – i.e., a set of label, statement pairs – with the invariant that the relation is functional (i.e., if (l, s) and (l, s') are both in a program, then $s = s'$).

In what follows, we use a few trivial functions whose definitions are omitted for space reasons.

```
labels : program → {label}
vars : program → {var}
uses : stmt → {var}
defines : stmt → {var}
```

The labels function returns the set of labels in a program. The vars function returns the set of variables used in a program (both in expressions and as targets for assignments). The uses function returns the set of variables used by the expressions in a statement. The defines function returns the set of variables defined by a statement (i.e., at most one variable – the target of the assignment).

Given a program, we define the 1-step control flow graph with the flow function.

```
type flow = {label × label}
flow : program → flow
flow c = ⋁((i, s) ∈ c)
           case s of IF _ j k → {(i, j), (i, k)}
                   | _ → {(i, i + 1) | i + 1 ∈? labels c}
```

It says that if (i, s) is a node of the program, then if s is a conditional jump IF _ j k, then control can flow from i to j, and from i to k – i.e., we add both (i, j) and (i, k) to the set of edges. Otherwise, it's an assignment, and control flows to the next statement (i.e., we add $(i, i + 1)$ to the set of edges).

Now, we can define liveness analysis, one of the classic "backwards" dataflow analyses. The type of live say that given a program and its flow graph, it returns a set of label/variable pairs, which determine a relation saying for each label which variables are live.

$$
\begin{array}{l}
\mathsf{live} \;:\; \mathsf{program} \to \mathsf{flow} \to \{\mathsf{label} \times \mathsf{var}\} \\
\mathsf{live}\ \textit{code flow} = \\
\quad \mathsf{fix}\ \mathbf{Live} \leq \mathsf{labels}\ \textit{code} \times \mathsf{vars}\ \textit{code}\ \mathsf{is} \\
\quad \bigvee((i, \textit{stmt}) \in \textit{code}) \\
\quad\quad (\quad \{(i, v) \mid v \in \mathsf{uses}\ \textit{stmt}\} \\
\quad\quad \vee \{(i, v) \mid (!i, j) \in \textit{flow}, \\
\quad\quad\quad\quad (!j, v) \in \mathbf{Live}, \\
\quad\quad\quad\quad \neg(v \in ?\ \mathsf{defines}\ \textit{stmt})\})
\end{array}
$$

For a statement *stmt* at label i, we say that the variable v is live at i if v is used by *stmt*. The variable v is also live at i if control flows from i to j, and and v is live at j, assuming that *stmt* isn't a definition site for v.

When computing this analysis, we again need to use a bounded fixed point, which we do by taking the Cartesian product of the labels and variables occuring in the program.

Next, we give one of the classic forwards dataflow analyses, reaching definitions. This analysis is used to figure out whether an assignment (a "definition") can influence the value of later expressions or not.

$$
\begin{array}{l}
\mathsf{reachingDefinitions} \;:\; \mathsf{program} \to \mathsf{flow} \to \{(\mathsf{label} \times \mathsf{var}) \times \mathsf{label}\} \\
\mathsf{reachingDefinitions}\ \textit{code flow} = \\
\quad \mathsf{fix}\ \mathbf{RD} \leq (\mathsf{labels}\ \textit{code} \times \mathsf{vars}\ \textit{code}) \times \mathsf{labels}\ \textit{code}\ \mathsf{is} \\
\quad \bigvee((i, \textit{stmt}) \in \textit{code}) \\
\quad\quad (\quad \{((i, v), i) \mid v \in \mathsf{defines}\ \textit{stmt}\} \\
\quad\quad \vee \{((l, v), i) \mid (j, !i) \in \textit{flow}, \\
\quad\quad\quad\quad ((l, v), !j) \in \mathbf{RD}, \\
\quad\quad\quad\quad \neg(v \in ?\ \mathsf{defines}\ \textit{stmt})\})
\end{array}
$$

We define a function reachingDefinitions which takes a program and a set of flows as arguments, and returns a relation of type $\{(\mathsf{label} \times \mathsf{var}) \times \mathsf{label}\}$. An entry $((l, v), i)$ in this relation means the definition of v at l reaches program point i.

This is then computed as a fixed point of two clauses. First, if there is a definition v at program point i, then i is reached by that definition. Second, if (l, v) reaches j, and j flows to i, then (l, v) reaches i as long as v is not re-defined at i.

As Whaley et al. (2005) observed, Datalog makes it very easy to express dataflow analyses, and it is similarly easy in Datafun.

4. Type system

Datafun's typing judgment $\Delta; \Gamma \vdash e : A$ is defined by the inference rules given in Figure 5. We gloss $\Delta; \Gamma \vdash e : A$ as follows: "expression e has type A using variables from $\Delta \cup \Gamma$, and moreover the value of e is *monotone* with respect to the variables in Γ".

The context Δ types discrete variables; Γ, monotone variables. Both admit the usual structural rules of exchange, weakening, and contraction. Variables from either context may be used freely (rules VAR, VAR$^+$).

For clarity, we also give typing rules for our syntax sugar, in Figure 6. These use the auxilliary judgments $\Delta; \Gamma \vdash p : A \Rightarrow \Delta'$, which can be read as saying that "in the contexts Δ and Γ, the pattern p typechecks at type A, binding the variables in Δ'"; and

$$\frac{\Delta; \Gamma \vdash \mathcal{L} \Rightarrow \Delta' \quad \Delta, \Delta'; \Gamma \vdash e : L}{\Delta; \Gamma \vdash \bigvee(\mathcal{L})\ e : L}$$

$$\frac{\Delta; \Gamma \vdash \mathcal{L} \Rightarrow \Delta' \quad \Delta, \Delta'; \Gamma \vdash e : A}{\Delta; \Gamma \vdash \{e \mid \mathcal{L}\} : \{A\}}$$

$$\frac{\Delta; \cdot \vdash e : A \quad \Delta; \Gamma \vdash p : A \Rightarrow \Delta' \quad \Delta, \Delta'; \Gamma \vdash e_1 : C \quad \Delta; \Gamma \vdash e_2 : C}{\Delta; \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ p \to e_1;\ _ \to e_2 : C}$$

$$\frac{\Delta; \Gamma \vdash \mathcal{L}_1 \Rightarrow \Delta_1 \quad \Delta; \Gamma \vdash \mathcal{L}_2 \Rightarrow \Delta_2}{\Delta; \Gamma \vdash \mathcal{L}_1, \mathcal{L}_2 \Rightarrow \Delta_1, \Delta_2} \qquad \frac{\Delta; \Gamma \vdash e : 2}{\Delta; \Gamma \vdash e \Rightarrow \cdot}$$

$$\frac{\Delta; \Gamma \vdash e : \{A\} \quad \Delta; \Gamma \vdash p : A \Rightarrow \Delta}{\Delta; \Gamma \vdash p \in e \Rightarrow \Delta'} \qquad \frac{}{\Delta; \Gamma \vdash x : A \Rightarrow x : A}$$

$$\frac{}{\Delta; \Gamma \vdash _ : A \Rightarrow \cdot}$$

$$\frac{\Delta; \Gamma \vdash p_1 : A_1 \Rightarrow \Delta_1 \quad \Delta; \Gamma \vdash p_1 : A_2 \Rightarrow \Delta_2}{\Delta; \Gamma \vdash (p_1, p_2) : A_1 \times A_2 \Rightarrow \Delta_1, \Delta_2}$$

$$\frac{\Delta; \Gamma \vdash e : \underset{eq}{A}}{\Delta; \Gamma \vdash !e : \underset{eq}{A} \Rightarrow \cdot} \qquad \frac{\Delta; \Gamma \vdash p : A_i \Rightarrow \Delta'}{\Delta; \Gamma \vdash \mathsf{in}_i\ p : A_1 + A_2 \Rightarrow \Delta'}$$

Figure 6. Typing rules for syntax sugar

$\Delta; \Gamma \vdash \mathcal{L} \Rightarrow \Delta'$, which says that "in the contexts Δ and Γ, the comprehension clauses in $\mathcal{L}$ bind the variables in Δ'".

4.1 Functions and application

Two function types require two function introduction rules: the discrete λ and the monotone λ^+. These simply introduce variables into their respective contexts. Monotone function application APP$^+$ is perfectly standard, but discrete function application APP has a pecularity: the argument e_2 gets an *empty* monotone context.

