Seminaïve Evaluation for a Higher-Order Functional Language

MICHAEL ARNTZENIUS, University of Birmingham, United Kingdom
NEEL KRISHNASWAMI, University of Cambridge, United Kingdom

One of the workhorse techniques for implementing bottom-up Datalog engines is seminaïve evaluation. This optimization improves the performance of Datalog's most distinctive feature: recursively defined predicates. These are computed iteratively, and under a naïve evaluation strategy, each iteration recomputes all previous values. Seminaïve evaluation computes a safe approximation of the *difference* between iterations. This can *asymptotically* improve the performance of Datalog queries.

Seminaïve evaluation is defined partly as a program transformation and partly as a modified iteration strategy, and takes advantage of the first-order nature of Datalog code. This paper extends the seminaïve transformation to higher-order programs written in the Datafun language, which extends Datalog with features like first-class relations, higher-order functions, and datatypes like sum types.

CCS Concepts: • **Theory of computation → Database query languages (principles)**; *Constraint and logic programming*; *Database query processing and optimization (theory)*; Modal and temporal logics; Logic and databases; • **Software and its engineering → Functional languages**; *Constraint and logic languages*; **Multiparadigm languages**; Data types and structures; Recursion.

Additional Key Words and Phrases: Datafun, Datalog, functional languages, relational languages, seminaïve evaluation, incremental computation

ACM Reference Format:
Michael Arntzenius and Neel Krishnaswami. 2020. Seminaïve Evaluation for a Higher-Order Functional Language. *Proc. ACM Program. Lang.* 4, POPL, Article 22 (January 2020), 28 pages. https://doi.org/10.1145/3371090

1 INTRODUCTION

Datalog [Ceri et al. 1989], along with the π-calculus and λ-calculus, is one of the jewel languages of theoretical computer science, connecting programming language theory, database theory, and complexity theory. In terms of programming languages, Datalog can be understood as a fully declarative subset of Prolog which is guaranteed to terminate and so can be evaluated in both top-down and bottom-up fashion. In terms of database theory, it is equivalent to the extension of relational algebra with a fixed point operator. In terms of complexity theory, stratified Datalog over ordered databases characterizes polytime computation [Dantsin et al. 2001].

In addition to its theoretical elegance, over the past twenty years Datalog has seen a surprisingly wide array of uses across a variety of practical domains, both in research and in industry. Whaley and Lam [Whaley 2007; Whaley and Lam 2004] implemented pointer analysis algorithms in Datalog, and found that they could reduce their analyses from thousands of lines of C code to

Authors' addresses: Michael Arntzenius, School of Computer Science, University of Birmingham, Birmingham, B15 2TT, United Kingdom, daekharel@gmail.com; Neel Krishnaswami, Department of Computer Science and Technology, University of Cambridge, Cambridge, CB2 1TN, United Kingdom, nk480@cl.cam.ac.uk.

2475-1421/2020/1-ART22
https://doi.org/10.1145/3371090

tens of lines of Datalog code, while retaining competitive performance. The DOOP pointer analysis framework [Smaragdakis and Balatsouras 2015], built using the Soufflé Datalog engine [Jordan et al. 2016], shows that this approach can handle almost all of industrial languages like Java, even analysing features like reflection [Fourtounis and Smaragdakis 2019]. Semmle has developed the Datalog-based .QL language [de Moor et al. 2007; Schäfer and de Moor 2010] 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 and retail prediction. 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. 2012] 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 each case, when the problem formulated in Datalog, the specification became directly implementable, while remaining dramatically shorter and clearer than alternatives implemented in more conventional languages.

However, even though each of these applications is supported by the skeleton of Datalog, they all had to extend it significantly beyond the theoretical core calculus. For example, core Datalog does not even support arithmetic, since its semantics only speaks of finite sets supporting equality of their elements. Moreover, arithmetic is not a trivial extension, since it can greatly complicates the semantics (for example, proving that termination continues to hold). So despite the fact that kernel Datalog has a very clean semantics, its metatheory seemingly needs to be laboriously re-established once again for each extension.