To understand why, recall our gloss: the application $e_1\ e_2$ must be monotone in Γ. But e_1 is a discrete, and in general *non-monotone*, function $A \to B$: there is no guarantee that it respect any order on its argument. (Suppose, for example, e_2 were some monotone variable $\mathbf{x} : A \in \Gamma$.) We work around this scoff-law behavior on e_1's part by ensuring its argument e_2 is *constant* with respect to Γ—which we accomplish by simply prohibiting e_2 from using any of Γ's variables.

This technique of *wiping clean* the monotone context to guarantee constancy[4] of a subterm recurs in several other rules. Readers familiar with linear logic's ! comonad (Girard 1987) or with judgmental formulations of modal logics of necessity (Pfenning and Davies 2001) may notice a feeling of *déjà vu*; indeed, there is a hidden comonad at work here. But we are getting ahead of ourselves. For more on that, turn to Section 5.

4.2 Products and sums

The pairing and projection rules, PAIR and π, are completely standard, as is the IN rule for sum introduction. Sum elimination, however, splits into two rules, CASE and CASE$^+$. CASE$^+$ requires its branches to be monotone in the variable $\mathbf{x}$ it introduces, and consequently its subject e is permitted access to the monotone context Γ. CASE, however, analyses its subject e as a constant

[4] Wherever we write "constant" in this section, substitute "constant with respect to the monotone context". The discrete context is never "wiped clean", and behaves entirely as it would in a simply-typed λ-calculus.

$$\frac{x : A \in \Delta}{\Delta; \Gamma \vdash x : A}\ \text{VAR} \qquad \frac{\mathbf{x} : A \in \Gamma}{\Delta; \Gamma \vdash \mathbf{x} : A}\ \text{VAR}^{+} \qquad \frac{\Delta, x : A; \Gamma \vdash e : B}{\Delta; \Gamma \vdash \lambda x.\, e : A \to B}\ \lambda \qquad \frac{\Delta; \Gamma \vdash e_1 : A \to B \quad \Delta; \cdot \vdash e_2 : A}{\Delta; \Gamma \vdash e_1\ e_2 : B}\ \text{APP}$$

$$\frac{\Delta; \Gamma, \mathbf{x} : A \vdash e : B}{\Delta; \Gamma \vdash \lambda \mathbf{x}.\, e : A \xrightarrow{+} B}\ \lambda^{+} \qquad \frac{\Delta; \Gamma \vdash e_1 : A \xrightarrow{+} B \quad \Delta; \Gamma \vdash e_2 : A}{\Delta; \Gamma \vdash e_1\ e_2 : B}\ \text{APP}^{+} \qquad \frac{\Delta; \Gamma \vdash e_i : A_i}{\Delta; \Gamma \vdash (e_1, e_2) : A_1 \times A_2}\ \text{PAIR} \qquad \frac{\Delta; \Gamma \vdash e : A_1 \times A_2}{\Delta; \Gamma \vdash \pi_i\ e : A_i}\ \pi$$

$$\frac{\Delta; \Gamma \vdash e : A_i}{\Delta; \Gamma \vdash \mathsf{in}_i\ e : A_1 + A_2}\ \text{IN} \qquad \frac{\Delta; \cdot \vdash e : A_1 + A_2 \quad \Delta, x : A_i; \Gamma \vdash e_i : C}{\Delta; \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{in}_1\ x \to e_1;\ \mathsf{in}_2\ x \to e_2 : C}\ \text{CASE} \qquad \frac{\Delta; \Gamma \vdash e : A_1 + A_2 \quad \Delta; \Gamma, \mathbf{x} : A_i \vdash e_i : C}{\Delta; \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{in}_1\ \mathbf{x} \to e_1;\ \mathsf{in}_2\ \mathbf{x} \to e_2 : C}\ \text{CASE}^{+}$$

$$\frac{}{\Delta; \Gamma \vdash \mathsf{true} : 2}\ \text{TRUE} \qquad \frac{}{\Delta; \Gamma \vdash \mathsf{false} : 2}\ \text{FALSE} \qquad \frac{\Delta; \cdot \vdash e : 2 \quad \Delta; \Gamma \vdash e_i : A}{\Delta; \Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : A}\ \text{IF} \qquad \frac{\Delta; \Gamma \vdash e : 2 \quad \Delta; \Gamma \vdash e_1 : L}{\Delta; \Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ \varepsilon : L}\ \text{IF}^{+}$$

$$\frac{}{\Delta; \Gamma \vdash \varepsilon : L}\ \varepsilon \qquad \frac{\Delta; \Gamma \vdash e_i : L}{\Delta; \Gamma \vdash e_1 \vee e_2 : L}\ \vee \qquad \frac{\Delta; \cdot \vdash e : A}{\Delta; \Gamma \vdash \{e\} : \{A\}}\ \{\} \qquad \frac{\Delta; \Gamma \vdash e_1 : \{A\} \quad \Delta, x : A; \Gamma \vdash e_2 : L}{\Delta; \Gamma \vdash \bigvee(x \in e_1)\ e_2 : L}\ \bigvee$$

$$\frac{\Delta; \Gamma, \mathbf{x} : L \vdash e : \underset{fin}{L}}{\Delta; \Gamma \vdash \mathsf{fix}\ \mathbf{x}\ \mathsf{is}\ e : \underset{fin}{L}}\ \text{FIX} \qquad \frac{\Delta; \Gamma \vdash e_1 : \underset{eq}{L} \quad \Delta; \Gamma, \mathbf{x} : \underset{eq}{L} \vdash e_2 : \underset{eq}{L}}{\Delta; \Gamma \vdash \mathsf{fix}\ \mathbf{x} \leq e_1\ \mathsf{is}\ e_2 : \underset{eq}{L}}\ \text{FIX}_{\leq}$$

Figure 5. Typing rules for core Datafun

— wiping clean its monotone context — and thus is allowed to introduce the variable x into the discrete context Δ.

4.3 Booleans

While TRUE and FALSE are straightforward, there are two rules for boolean elimination, IF and IF^{+}. This is because in Datafun, 1 plus 1 does not equal 2: booleans are *not* a sum of units.[5] At the type $1+1$, $\mathsf{in}_1\langle\rangle$ and $\mathsf{in}_2\langle\rangle$ are incomparable. But in Datafun, $\mathsf{true} > \mathsf{false}$. Therefore, to eliminate a boolean in a monotone fashion, one must ensure one's *then*-branch is always greater than one's *else*-branch.

Thus Datafun has two if rules. First, IF, where the boolean e being analysed is constant (has an empty monotone context), and so the branches e_1, e_2 may be arbitrary expressions.

Second, IF^{+}, where the subject e has full access to Γ, but the if-expression must have *semilattice type*, and the *else*-branch is constrained to be ε — the least value, thus smaller than e_1.

This is a conservative approach: there are many semantically monotone, but untypeable, if-terms. However, it is complete for semilattice types, for in that case ($\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$) may be rewritten ($e_2 \vee \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ \varepsilon$); as long as $e_1 \geq e_2$ and so $e_2 \vee e_1 = e_1$, this will not change the meaning of the expression, only (potentially) its execution efficiency.

Thus the only meaningful restriction here is to semilattice types. In practice, we have yet to find a case where this is problematic.

4.4 Semilattices and sets

The semilattice ε and $\vee$ operations are typed by the rules of the same name. As $\vee$ is monotone, its arguments have full access to the monotone context Γ.

Recall that sets are ordered by inclusion: although $2 \leq 3$, nonetheless $\{2\} \not\leq \{3\}$. For this reason the rule $\{\}$ for constructing a singleton set $\{e\}$ wipes clean its element e's monotone context. Datafun does not need empty-set or union operators, since ε and $\vee$ generalize them.

Finally, we come to $\bigvee$, the set-comprehension rule. This rule has the flavor of a monadic "bind" operation, but generalized to a result of any semilattice type. This operation is naturally monotone both in the set e_1 being iterated over and in the expression e_2 which we are taking the least upper bound of. Since sets are ordered by inclusion regardless of the ordering on their elements, e_2 is not required to be monotone in the variable x.

[5] For simplicity, we have omitted the unit type 1 from our presentation of Datafun, but it is easy enough to imagine including it.

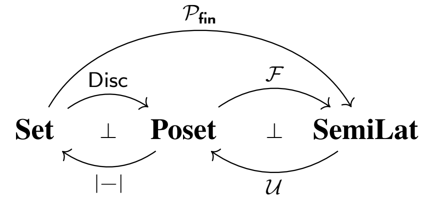

Figure 7. Semantic categories of Datafun

$\lvert P \rvert$	Underlying set of the poset P
$\mathbb{S}$	Set of strings
$\mathbf{1}$	One-element poset $\{\langle\rangle\}$
$\mathbf{2}$	Two-element poset $\{\mathsf{ff}, \mathsf{tt}\}$, with $\mathsf{ff} < \mathsf{tt}$
$\mathbb{N}_{\leq}$	The naturals $\mathbb{N}$, as a (totally ordered) poset
$P + Q$	Disjointly-ordered poset on disjoint union of P, Q
$P \times Q$	Pointwise poset on pairs of Ps and Qs
$P \Rightarrow Q$	Pointwise poset on monotone maps $\mathsf{Poset}(P, Q)$
$\mathcal{F}\ P$	Free semilattice on a poset P
$\mathcal{U}\ L$	Underlying poset of a semilattice L
$\mathsf{Disc}\ A$	Discrete poset on underlying set A
$\mathcal{P}_{\mathsf{fin}}\ A$	Free semilattice on a set A; same as $\mathcal{F}\ (\mathsf{Disc}\ A)$
$\downarrow(x : P)$	The sub-poset of P below x: $\{y \in P \mid y \leq x\}$

Figure 8. Semantic notation

4.5 Fixed points

The reason Datafun tracks monotonicity is to permit taking fixed-points of monotone functions. FIX expresses exactly that. As mentioned in Section 2, however, it is limited to types of the form $\underset{fin}{L}$: finite semilattice eqtypes.

FIX$_{\leq}$ loosens this restriction by letting us take fixed points at (not-necessarily-finite) semilattice eqtypes $\underset{eq}{L}$, as long as we provide an upper bound $e_1 : \underset{eq}{L}$ which we can check we do not exceed.

Derivation | **Denotation**

$$[\![\Delta; \Gamma \vdash e : A]\!] \in \mathsf{Set}(|[\![\Delta]\!]|, \mathsf{Poset}([\![\Gamma]\!], [\![A]\!]))$$

$$\left[\!\!\left[\frac{}{x_1 : A_1, ..., x_n : A_N; \Gamma \vdash x_i : A_i}\right]\!\!\right] \delta\, \gamma = \pi_i\, \delta$$

$$\left[\!\!\left[\frac{}{\Delta; \mathbf{x}_1 : L_1, ..., \mathbf{x}_n : L_n \vdash \mathbf{x}_i : L_i}\right]\!\!\right] \delta\, \gamma = \pi_i\, \gamma$$

$$\left[\!\!\left[\frac{\Delta, x : A; \Gamma \vdash e : B}{\Delta; \Gamma \vdash \lambda x.\, e : A \to B}\right]\!\!\right] \delta\, \gamma = x \mapsto [\![e]\!]\, \langle \delta, x\rangle\, \gamma$$

$$\left[\!\!\left[\frac{\Delta; \Gamma, \mathbf{x} : A \vdash e : B}{\Delta; \Gamma \vdash \lambda \mathbf{x}.\, e : A \xrightarrow{+} B}\right]\!\!\right] \delta\, \gamma = x \mapsto [\![e]\!]\, \delta\, \langle \gamma, x\rangle$$

$$\left[\!\!\left[\frac{\Delta; \Gamma \vdash e_1 : A \to B \quad \Delta; \cdot \vdash e_2 : A}{\Delta; \Gamma \vdash e_1\ e_2 : B}\right]\!\!\right] \delta\, \gamma = [\![e_1]\!]\, \delta\, \gamma\, ([\![e_2]\!]\, \delta\, \langle\rangle)$$

$$\left[\!\!\left[\frac{\Delta; \Gamma \vdash e_1 : A \xrightarrow{+} B \quad \Delta; \Gamma \vdash e_2 : B}{\Delta; \Gamma \vdash e_1\ e_2 : B}\right]\!\!\right] \delta\, \gamma = [\![e_1]\!]\, \delta\, \gamma\, ([\![e_2]\!]\, \delta\, \gamma)$$

$$\left[\!\!\left[\frac{\Delta; \Gamma \vdash e_i : A_i}{\Delta; \Gamma \vdash (e_1, e_2) : A_1 \times A_2}\right]\!\!\right] \delta\, \gamma = \langle [\![e_1]\!]\, \delta\, \gamma,\ [\![e_2]\!]\, \delta\, \gamma\rangle$$

$$\left[\!\!\left[\frac{\Delta; \Gamma \vdash e : A_1 + A_2}{\Delta; \Gamma \vdash \pi_i\ e : A_i}\right]\!\!\right] \delta\, \gamma = \pi_i\, ([\![e]\!]\, \delta\, \gamma)$$

$$\left[\!\!\left[\frac{\Delta; \Gamma \vdash e : A_i}{\Delta; \Gamma \vdash \mathsf{in}_i\ e : A_1 + A_2}\right]\!\!\right] \delta\, \gamma = \mathsf{in}_i\, ([\![e]\!]\, \delta\, \gamma)$$

$$\left[\!\!\left[\frac{\Delta; \cdot \vdash e : A_1 + A_2 \quad \Delta, x : A_i; \Gamma \vdash e_i : B}{\Delta; \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{in}_1\, x \to e_1;\ \mathsf{in}_2\, x \to e_2 : B}\right]\!\!\right] \delta\, \gamma = \begin{cases} [\![e_1]\!]\, \langle \delta, x\rangle\, \gamma & \text{if } [\![e]\!]\, \delta\, \langle\rangle = \mathsf{in}_1\ x \\ [\![e_2]\!]\, \langle \delta, x\rangle\, \gamma & \text{if } [\![e]\!]\, \delta\, \langle\rangle = \mathsf{in}_2\ x \end{cases}$$

$$\left[\!\!\left[\frac{\Delta; \Gamma \vdash e : A_1 + A_2 \quad \Delta; \Gamma, \mathbf{x} : A_i \vdash e_i : B}{\Delta; \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{in}_1\, \mathbf{x} \to e_1;\ \mathsf{in}_2\, \mathbf{x} \to e_2 : B}\right]\!\!\right] \delta\, \gamma = \begin{cases} [\![e_1]\!]\, \delta\, \langle \gamma, x\rangle & \text{if } [\![e]\!]\, \delta\, \gamma = \mathsf{in}_1\ x \\ [\![e_2]\!]\, \delta\, \langle \gamma, x\rangle & \text{if } [\![e]\!]\, \delta\, \gamma = \mathsf{in}_2\ x \end{cases}$$

$$\left[\!\!\left[\frac{}{\Delta; \Gamma \vdash \mathsf{true} : 2}\right]\!\!\right] \delta\, \gamma = \mathsf{tt}$$

$$\left[\!\!\left[\frac{}{\Delta; \Gamma \vdash \mathsf{false} : 2}\right]\!\!\right] \delta\, \gamma = \mathsf{ff}$$

$$\left[\!\!\left[\frac{\Delta; \cdot \vdash e : 2 \quad \Delta; \Gamma \vdash e_i : A}{\Delta; \Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : A}\right]\!\!\right] \delta\, \gamma = \begin{cases} [\![e_1]\!]\, \delta\, \gamma & \text{if } [\![e]\!]\, \delta\, \langle\rangle = \mathsf{tt} \\ [\![e_2]\!]\, \delta\, \gamma & \text{if } [\![e]\!]\, \delta\, \langle\rangle = \mathsf{ff} \end{cases}$$

$$\left[\!\!\left[\frac{\Delta; \Gamma \vdash e : 2 \quad \Delta; \Gamma \vdash e_1 : L}{\Delta; \Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ \varepsilon : L}\right]\!\!\right] \delta\, \gamma = \begin{cases} [\![e_1]\!]\, \delta\, \gamma & \text{if } [\![e]\!]\, \delta\, \gamma = \mathsf{tt} \\ \varepsilon_{[\![L]\!]} & \text{if } [\![e]\!]\, \delta\, \gamma = \mathsf{ff} \end{cases}$$

$$\left[\!\!\left[\frac{}{\Delta; \Gamma \vdash \varepsilon : L}\right]\!\!\right] \delta\, \gamma = \varepsilon_{[\![L]\!]}$$

$$\left[\!\!\left[\frac{\Delta; \Gamma \vdash e_i : L}{\Delta; \Gamma \vdash e_1 \vee e_2 : L}\right]\!\!\right] \delta\, \gamma = [\![e_1]\!]\, \delta\, \gamma \vee_{[\![L]\!]} [\![e_2]\!]\, \delta\, \gamma$$