A natural approach to solving this problem is to find a language in which to write the extensions, which preserves the semantic guarantees that Datalog offers. Two such proposals are Flix [Madsen et al. 2016] and Datafun [Arntzenius and Krishnaswami 2016]. Conveniently for our exposition, these two languages embody two alternative design strategies.

Flix extends a Datalog-like relational language, generalized to handle arbitrary semilattices instead of only finite sets, with a functional sublanguage, roughly comparable to ML or Haskell. The functional side can be used to implement custom semilattices and data structures which can then be used from the Datalog side. Flix is aimed at static analysis, where working in a semilattice other than Datalog's native finite powersets can be highly useful. To this end, Flix integrates with SMT solvers for lightweight verification of properties such as monotonicity, soundness, and completeness. However, this SMT-based approach works best for first-order code, and Flix maintains a pretty clear (if permeable) separation between its relational and functional sublanguages.

Datafun, by contrast, is a functional language capable of expressing relational idioms directly. Datafun's type system tracks *monotonicity* of functions, including higher-order functions. Datalog-style recursively defined relations are given via an explicit fixed point operator; monotonicity ensures uniqueness of this fixed point, playing a role similar to stratification in Datalog. Tracking monotonicity permits a tighter integration between the functional and relational styles, but it comes at a cost: many of Datalog's standard implementation techniques, developed in the context of a first-order logic language, are not obviously applicable in a higher-order functional setting.

Indeed, making Datalog perform well enough to use in practice calls for very sophisticated program analysis and compiler engineering. (This is a familiar experience from the database community, where query planners must encode a startling amount of knowledge to optimize relatively simple SQL queries.) A wide variety of techniques for optimizing Datalog programs have been studied in the literature, such as using binary decision diagrams to represent relations [Whaley 2007], exploiting the structure of well-behaved subsets (e.g., CFL-reachability problems correspond to the "chain program" fragment of Datalog [Afrati and Papadimitriou 1993]), and combining top-down and bottom-up evaluation via the "magic sets" algorithm [Bancilhon et al. 1986].

Today, one of the workhorse techniques for implementing bottom-up Datalog engines is *seminaïve evaluation* [Bancilhon 1986]. This optimization improves the performance of Datalog's most distinctive feature: recursively defined predicates. These can be understood as the fixed point of a set-valued function f. The naïve way to compute this is to iterate the sequence $\emptyset, \mathsf{f}(\emptyset), \mathsf{f}^2(\emptyset), \ldots$ until $\mathsf{f}^{\mathsf{i}}(\emptyset) = \mathsf{f}^{\mathsf{i}+1}(\emptyset)$. However, each iteration will recompute all previous values. Seminaïve evaluation instead computes a safe approximation of the *difference* between iterations. This optimization is critical, as it can asymptotically improve the performance of Datalog queries.

Contributions. The seminaïve evaluation algorithm is defined partly as a program transformation on sets of Datalog rules, and partly as a modification of the fixed point computation algorithm. The central contribution of this paper is to give an extended version of this transformation which works on higher-order programs written in the Datafun language.

- We reformulate Datafun in terms of a kernel calculus based on the modal logic S4. Instead of giving a calculus with distinct monotonic and discrete function types, as in the original Datafun paper, we make discreteness into a comonad. In addition to regularizing the calculus and slightly improving its expressiveness, the explicit comonadic structure lets us impose a modal constraint on recursion reminiscent of Hoffman's work on safe recursion [Hofmann 1997]. This brings the semantics of Datafun more closely in line with Datalog's, and substantially simplifies the program transformations we present.
- We define two type-and-syntax-directed program transformations on Datafun: one to implement seminaïve evaluation, and an auxilliary translation that incrementalizes programs with respect to increasing changes. We build on the *change structure* approach to static program incrementalization introduced by Cai et al. [2014], extending it to support sum types, set types, a comonad, and (well-founded) fixed points.
- We establish the correctness of these transformations using a novel logical relation which captures the relation between the source program, its incrementalization, and its seminaïve translation. The fundamental lemma shows that our transformation is semantics-preserving: any closed program of first-order type has the same meaning after optimization.
- We discuss our implementation of a compiler from Datafun to Haskell, in both naïve and seminaïve form. This lets us empirically demonstrate the asymptotic speedups predicted by the theory. We additionally discuss the (surprisingly modest) set of auxilliary optimizations we found helpful for putting seminaïve evaluation into practice.