$$\left[\!\!\left[\frac{\Delta; \cdot \vdash e : A}{\Delta; \Gamma \vdash \{e\} : \{A\}}\right]\!\!\right] \delta\, \gamma = \{[\![e]\!]\, \delta\, \langle\rangle\}$$

$$\left[\!\!\left[\frac{\Delta; \Gamma \vdash e_1 : \{A\} \quad \Delta, x : A; \Gamma \vdash e_2 : L}{\Delta; \Gamma \vdash \bigvee(x \in e_1)\ e_2 : L}\right]\!\!\right] \delta\, \gamma = \bigvee \{[\![e_2]\!]\, \langle \delta, x\rangle\, \gamma \mid x \in [\![e_1]\!]\, \delta\, \gamma\}$$

$$\left[\!\!\left[\frac{\Delta; \Gamma, \mathbf{x} : \underset{fin}{L} \vdash e : \underset{fin}{L}}{\Delta; \Gamma \vdash \mathsf{fix}\ \mathbf{x}\ \mathsf{is}\ e : \underset{fin}{L}}\right]\!\!\right] \delta\, \gamma = \mathrm{lfp}\,(x \mapsto [\![e]\!]\, \delta\, \langle \gamma, x\rangle) \in [\![\underset{fin}{L}]\!]$$

$$\left[\!\!\left[\frac{\Delta; \Gamma \vdash e_1 : \underset{eq}{L} \quad \Delta; \Gamma, \mathbf{x} : \underset{eq}{L} \vdash e_2 : \underset{eq}{L}}{\Delta; \Gamma \vdash \mathsf{fix}\ \mathbf{x} \le e_1\ \mathsf{is}\ e_2 : \underset{eq}{L}}\right]\!\!\right] \delta\, \gamma = \mathrm{lfp}\left(x \mapsto \begin{cases} [\![e_2]\!]\, \delta\, \langle \gamma, x\rangle & \text{if it's} \le [\![e_1]\!]\, \delta\, \gamma \\ [\![e_1]\!]\, \delta\, \gamma & \text{otherwise} \end{cases}\right) \in \downarrow([\![e_1]\!]\, \delta\, \gamma : [\![\underset{eq}{L}]\!])$$

Figure 10. Denotations of Datafun typing derivations

$$
\begin{array}{rcl}
[\![A]\!] & \in & \mathsf{Poset}_0 \\
[\![2]\!] & = & \mathbf{2} \\
[\![\mathbb{N}]\!] & = & \mathbb{N}_{\leq} \\
[\![\mathsf{str}]\!] & = & \mathsf{Disc}\ \mathbb{S} \\
[\![A \times B]\!] & = & [\![A]\!] \times [\![B]\!] \\
[\![A + B]\!] & = & [\![A]\!] + [\![B]\!] \\
[\![A \xrightarrow{+} B]\!] & = & [\![A]\!] \Rightarrow [\![B]\!] \\
[\![A \to B]\!] & = & \mathsf{Disc}\ |[\![A]\!]| \Rightarrow [\![B]\!] \\
[\![\{A\}]\!] & = & \mathcal{P}_{\mathsf{fin}}\ |[\![A]\!]|
\end{array}
$$

$$
\begin{array}{rcl}
[\![\Delta]\!], [\![\Gamma]\!] & \in & \mathsf{Poset}_0 \\
[\![\cdot]\!] & = & \mathbf{1} \\
[\![\Delta, x : A]\!] & = & [\![\Delta]\!] \times [\![A]\!] \\
[\![\Gamma, \mathbf{x} : A]\!] & = & [\![\Gamma]\!] \times [\![A]\!]
\end{array}
$$

Figure 9. Denotations of Datafun types and contexts

5. Denotational semantics

We give a denotational semantics for Datafun in terms of three categories (Set, Poset, and SemiLat) and two adjunctions between them (see Figure 7). We present the notation we use in Figure 8; we take care to distinguish between sets and posets, and since posets are more central to our semantics, most of our notation concerns them. We take less care to distinguish posets and semilattices, since while a set can be partially ordered in many ways, a poset either *is* or *is not* a semilattice.

5.1 The category SemiLat

SemiLat is the category of join-semilattices with least elements, which we call simply “semilattices”.

Directly, a semilattice is a poset L, with a least element ε, in which any two elements a, b have a least-upper-bound $a \vee b$. From ε and $\vee$ it follows that any finite subset $X \subseteq_{\mathsf{fin}} |L|$ has a least upper bound, written $\bigvee X$.

A morphism $f \in \mathsf{SemiLat}(L, M)$ is a function from $|L|$ to $|M|$ satisfying:

$$
\begin{array}{rcl}
f(a \vee_A b) & = & f(a) \vee_B f(b) \\
f(\varepsilon_A) & = & \varepsilon_B
\end{array}
$$

SemiLat is a subcategory of Poset; every SemiLat-morphism f is monotone, since $a \leq b \iff a \vee b = b$, and so from $a \leq b$ we know $f(a) \vee f(b) = f(a \vee b) = f(b)$, thus $f(a) \leq f(b)$. Since it is a subcategory, we will typically not explicitly write the forgetful functor $\mathcal{U}\,L$ which sends semilattices to posets by forgetting the lattice structure.

5.2 Denotation of Datafun types

Datafun types and contexts denote posets as shown in Figure 9. To complete our semantics, we will need a few simple lemmas about the denotations of Datafun types. First, we need to know that our semilattice types are semilattices, and that our finite types are finite:

Lemma 1. *The denotation* $[\![L]\!]$ *of a semilattice type* L *is a semilattice.*

Lemma 2. *The poset* $[\![\underset{fin}{A}]\!]$ *denoted by a finite eqtype* $\underset{fin}{A}$ *is finite.*

Second, to show that bounded fixed-points ($\mathsf{fix}\ \mathbf{x} \leq e_\top\ \mathsf{is}\ e$) terminate, we need any possible $e_\top$ to pick out a finite-height subposet:

Lemma 3. *For any semilattice equality type* $\underset{eq}{L}$*, for any* $x \in [\![\underset{eq}{L}]\!]$*, the height of* $\downarrow(x : [\![\underset{eq}{L}]\!])$ *is finite.*

All of these are trivial to prove by induction over types and the definition of $[\![-]\!]$.

5.3 Denotation of Datafun terms

In Figure 10 we give a denotation for typing derivations with the following signature:

$$[\![\Delta; \Gamma \vdash e : A]\!] \quad \in \quad \mathsf{Set}(|[\![\Delta]\!]|, \mathsf{Poset}([\![\Gamma]\!], [\![A]\!]))$$

Colloquially, $\Delta; \Gamma \vdash e : A$ denotes a function from $[\![\Delta]\!] \times [\![\Gamma]\!]$ to $[\![A]\!]$ that must be monotone in $[\![\Gamma]\!]$ (but not in $[\![\Delta]\!]$).

Our semantics requires the following lemma regarding fixed-points of monotone functions:

Lemma 4 (Fixed points in finite-height pointed posets)**.** *Any monotone map* $f : P \to P$ *on a poset* P *of finite height with a least element* ε *has a least fixed point of the form* $f^n(\varepsilon)$.

Proof. Consider the sequence $\varepsilon, f(\varepsilon), f^2(\varepsilon), f^3(\varepsilon), \ldots$. Note that $\varepsilon \leq f(\varepsilon)$, so by monotonicity of f and induction $f^i(\varepsilon) \leq f^{i+1}(\varepsilon)$. Thus this sequence forms an ascending chain. Since P has finite height, this chain cannot be infinite; thus there is an n such that $f^n(\varepsilon) = f^{n+1}(\varepsilon)$, i.e. $f^n(\varepsilon)$ is a fixed-point of f.

Now consider any fixed-point x of f. Since $\varepsilon \leq x$, by monotonicity of f, induction, and $x = f(x)$, we have $f^n(\varepsilon) \leq x$. Thus $f^n(\varepsilon)$ is the least fixed point of f. □

We write ($\mathrm{lfp}\ f \in L$) for the least fixed point of a monotone map f on a semilattice L of finite height.

5.4 Metatheory

We have proven the following theorems:

Theorem 1 (Weakening and exchange)**.** *The rules*

$$\frac{\Delta; \Gamma \vdash e : A}{\Delta, \Delta'; \Gamma, \Gamma' \vdash e : A}\ \textsc{weak} \qquad \frac{\Delta_2, \Delta_1; \Gamma_2, \Gamma_1 \vdash e : A}{\Delta_1, \Delta_2; \Gamma_1, \Gamma_2 \vdash e : A}\ \textsc{xchg}$$

are admissible.

Theorem 2 (Substitution, discrete)**.** *From*

- $\Delta; \cdot \vdash e_1 : A$,
- *and* $\Delta, x : A; \Gamma \vdash e_2 : B$,

it follows that

- $\Delta; \Gamma \vdash [e_1/x]\, e_2 : B$,
- *and* $[\![[e_1/x]\, e_2]\!]\ \delta\ \gamma = [\![e_2]\!]\ \langle \delta,\ [\![e_1]\!]\ \delta\ \gamma \rangle\ \gamma$.

Theorem 3 (Substitution, monotone)**.** *From*

- $\Delta; \Gamma \vdash e_1 : A$,
- *and* $\Delta; \Gamma, \mathbf{x} : A \vdash e_2 : B$

it follows that

- $\Delta; \Gamma \vdash [e_1/\mathbf{x}]\, e_2 : B$,
- *and* $[\![[e_1/\mathbf{x}]\, e_2]\!]\ \delta\ \gamma = [\![e_2]\!]\ \delta\ \langle \gamma,\ [\![e_1]\!]\ \delta\ \gamma \rangle$.

5.5 Discussion

It has been known for a very long time that database queries have a monadic structure arising from the adjunction between Set and SemiLat — indeed, the very name of the Kleisli (Wong 2000) database system was chosen to reflect this fact!

However, our decomposition of this adjunction into two smaller adjunctions, with an intermediate way-station in Poset is new. By interpreting our types in the intermediate category Poset, we gain access to the comonad Disc $|A|$. This lets us distinguish between monotone and non-monotone computations, which is the critical property letting us interpret fixed points in a sensible way. Indeed, it would also have been possible to directly reflect the adjunctions

in the syntax (in the style of Benton and Wadler (1996)), but we chose not to because the explicit coercions were somewhat noisy in practice. However, the ghost of this logic persists, as can be seen in the context-clearing actions in our typing rules.

6. An operational semantics

We consider the denotational semantics to be primary in Datafun; as with Datalog, any implementation technique is valid so long as it lines up with these semantics. As a proof of concept, however, we present a simple call-by-value structural operational semantics in Figure 11 and show that all well-typed terms terminate.

In our operational semantics we:

1. Assume all semilattice operations (ε, $\vee$, $\bigvee$, and fix) are subscripted with their type.
2. Drop the distinction between discrete and monotone variables, and write x, y for arbitrary variables.
3. Add iter expressions, which occur as intermediate forms in the evaluation of fix.
4. Classify some expressions e as values v, and add a value-form $\{\vec{v}\}$ for finite sets.

We use a small-step operational semantics using *evaluation contexts* E after the style of Felleisen and Hieb (1992) to enforce a call-by-value evaluation order; an evaluation context E is an expression with a hole in it (written $[]$) such that whatever is in the hole is next in line to be evaluated (if it is not a value already). To fill the hole $[]$ in an evaluation context E with the expression e, we write $E[e]$.

We define a relation $e \mapsto e'$ for expressions e whose outermost structure is immediately reducible; we extend this relation to all expressions with the rule

$$\frac{e \mapsto e'}{E[e] \mapsto E[e']}$$

In our rules for $e \mapsto e'$ where e is an iter expression we make use of a decidable ordering test on values, $v \sqsubseteq u : \underset{eq}{A}$, and a corresponding equality test $v \equiv u : \underset{eq}{A}$. We define these using inference rules, but they are easily seen to be decidable by induction on $\underset{eq}{A}$. The quantifiers in the premise of the rule $\subseteq$ range over finite domains, and thus pose no issue.

6.1 Computing fix-ed points via iter-ation

Our implementation strategy for $\mathsf{fix}\ x\ \mathsf{is}\ e$ is exactly that suggested by the proof of Lemma 4: starting from ε, iteratively apply $\lambda x.\, e$ until a fixed point is reached.

To model this iteration, we introduce iter forms into our syntax. The fixed point expression $\mathsf{fix}_{\underset{fin}{L}}\ x\ \mathsf{is}\ e$ immediately steps to the form $\mathsf{iter}_{\underset{fin}{L}}(\varepsilon_{\underset{fin}{L}}; x.\, e)$, which can be thought of as an initial state of the iteration, starting with $\varepsilon_{\underset{fin}{L}}$.

The intuition for iterating to the fixed point is that we apply the body and then check to see if the result changed. This is why we also introduce the two-place version $\mathsf{iter}_{\underset{eq}{A}}(v_1; e_2; x.\, e)$ which remembers the old value v_1, so that we can test it against e_2 (when it reaches a value v_2) to determine whether we've reached a fixed point. If not, we can continue to iterate from v_2 with $\mathsf{iter}_{\underset{fin}{L}}(v_2; x.\, e)$.

The $\mathsf{iter}_{\leq}$ forms are similar, but additionally check that the iteration value never exceeds their first argument, to implement the clamping behavior of $\mathsf{fix}\ x \leq e_{\top}\ \mathsf{is}\ e$.

6.2 A logical relation for termination

To prove that all well-typed terms terminate according to our operational semantics, we use a logical relations argument.

The standard approach of interpreting each type as a partial equivalence relation (PER) on closed terms turns out not to be sufficient in our case, and we need to extend it to prove termination. Just as posets are sets equipped with an order structure, we define our semantic types as PERs equipped with a preorder respecting the PER structure. The intuition is that since we needed the order structure in the denotational semantics to prove the definedness of fixed points, we will similarly need an order structure on the syntax to prove the termination of fixed points. Therefore, we inductively define relations $a \sqsubseteq b \mid A$ for each type A, then show how to understand these relations as preordered PERs.

As a matter of notation, a, b, c range over closed expressions; γ, σ over substitutions containing only closed expressions; and $\mathsf{Ctx}(\Gamma)$ is the set of all substitutions of closed expressions for the variables in Γ.

Because it simplifies our definitions and proofs, we introduce an additional pseudo-type $\Box A$, which orders A *discretely*, $x \leq y \iff x = y$.

$$\text{types} \quad A \quad ::= \quad \ldots \mid \Box A$$

This represents the $\mathsf{Disc}\ |-|$ comonad on Poset present in our denotational semantics. Observe that under this interpretation $A \to B$ is just $\Box A \xrightarrow{+} B$. In particular, if we let $[\![\Box A]\!] = \mathsf{Disc}\ |[\![A]\!]|$ then $[\![A \to B]\!] = [\![\Box A \xrightarrow{+} B]\!] = \mathsf{Disc}\ |[\![A]\!]| \Rightarrow [\![B]\!]$.

We define the following relations:

$$\begin{array}{rll}
a \sqsubseteq b \mid A & & \text{definition given below} \\
\gamma \sqsubseteq \sigma \mid \Gamma & \text{iff} & \forall (x{:}A \in \Gamma)\ \gamma(x) \sqsubseteq \sigma(x) \mid A \\
& & (\text{for } \gamma, \sigma \in \mathsf{Ctx}(\Gamma)) \\
\Gamma \vdash e_1 \sqsubseteq e_2 \mid A & \text{iff} & \forall (\gamma_1 \sqsubseteq \gamma_2 \mid \Gamma)\ \gamma_1, e_1 \sqsubseteq \gamma_2, e_2 \mid A \\
\gamma_1, e_1 \sqsubseteq \gamma_2, e_2 \mid A & \text{iff} & \forall (i = 1, 2)\ \gamma_i(e_1) \sqsubseteq \gamma_i(e_2) \mid A \\
& & \wedge\ \gamma_1(e_i) \sqsubseteq \gamma_2(e_i) \mid A
\end{array}$$

$\gamma_1, e_1 \sqsubseteq \gamma_2, e_2 \mid A$ may be seen as a transitive square:

$$\begin{array}{ccc}
\gamma_1(e_1) & \sqsubseteq & \gamma_1(e_2) \\
\sqcap\!| & & \sqcap\!| \\
\gamma_2(e_1) & \sqsubseteq & \gamma_2(e_2)
\end{array}$$

As a matter of notation, for any relation $Y \sqsubseteq Z \mid X$, we write:

$$\begin{array}{rll}
Y \equiv Z \mid X & \text{iff} & Y \sqsubseteq Z \mid X \wedge Z \sqsubseteq Y \mid X \\
Y \mid X & \text{iff} & Y \sqsubseteq Y \mid X
\end{array}$$