2 DATALOG AND DATAFUN, INFORMALLY

2.1 Datalog

Datalog's syntax is a subset of Prolog's. Programs are collections of predicate declarations:

$$\begin{aligned}&\mathit{parent}(\mathsf{aerys}, \mathsf{rhaegar})\\&\mathit{parent}(\mathsf{rhaegar}, \mathsf{jon})\\&\mathit{parent}(\mathsf{lyanna}, \mathsf{jon})\end{aligned}$$

$$\begin{aligned}&\mathit{ancestor}(\mathsf{X}, \mathsf{Z}) \leftarrow \mathit{parent}(\mathsf{X}, \mathsf{Z})\\&\mathit{ancestor}(\mathsf{X}, \mathsf{Z}) \leftarrow \mathit{parent}(\mathsf{X}, \mathsf{Y}) \wedge \mathit{ancestor}(\mathsf{Y}, \mathsf{Z})\end{aligned}$$

This defines two binary relations, *parent* and *ancestor*. Lowercase sans-serif words like aerys and rhaegar are symbols à la Lisp, and uppercase characters like X, Y, Z are variables. The *parent* relation is defined as a set of ground facts: we assert that aerys is rhaegar's parent, that rhaegar is jon's parent, and so on. The *ancestor* relation is defined by a pair of rules: first, that X is Z's ancestor if X is Z's parent; second, that X is Z's ancestor if X has a child Y who is an ancestor of Z.

$$\begin{array}{rrcl}
\text{types} & A, B & ::= & 1 \mid A \times B \mid A + B \mid A \to B \mid \Box A \mid \{\underset{\text{EQ}}{A}\} \\
\text{eqtypes} & \underset{\text{EQ}}{A}, \underset{\text{EQ}}{B} & ::= & \{\underset{\text{EQ}}{A}\} \mid 1 \mid \underset{\text{EQ}}{A} \times \underset{\text{EQ}}{B} \mid \underset{\text{EQ}}{A} + \underset{\text{EQ}}{B} \\
\text{semilattices} & L, M & ::= & \{\underset{\text{EQ}}{A}\} \mid 1 \mid L \times M \\
\text{finite eqtypes[2]} & \underset{\text{FIN}}{A}, \underset{\text{FIN}}{B} & ::= & \{\underset{\text{FIN}}{A}\} \mid 1 \mid \underset{\text{FIN}}{A} \times \underset{\text{FIN}}{B} \mid \underset{\text{FIN}}{A} + \underset{\text{FIN}}{B} \\
\text{fixtypes} & \underset{\text{FIX}}{L}, \underset{\text{FIX}}{M} & ::= & \{\underset{\text{FIN}}{A}\} \mid 1 \mid \underset{\text{FIX}}{L} \times \underset{\text{FIX}}{M} \\
\text{terms} & e, f, g & ::= & x \mid x \mid \lambda x.\ e \mid e\ f \mid () \mid (e, f) \mid \pi_i\ e \\
 & & & \mathrm{in}_i\ e \mid \mathbf{case}\ e\ \mathbf{of}\ (\mathrm{in}_i\ x_i \to f_i)_{i \in \{1,2\}} \\
 & & & [e] \mid \mathbf{let}\ [x] = e\ \mathbf{in}\ f \mid e = f \mid \mathit{empty?}\ e \mid \mathit{split}\ e \\
 & & & \bot \mid e \vee f \mid \{e_i\}_i \mid \mathbf{for}\ (x \in e)\ f \mid \mathit{fix}\ e
\end{array}$$

Fig. 1. Datafun syntax

Semantically, a predicate denotes the set of tuples that satisfy it. Compared to Prolog, one of the key restrictions Datalog imposes is that these sets are always *finite*. This helps keep proof search decidable, allowing for a variety of implementation strategies. In practice, most Datalog engines use bottom-up evaluation instead of Prolog's top-down backtracking search.

Recursive definitions like *ancestor* give rise to the set of facts deducible from the rules defining them. More formally, we can view these rules as defining a relation transformer and producing its least fixed point. For this to make sense, these rules must be *stratified*: a recursive definition cannot refer to itself beneath a negation. For example, the liar paradox is prohibited:

$\mathit{liar}() \leftarrow \neg \mathit{liar}()$ ✗ NOT VALID DATALOG

Stratification ensures the transformer the rules define is monotone, guaranteeing a unique least fixed point.

2.2 Datafun

The idea behind Datafun is to capture the essence of Datalog in a typed, higher-order, functional setting. Since the key restriction that makes Datalog tractable – stratification – requires tracking *monotonicity*, we locate Datafun's semantics in the category **Poset** of partial orders and monotone maps. Since **Poset** is bicartesian closed, it can interpret the simply typed λ-calculus, giving us a notation for writing monotone and higher-order functions. This lets us *abstract* over Datalog rules, something not possible in Datalog itself! In the remainder of this section we reconstruct Datafun hewing closely to this semantic intuition.

Datafun begins as the simply-typed λ-calculus with functions ($\lambda x.\ e$ and $e\ f$), sums ($\mathrm{in}_i\ e$ and $\mathbf{case}\ e\ \mathbf{of}\ \ldots$), and products ($(e, f)$ and $\pi_i\ e$). To represent relations, we add a type of finite sets $\{\underset{\text{EQ}}{A}\}$,[1] introduced with set literals $\{e_0, \ldots e_n\}$, and eliminated using Moggi's monadic bind syntax, $\mathbf{for}\ (x \in e_1)\ e_2$, signifying the union over all $x \in e_1$ of e_2. Since we are working in **Poset**, each type comes with a partial order on it; sets are ordered by inclusion, $x \leqslant y : \{\underset{\text{EQ}}{A}\} \iff x \subseteq y$.

As long as all primitives are monotone, every definable function is also monotone. This is necessary for defining fixed points, but may seem too restrictive. There are many useful non-monotone operations, such as equality tests $e = f$. For example, $\{\} = \{\}$ is true, but if the first argument increases to $\{1\}$ it becomes false, a *decrease* (as we'll see later, in Datafun, $\mathit{false} < \mathit{true}$).

[1] To implement set types, their elements must support decidable equality. In our core calculus, we use a subgrammar of "eqtypes", and in our implementation (which compiles to Haskell) we use typeclass constraints to pick out such types.

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\mathit{bool} & \xrightarrow{\text{rewrite}} & \{1\} & \{e \mid \varepsilon\} & \xrightarrow{\text{rewrite}} & \{e\} \\
\mathit{false} & \xrightarrow{\text{rewrite}} & \{\} & \{e \mid p \in f, \ldots\} & \xrightarrow{\text{rewrite}} & \textbf{for}\ (p \in f)\ \{e \mid \ldots\} \\
\mathit{true} & \xrightarrow{\text{rewrite}} & \{()\} & \{e \mid f, \ldots\} & \xrightarrow{\text{rewrite}} & \textbf{when}\ (f)\ \{e \mid \ldots\} \\
\textbf{fix}\ x\ \textbf{is}\ e & \xrightarrow{\text{rewrite}} & \mathit{fix}\ [\lambda x.\ e] & \textbf{when}\ (e)\ f & \xrightarrow{\text{rewrite}} & \textbf{for}\ (() \in e)\ f
\end{array}$$

Fig. 2. Syntactic sugar

How can we express non-monotone operations if all functions are monotone? We square this circle by introducing the *discreteness* type constructor, $\Box A$. The elements of $\Box A$ are the same as those of A, but the partial order on $\Box A$ is discrete, $x \leqslant y : \Box A \iff x = y$. Monotonicity of a function $\Box A \to B$ is vacuous: $x = y$ implies $f(x) \leqslant f(y)$ by reflexivity! In this way we represent ordinary, possibly non-monotone, functions $A \to B$ as monotone functions $\Box A \to B$.