We now give the definition of $a \sqsubseteq b \mid A$ by induction on A:

$$\begin{array}{rll}
a \sqsubseteq b \mid \Box A & \text{iff} & a \equiv b \mid A \\
a \sqsubseteq b \mid 2 & \text{iff} & a \mapsto^* v \wedge b \mapsto^* u \wedge v \sqsubseteq u : 2 \\
a \sqsubseteq b \mid \{A\} & \text{iff} & a \mapsto^* \{\vec{v_i}\} \wedge b \mapsto^* \{\vec{u_i}\} \\
& & \wedge\ \forall v_i\ \exists u_j\ (v_i \equiv u_j \mid A) \\
a \sqsubseteq b \mid A_1 + A_2 & \text{iff} & a \mapsto^* \mathsf{in}_i\ v \wedge b \mapsto^* \mathsf{in}_i\ u \wedge v \sqsubseteq u \mid A_i \\
a \sqsubseteq b \mid A_1 \times A_2 & \text{iff} & a \mapsto^* (v_1, v_2) \wedge b \mapsto^* (u_1, u_2) \\
& & \wedge\ \forall i\ (v_i \sqsubseteq u_i \mid A_i) \\
a \sqsubseteq b \mid A \to B & \text{iff} & a \sqsubseteq b \mid \Box A \xrightarrow{+} B \\
a \sqsubseteq b \mid A \xrightarrow{+} B & \text{iff} & a \mapsto^* \lambda x.\, e_1 \wedge b \mapsto^* \lambda x.\, e_2 \\
& & \wedge\ (x{:}A \vdash e_1 \sqsubseteq e_2 \mid B)
\end{array}$$

It may not be immediately obvious that (for a given A) the relation $a \sqsubseteq b \mid A$ can be seen as a preordered PER. This requires the following two theorems, proven by induction on A:

Theorem 4 (Partial reflexivity)**.** *If* $a \sqsubseteq b \mid A$ *then* $a \sqsubseteq a \mid A$ *and* $b \sqsubseteq b \mid A$.

Theorem 5 (Transitivity)**.** *If* $a \sqsubseteq b \mid A$ *and* $b \sqsubseteq c \mid A$*, then* $a \sqsubseteq c \mid A$.

Additional syntax

$$
\begin{array}{lcl}
\begin{array}{c} e, f, g \\ \text{expressions} \end{array} & ::= & \ldots \mid \{\vec{v}\} \mid \varepsilon_L \mid e \vee_L e \mid \bigvee_L (x \in e)\ e \\
& & \mathsf{fix}_{\underset{fin}{L}}\ x \text{ is } e \mid \mathsf{fix}_{\underset{eq}{L}}\ x \le e \text{ is } e \\
& & \mathsf{iter}_{\underset{eq}{A}}(e; x.\, e) \mid \mathsf{iter}_{\underset{eq}{A}}(e; e; x.\, e) \\
& & \mathsf{iter}_{\le \underset{eq}{A}}(e; e; x.\, e) \mid \mathsf{iter}_{\le \underset{eq}{A}}(e; e; e; x.\, e) \\
\begin{array}{c} v, u, w \\ \text{values} \end{array} & ::= & \lambda x.\, e \mid (v, v) \mid \mathsf{in}_i\ v \mid \mathsf{true} \mid \mathsf{false} \mid \{\vec{v}\} \\
\begin{array}{c} E \\ \text{evaluation} \\ \text{contexts} \end{array} & ::= & [\,] \mid E\ e \mid v\ E \mid (E, e) \mid (v, E) \mid \mathsf{in}_i\ E \mid \pi_i\ E \\
& & E \vee_L e \mid v \vee_L E \mid \bigvee_L (x \in E)\ e \\
& & \mathsf{if}\ E\ \mathsf{then}\ e\ \mathsf{else}\ e \\
& & \mathsf{case}\ E\ \mathsf{of}\ \mathsf{in}_1\ x \to e;\ \mathsf{in}_2\ x \to e \\
& & \mathsf{iter}_{\underset{eq}{A}}(E; x.\, e) \mid \mathsf{iter}_{\underset{eq}{A}}(v; E; x.\, e) \\
& & \mathsf{iter}_{\le \underset{eq}{A}}(E; e; x.\, e) \mid \mathsf{iter}_{\le \underset{eq}{A}}(v; E; x.\, e) \\
& & \mathsf{iter}_{\le \underset{eq}{A}}(v; v; E; x.\, e)
\end{array}
$$

Rules for $v \sqsubseteq u : \underset{eq}{A}$ and $v \equiv u : \underset{eq}{A}$

$$
\frac{}{\mathsf{false} \sqsubseteq \mathsf{false} : 2} \qquad \frac{}{\mathsf{false} \sqsubseteq \mathsf{true} : 2} \qquad \frac{}{\mathsf{true} \sqsubseteq \mathsf{true} : 2}
$$

$$
\frac{\forall v_i\ \exists u_j\ (v_i \equiv u_j : \underset{eq}{A})}{\{\vec{v_i}\} \sqsubseteq \{\vec{u_i}\} : \{\underset{eq}{A}\}} \subseteq \qquad \frac{v_1 \sqsubseteq v_2 : \underset{eq}{A} \quad u_1 \sqsubseteq u_2 : \underset{eq}{B}}{(v_1, u_1) \sqsubseteq (v_2, u_2) : \underset{eq}{A} \times \underset{eq}{B}}
$$

$$
\frac{v \sqsubseteq u : \underset{eq}{A_i}}{\mathsf{in}_i\ v \sqsubseteq \mathsf{in}_i\ u : \underset{eq}{A_1} + \underset{eq}{A_2}} \qquad \frac{v \sqsubseteq u : \underset{eq}{A} \quad u \sqsubseteq v : \underset{eq}{A}}{v \equiv u : \underset{eq}{A}} \equiv
$$

β-reductions

$$
\begin{array}{rcl}
(\lambda x.\, e_1)\ e_2 & \mapsto & [e_2/x]\ e_1 \\
\pi_i\ (v_1, v_2) & \mapsto & v_i \\
\mathsf{case}\ \mathsf{in}_i\ v\ \mathsf{of}\ \overrightarrow{\mathsf{in}_j\ x_j \to e_j} & \mapsto & [v/x_i]\ e_i \\
\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \mapsto & e_1 \\
\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \mapsto & e_2
\end{array}
$$

Evaluating ε

$$
\begin{array}{rcl}
\varepsilon_2 & \mapsto & \mathsf{false} \\
\varepsilon_{\{A\}} & \mapsto & \{\} \\
\varepsilon_{L \times M} & \mapsto & (\varepsilon_L, \varepsilon_M) \\
\varepsilon_{A \to L} & \mapsto & \lambda x.\, \varepsilon_L \\
\varepsilon_{A \overset{+}{\to} L} & \mapsto & \lambda x.\, \varepsilon_L
\end{array}
$$

Evaluating ∨

$$
\begin{array}{rcl}
\mathsf{false} \vee_2 v & \mapsto & v \\
\mathsf{true} \vee_2 v & \mapsto & \mathsf{true} \\
\{\vec{v}\} \vee_{\{A\}} \{\vec{u}\} & \mapsto & \{\vec{v}, \vec{u}\} \\
(v_1, v_2) \vee_{L \times M} (u_1, u_2) & \mapsto & (v_1 \vee_L u_1, v_2 \vee_M u_2) \\
v \vee_{A \to L} u & \mapsto & \lambda x.\, v\ x \vee_L u\ x \\
v \vee_{A \overset{+}{\to} L} u & \mapsto & \lambda x.\, v\ x \vee_L u\ x
\end{array}
$$

Evaluating ⋁

$$
\begin{array}{rcl}
\bigvee_L (x \in \{\})\ e & \mapsto & \varepsilon_L \\
\bigvee_L (x \in \{v, \vec{u}\})\ e & \mapsto & [v/x]\ e \vee_L \bigvee_L (x \in \{\vec{u}\})\ e
\end{array}
$$

Evaluating fix and iter

$$
\begin{array}{rcl}
\mathsf{fix}_{\underset{fin}{L}}\ x \text{ is } e & \mapsto & \mathsf{iter}_{\underset{fin}{L}}(\varepsilon_{\underset{fin}{L}}; x.\, e) \\
\mathsf{iter}_{\underset{eq}{A}}(v; x.\, e) & \mapsto & \mathsf{iter}_{\underset{eq}{A}}(v; [v/x]\ e; x.\, e) \\
\mathsf{iter}_{\underset{eq}{A}}(v_1; v_2; x.\, e) & \mapsto & \begin{cases} v_1 & \text{if } v_1 \equiv v_2 : \underset{eq}{A} \\ \mathsf{iter}_{\underset{eq}{A}}(v_2; x.\, e) & \text{otherwise} \end{cases} \\
\mathsf{fix}_{\underset{eq}{L}}\ x \le e_\top \text{ is } e & \mapsto & \mathsf{iter}_{\le \underset{eq}{L}}(e_\top; \varepsilon_{\underset{eq}{L}}; x.\, e) \\
\mathsf{iter}_{\le \underset{eq}{A}}(v_\top; v; x.\, e) & \mapsto & \begin{cases} \mathsf{iter}_{\le \underset{eq}{A}}(v_\top; v; [v/x]\ e; x.\, e) & \text{if } v \sqsubseteq v_\top : \underset{eq}{A} \\ v_\top & \text{otherwise} \end{cases} \\
\mathsf{iter}_{\le \underset{eq}{A}}(v_\top; v_1; v_2; x.\, e) & \mapsto & \begin{cases} v_1 & \text{if } v_1 \equiv v_2 : \underset{eq}{A} \\ \mathsf{iter}_{\le \underset{eq}{A}}(v_\top; v_2; x.\, e) & \text{otherwise} \end{cases}
\end{array}
$$

Figure 11. Operational semantics

From these it follows immediately that $a \equiv b \mid A$ is a PER, and $a \sqsubseteq b \mid A$ forms a preorder over this PER which respects it.

Theorem 6 (Termination)**.** *If* $a \mid A$ *then* $a \mapsto^* v$.

Proof. By cases on the definition of $a \sqsubseteq a \mid A$. □

Theorem 7 (Fundamental theorem)**.** *If* $\Delta; \Gamma \vdash e : A$ *then* $\Box\Delta, \Gamma \vdash a \mid A$.

Proof. By induction on $\Delta; \Gamma \vdash e : A$; full proof available at `https://github.com/rntz/datafun/`. The key case is the fixed point rule, whose proof is a syntactic version of the proof of definedness of fixed points in the denotational semantics. □

It follows as immediate corollary of termination and the fundamental property that every closed, well-typed program terminates.

7. Comparing Datalog and Datafun

At this point, we have demonstrated by example that Datafun programs are rather similar to Datalog programs, and we have given the typing and denotational semantics of Datafun. However, we still need to explain *why* our semantics lets us express Datalog-style programs.

To understand this, recall that Datalog is a bottom-up logic programming language. A program consists of a primitive database of facts, along with a set of rules the rules the programmer wrote. A Datalog program executes by using the rules to derive new conclusions from the database, and extending the database with them, until no additional conclusions can be drawn. Then the query can be checked simply by seeing if it occurs in the final database.

This is, essentially, a fixed point computation – each stage of execution of a Datalog program takes a database and returns an extended database, until a fixed point is reached. The stratified negation restriction essentially ensures that the database transformer defined by a Datalog program is a monotone function on the set of

facts. This is why the type system of Datafun tracks the monotonicity of functions — since we permit both higher-order definitions and taking fixed points, we need to ensure that the body of a fixed point definition is monotone in order to guarantee that the recursion is well-founded.

This ensures that the recursive definition is well-defined, but is not sufficient by itself to guarantee termination. To manage this, Datalog depends upon the other two restrictions described in the introduction. By restricting terms occuring in predicates to consist of either atoms or variables, Datalog ensures that quantifiers need only be instantiated with the atoms used in a program. By requiring every variable in the consequent of rules to also occur in the premise of a rule, it ensures that every consequent will also only feature atoms occuring in the original program.

Then, since there can only be finitely many atoms in a finite program, this means that the set of possible arguments to a predicate is itself finite. Then the lattice of sets of atomic predicates ordered by inclusion will be finite, and so fixed point iteration is guaranteed to terminate.

Instead of this (rather indirect) scheme, Datafun directly tracks the finiteness of types, permitting recursion only if it is over a finite type, or is bounded explicitly. These two approaches achieve the same effect, albeit in different ways. Datalog's approach has the benefit that no type discipline is needed to ensure finiteness. One advantage of our choice is that we permit recursion over any semilattice, not just the semilattice of sets. A much more serious advantage of our approach is that it makes it much easier to write fixed-point computations which actually *compute* with the data they see (for example, the CYK parser we wrote computed lengths of substrings).

8. Implementation

We have built a proof-of-concept implementation of Datafun in Racket, available at `https://github.com/rntz/datafun/`. In addition to core Datafun, it supports pattern-matching, variant types, record types, dictionaries, subtyping, antitone functions, and unbounded (potentially nonterminating) fixed points. We implement everything in a naïve style, and perform no optimizations.

Type inference As a practical matter, type-checking needs to distinguish between discrete and monotone λ, application, case, let, and if. In our implementation we solve this in two ways:

1. Bidirectional type inference (Pierce and Turner 2000) determines whether λs and applications are discrete or monotone.
2. For if, case, and let, the programmer annotates which form is intended; for example, $(\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ \varepsilon)$ is written (`when e then e1`) to indicate the rule IF^+ applies.

We believe that this scheme could be extended to support polymorphism in the style of Dunfield and Krishnaswami (2013). However, it would not be an entirely off-the-shelf affair, since we would want to add support for polymorphism over the tones of function, so that, for example, $\lambda f.\,\lambda x.\,f\ x$ can be assigned the principal type $\forall o : \mathsf{tone}.\,\forall \alpha, \beta : \mathsf{type}.\,(\alpha \xrightarrow{o} \beta) \xrightarrow{+} (\alpha \xrightarrow{o} \beta)$, where $\xrightarrow{o}$ indicates a function of tone o; a tone may be empty (for a discrete function) or $+$ for a monotone function.

9. Related and future work

Aggregation Aggregation of values — for example, taking the sum $\sum_{x \in A} f\ x$ of a function f across a set A — is a useful and ubiquitous database operation. Datafun naturally supports *semilattice* aggregation via $\bigvee$, but many natural operations such as summation do not form semilattices on their underlying type. There are several potential ways to add support for aggregations to Datafun:

- Common aggregations can be provided as primitive functions, for example $\mathsf{size} : \{\underset{eq}{A}\} \xrightarrow{+} \mathbb{N}$ or $\mathsf{sum} : (\underset{eq}{A} \to \mathbb{N}) \to \{\underset{eq}{A}\} \xrightarrow{+} \mathbb{N}$.
- In the style of Machiavelli (Ohori et al. 1989), one could add a general operator $\mathsf{hom} : B \to (A \to B \to B) \to \{A\} \to B$, which effectively linearizes a set in an unspecified order. The semantics of hom are, alas, necessarily nondeterministic.
- One could augment Datafun with a type of bags (multisets) A^*; bags naturally support a much broader class of aggregation — commutative monoids — than sets. See, for example, Budiu and Plotkin (2014) and Gibbons et al. (2015).

Optimization Because Datalog is so strongly constrained, there has been a lot of very successful work on optimizing it, ranging from compilation into binary decision diagrams (Bryant 1992) by Whaley et al. (2005), to the famous "magic sets" (Bancilhon et al. 1986) algorithm.

From our perspective, magic sets are a natural next step for investigation into how to optimize Datafun. Intuitively, the magic sets algorithm exploits the fact that Datalog (as a total logic language) has both a top-down and bottom-up reading, and rewrites the program so that it does bottom-up search, while using top-down reasoning to strategically avoid adding useless facts to the database. Transplanting this analysis to Datafun would essentially give us optimized implementations of fixed points, but extending the magic sets algorithm is likely to be very subtle, since Datafun has higher-order functions and Datalog does not. As a result, our goal is to first see if magic sets can be applied to first-order Datafun programs, and then use defunctionalization (Reynolds 1998) to extend it to full Datafun.

Very recently, Madsen et al. (2016) have introduced the Flix language, which extends the semantics of Datalog to support defining relations valued in arbitrary lattices (rather than just the powerset of atoms). Like Datafun, this lets Flix support using monotone functions (on suitable lattices) in program expressions. Unlike Datafun, Flix does not yet have monotonicity checking for programmer-defined operators. However, because Flix does not extend Datalog to higher order, efficient Datalog implementation strategies (such as semi-naive evaluation) continue to apply.