Semantically, $\Box$ is a monoidal comonad or necessity modality, and so we base our syntax on Pfenning and Davies [2001]'s syntax for the necessity fragment of constructive S4 modal logic. This involves distinguishing two kinds of variable: discrete variables x are in *red italics*, while monotone variables $\mathtt{x}$ are in `upright black script`. Discrete variables may be used wherever they're in scope, but crucially, monotone variables are hidden within non-monotone expressions. For example, in an equality test $e = f$, the terms e and f cannot refer to monotone variables bound outside the equality expression. We highlight such expressions with a yellow background. Putting this all together, we construct the type $\Box A$ with the non-monotone introduction form $[e]$ and eliminate it by pattern-matching, **let** $[\mathit{x}] = e$ **in** f, giving access to a discrete variable x.

Finally, Datafun includes fixed points, *fix* f. The *fix* combinator takes a function $\Box(\underset{\text{FIX}}{L} \to \underset{\text{FIX}}{L})$ and returns its least fixed point. Besides monotonicity of the function, we impose two restrictions on the fixed point operator to ensure well-definedness and termination. First, we require that recursion occur at *semilattice types with no infinite ascending chains*, $\underset{\text{FIX}}{L}$. A join-semilattice is a partial order with a least element $\bot$ and a least upper bound operation $\vee$ ("join"). Finite sets (with the empty set as least element, and union as join) are an example, as are tuples of semilattices. As long as the semilattice has no infinite ascending chains $x_0 < x_1 < x_2 < \cdots$, iteration from the bottom element is guaranteed to find the least fixed point.[2]

Second, we require that the recursive function be boxed, $\Box(\underset{\text{FIX}}{L} \to \underset{\text{FIX}}{L})$. Since boxed expressions can only refer to discrete values, and fixed point functions themselves must be monotone, this has the effect of preventing semantically nested fixed points. We discuss this in more detail in §10. Note that this does not prevent mutual recursion, which can be expressed by taking a fixed point at product type, nor stratified fixed points à la Datalog.

3 DATAFUN BY EXAMPLE

For brevity and clarity, the examples that follow make use of some syntax sugar:

(1) We mentioned earlier that Datafun's boolean type *bool* is ordered *false* $<$ *true*. This is because we encode booleans as sets of empty tuples, $\{1\}$, with *false* being the empty set $\{\}$ and *true* being the singleton $\{()\}$. At semilattice type we also permit a "one-sided" conditional test, **when** (b) e, which yields e if b is *true* and $\bot$ otherwise. Encoding booleans as sets has the advantage that **when** (b) e is monotone in the condition b.

[2] As a technical detail, the finite set type $\{\underset{\text{EQ}}{A}\}$ will possess infinite ascending chains if $\underset{\text{EQ}}{A}$ has infinitely many inhabitants. Thus we need to distinguish a class of *finite* eqtypes $\underset{\text{FIN}}{A}$. Although their grammars in figure 1 are identical, their intent is different. For example, if we extended Datafun with integers, they would form an eqtype, but not a finite one.

(2) We make use of set comprehensions, which can be desugared into the monadic operators **for** and **when** in the usual way [Wadler 1992].

(3) It is convenient to treat *fix* as a binding form, $\textbf{fix}\ x\ \textbf{is}\ e$, rather than explicitly supplying a boxed function, *fix* $[\lambda x.\ e]$.

(4) Finally, we make free use of curried functions and pattern matching. Desugaring these is relatively standard, and so we will say little about it, with one exception: the box-elimination form $\textbf{let}\ [x] = e\ \textbf{in}\ e'$ is a pattern matching form, and so we allow it to occur inside of patterns. The effect of a box pattern $[p]$ is to ensure that all of the variables bound in the pattern p are treated as discrete variables.

We summarize (except for pattern matching) the desugaring rules we use in figure 2.

3.1 Set Operations

Even before higher-order functions, one of the main benefits of Datafun over Datalog is that it permits manipulating relations as first class values. In this subsection we will show how a variety of standard operations on sets can be represented in Datafun. The first operation we consider is testing membership:

$$\begin{aligned}&\mathit{member}\ :\ \Box \underset{\text{EQ}}{A} \to \{\underset{\text{EQ}}{A}\} \to \mathit{bool}\\&\mathit{member}\ [x]\ s = \textbf{for}\ (y \in s)\ x = y\end{aligned}$$

This checks if x is equal to any element $y \in s$. The argument x is discrete because increasing x might send it from being *in* the set to being *outside* the set (e.g. $1 \in \{1\}$ but $2 \notin \{1\}$). Notice that here we're taking advantage of encoding booleans as sets of empty tuples – unioning these sets implements logical *or*.

Using *member* we can define set intersection by taking the union of every singleton set $\{x\}$ where x is an element of both s and t:

$$\begin{aligned}&\_ \cap \_\ :\ \{\underset{\text{EQ}}{A}\} \to \{\underset{\text{EQ}}{A}\} \to \{\underset{\text{EQ}}{A}\}\\&s \cap t = \textbf{for}\ (x \in s)\ \textbf{when}\ (\mathit{member}\ [x]\ t)\ \{x\}\end{aligned}$$

Using comprehensions, this could alternately be written as:

$$s \cap t = \{x \mid x \in s,\ \mathit{member}\ [x]\ t\}$$

From now on, we'll use comprehensions whenever possible. For example, we can also define the composition of two relations in Datafun:

$$\begin{aligned}&\_ \bullet \_\ :\ \{\underset{\text{EQ}}{A} \times \underset{\text{EQ}}{B}\} \to \{\underset{\text{EQ}}{B} \times \underset{\text{EQ}}{C}\} \to \{\underset{\text{EQ}}{A} \times \underset{\text{EQ}}{C}\}\\&s \bullet t = \{(a, c) \mid (a, b_1) \in s,\ (b_2, c) \in t,\ b_1 = b_2\}\end{aligned}$$

This is basically a transcription of the mathematical definition, where we build those pairs which agree on their B-typed components.

We can also define set difference, although we must first detour into boolean negation:

$$\begin{aligned}&\neg\ :\ \Box \mathit{bool} \to \mathit{bool}\\&\neg[t] = \textbf{case}\ \mathit{empty?}\ t\ \textbf{of}\ \mathrm{in}_1\ () \to \mathit{true}\ ;\ \mathrm{in}_2\ () \to \mathit{false}\end{aligned}$$

$$\begin{aligned}&\_ \setminus \_\ :\ \{\underset{\text{EQ}}{A}\} \to \Box\{\underset{\text{EQ}}{A}\} \to \{\underset{\text{EQ}}{A}\}\\&s \setminus [t] = \{x \mid x \in s,\ \neg[\mathit{member}\ [x]\ t]\}\end{aligned}$$

To implement boolean negation, we need the primitive operator $\mathit{empty?}\ e$, which produces a tag indicating whether its argument e (a set of empty tuples, i.e. a boolean) is the empty set. This in turn lets us define set difference, the analogue in Datafun of negation in Datalog. Note that in both boolean negation and set difference the "negated" argument t is boxed, because the operation is not monotone in t. This enforces stratification.