Databases Datalog has sometimes been described as "relational algebra plus fixed points", and there is a long line of work on embedding database query languages into general-purpose languages, including pioneering efforts such as Machiavelli (Ohori et al. 1989) and Kleisli (Wong 2000), as well as more recent systems such as Ferry (Grust et al. 2009) and LINQ in C# (Cheney et al. 2013). The focus of this work has been on embedding query languages based on relational algebra into general purpose languages, with an emphasis on statically compiling higher-order queries into the first-order queries supported by existing database systems (Cheney et al. (2014) is a representative example).

Our approach is a little bit different. Instead of embedding Datalog into a general purpose language, Datafun is *also* a "little language", albeit one that happens to be a higher-order functional language. We very deliberately did not try to embed Datafun into an existing language, because that would have greatly complicated the context-management operations needed to ensure monotonicity.

In fact, from a language designer's perspective, Datafun can be seen as an argument in favor of extending functional languages to support programming with user-defined, non-strong comonads.

Deletion Ganzinger and McAllester (2002) showed how forward-chaining logic programming permits concise and elegant expression of a wide variety of algorithms, including a natural cost semantics. However, they noted that there were some algorithms (such as union-

find and greedy algorithms) which could be formulated in this style, *if* there were additionally support for deleting facts from a database. Later, Simmons and Pfenning (2009) went on to show how deletion could be given a logical interpretation by formulating in terms of linear logic programming.

This naturally raises the question of whether we could identify a "linear Datafun" corresponding to this style of programming, where we might linear types to model features like deletion. There are many nontrivial semantic issues (e.g., how to define monotonicity), but it seems a promising question for future work.

Termination Datafun as presented is Turing-incomplete. This is advantageous for optimization; for example, one powerful optimization technique is *loop reordering* (in SQL terminology, *join reordering*), that is, taking advantage of the equation

$$\bigvee(x \in e_1)\ \bigvee(y \in e_2)\ e \quad = \quad \bigvee(y \in e_2)\ \bigvee(x \in e_1)\ e$$

when $x, y \notin \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2)$. But this equation does not always hold in the presence of nontermination; for example, if $e_1 = \varepsilon$ and e_2 diverges.

Nonetheless, without adding advanced facilities for termination checking, there are many functions it is difficult to implement without use of general recursion. So a natural direction for future work is to study how to add support for general recursion to Datafun. Because domains (Abramsky and Jung 1994) can be understood as partial orders with directed joins, there are likely many interesting categorical structures connecting the category of domains to the category of posets, some of which will hopefully lead to a principled type-theoretic integration of partial functions into Datafun.

User-Defined Posets and Semilattices The two fundamental semilattice types Datafun provides are booleans and sets; products and functions merely preserve semilattice structure where they find it. One might contemplate allowing the programmer to define their own semilattice structures using something like Haskell's `newtype`/`instance`. In general, this is a difficult problem, because we may need to do serious mathematical reasoning to prove that a comparison function implements a partial ordering, or that a datatype can be equipped with a semilattice structure obeying this partial ordering which is commutative, associative and idempotent.

One example of such a family of types are the *lexicographic sum types*. Given two posets P and Q, their disjoint union $\mathsf{P} + \mathsf{Q}$ is also a poset, with left values compared by the P-ordering, and right values compared by the Q-ordering, and no ordering between left and right values. However, this is not the only way that the disjoint union could be equipped with an order structure.

For example, we could define the *lexicographic* sum $\mathsf{P} \lhd \mathsf{Q}$, which has the same elements as the sum, but extending the coproduct order relation with the additional facts that $\mathsf{in}_1(p) \leq \mathsf{in}_2(q)$. Indeed, we already have a special case of this: as we noted earlier, our boolean type is not $1 + 1$, but it *is* $1 \lhd 1$.

But as our Booleans already show, giving good syntax for their eliminators is difficult, because we have to show that not just a term is monotone, but that the different branches of a lexicographic case expression are ordered with respect to *each other*. For the case of ordered Booleans, we were able to give a special eliminator which guaranteed it, but in general it requires proof.

One natural direction for future work is to extend the syntax of Datafun with support for these kinds of proofs, perhaps taking inspiration from dependent type theory.

Acknowledgments

Thank you to:

- Chris Martens, for helpful feedback and discussion on connections to logic programming.
- The anonymous ICFP reviewers, for pointing out areas which were previously unclear or incomplete.

References

S. Abramsky and A. Jung. Domain theory. *Handbook of logic in computer science*, 3:1–168, 1994.

P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011.

S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, Apr. 2010. ISSN 0001-0782.

M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1371–1382, 2015.

F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM. ISBN 0-89791-179-2. doi: 10.1145/6012.15399. URL `http://doi.acm.org/10.1145/6012.15399`.

M. Y. Becker, C. Fournet, and A. D. Gordon. Secpal: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4):619–665, 2010.

P. N. Benton and P. Wadler. Linear logic, monads and the lambda calculus. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 420–431, 1996.

R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

M. Budiu and G. Plotkin. Multilinear programming with big data. Festschrift for Luca Cardelli, September 2014. URL `http://budiu.info/work/budiu-festschrift14.pdf`.

J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 403–416, 2013.

J. Cheney, S. Lindley, and P. Wadler. Query shredding: efficient relational evaluation of queries over nested multisets. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1027–1038, 2014.

O. de Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble. .QL: Object-oriented queries made easy. In *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, pages 78–133, 2007.

J. Dunfield and N. R. Krishnaswami. Complete and easy bidirectional type-checking for higher-rank polymorphism. In *International Conference on Functional Programming (ICFP)*, Sept. 2013. `arXiv:1306.6032[cs.PL]`.

M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, Sept. 1992. ISSN 0304-3975. doi: 10.1016/0304-3975(92)90014-7. URL `http://dx.doi.org/10.1016/0304-3975(92)90014-7`.

D. P. Friedman, W. E. Byrd, and O. Kiselyov. *The reasoned schemer*. MIT Press, 2005. ISBN 978-0-262-56214-0.

H. Gallaire and J. Minker, editors. *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, 1977*, Advances in Data Base Theory, New York, 1978. Plemum Press. ISBN 0-306-40060-X.

H. Ganzinger and D. A. McAllester. Logical algorithms. In *Logic Programming, 18th International Conference, ICLP 2002, Copenhagen, Denmark, July 29 - August 1, 2002, Proceedings*, pages 209–223, 2002.

J. Gibbons, F. Henglein, R. Hinze, and N. Wu. Relational algebra by way of adjunctions. In *Database Programming Languages*, October 2015. URL `http://www.cs.ox.ac.uk/jeremy.gibbons/publications/reladj-dbpl.pdf`. Talk only.

J. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi: 10.1016/0304-3975(87)90045-4. URL `http://dx.doi.org/10.1016/0304-3975(87)90045-4`.

T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. FERRY: database-supported program execution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 1063–1066, 2009.

R. Hickey, S. Halloway, and J. Gehtland. Datomic: The fully transactional, cloud-ready, distributed database. `http://www.datomic.com`. Accessed: 11 March 2016.

M. Madsen, M. Yee, and O. Lhoták. From datalog to flix: a declarative language for fixed points on lattices. In C. Krintz and E. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 194–208. ACM, 2016. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908096. URL `http://doi.acm.org/10.1145/2908080.2908096`.

A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli—a polymorphic language with static type inference. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 46–57, New York, NY, USA, 1989. ACM. ISBN 0-89791-317-5. doi: 10.1145/67544.66931. URL `http://doi.acm.org/10.1145/67544.66931`.

F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001. doi: 10.1017/S0960129501003322. URL `http://dx.doi.org/10.1017/S0960129501003322`.

B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

M. Schäfer and O. de Moor. Type inference for datalog with complex type hierarchies. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 145–156, 2010.

S. M. Shieber, Y. Schabes, and F. C. N. Pereira. Principles and implementation of deductive parsing. *J. Log. Program.*, 24(1&2):3–36, 1995.

R. J. Simmons and F. Pfenning. Linear logical approximations. In *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009*, pages 9–20, 2009.

Z. Somogyi, F. Henderson, and T. C. Conway. The implementation of mercury, an efficient purely declarative logic programming language. In *ILPS 1994, Workshop 4: Implementation Techniques for Logic Programming Languages, Ithaca, New York, USA, November 17, 1994*, 1994.

J. Whaley. *Context-Sensitive Pointer Analysis using Binary Decision Diagrams*. PhD thesis, Stanford University, Mar. 2007.

J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog and binary decision diagrams for program analysis. In K. Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, volume 3780. Springer-Verlag, Nov. 2005.

L. Wong. Kleisli, a functional query system. *J. Funct. Program.*, 10(1): 19–56, 2000.