Finally, generalizing the *ancestor* relation from the Datalog program in §2.1, we can define the transitive closure of a relation:

$$\begin{aligned}&\mathit{trans} \;:\; \square\{\underset{\text{EQ}}{A} \times \underset{\text{EQ}}{A}\} \to \{\underset{\text{EQ}}{A} \times \underset{\text{EQ}}{A}\}\\&\mathit{trans}\ [\mathit{edge}] = \mathbf{fix}\ s\ \mathbf{is}\ \mathit{edge} \vee (\mathit{edge} \bullet s)\end{aligned}$$

This definition uses a least fixed point, just like the mathematical definition – a transitive closure is the least relation R containing the original relation *edge* and the composition of *edge* with R. However, one feature of this definition peculiar to Datafun is that the argument type is $\square\{\underset{\text{EQ}}{A} \times \underset{\text{EQ}}{A}\}$; the transitive closure takes a *discrete* relation. This is because we must use the relation within the fixed point, and so its parameter needs to be discrete to occur within. This restriction is artificial – transitive closure is semantically a monotone operation – but we'll see why it's useful in §6.

3.2 Regular Expression Combinators

Datafun permits tightly integrating the higher-order functional and bottom-up logic programming styles. In this section, we illustrate the benefits of doing so by showing how to implement a regular expression matching library in combinator style. Like combinator parsers in functional languages, the code is very concise. However, support for the relational style ensures we can write naïve code *without* the exponential backtracking cliffs typical of parser combinators in functional languages.

For these examples we'll assume the existence of eqtypes *string*, *char*, and *int*, an addition operator $+$, and functions *length* and *chars* satisfying:

$$\begin{aligned}&\mathit{length} \;:\; \square \mathit{string} \to \mathit{int}\\&\mathit{length}\ [s] = \text{the length of } s\end{aligned}$$

$$\begin{aligned}&\mathit{chars} \;:\; \square \mathit{string} \to \{\mathit{int} \times \mathit{char}\}\\&\mathit{chars}\ [s] = \{(i, c) \mid \text{the } i^{\text{th}} \text{ character of } s \text{ is } c\}\end{aligned}$$

Note that by always boxing string arguments, we avoid committing ourselves to any particular partial ordering on *string*.

These assumed, we define the type of regular expression matchers:

$$\mathbf{type}\ \mathit{re} = \square \mathit{string} \to \{\mathit{int} \times \mathit{int}\}$$

A regular expression takes a discrete string $[s]$ and returns the set of all pairs (i, j) such that the substring $s_i, \ldots, s_{j-1}$ matches the regular expression. For example, to find all matches for a single character c, we return the range $(i, i+1)$ whenever $(i, c) \in \mathit{chars}\ [s]$:

$$\begin{aligned}&\mathit{sym} \;:\; \square \mathit{char} \to \mathit{re}\\&\mathit{sym}\ [c]\ [s] = \{(i, i+1) \mid (i, c') \in \mathit{chars}\ [s],\ c = c'\}\end{aligned}$$

To find all matches for the empty regex, i.e. all empty substrings, including the one "beyond the last character":

$$\begin{aligned}&\mathit{nil} \;:\; \mathit{re}\\&\mathit{nil}\ [s] = \{i \mid (i, \_) \in \mathit{chars}\ [s]\} \vee \{\mathit{length}\ [s]\}\end{aligned}$$

Appending regexes r_1, r_2 amounts to relation composition, since we wish to find all substrings consisting of adjacent substrings $s_i \ldots s_{j-1}$ and $s_j \ldots s_{k-1}$ matching r_1 and r_2 respectively:

$$\begin{aligned}&\mathit{seq} \;:\; \mathit{re} \to \mathit{re} \to \mathit{re}\\&\mathit{seq}\ r_1\ r_2\ s = r_1\ s \bullet r_2\ s\end{aligned}$$

Similarly, regex alternation $r_1 \mid r_2$ is accomplished by unioning all matches of each:

$$\begin{aligned} &\mathit{alt} \ : \ \mathit{re} \to \mathit{re} \to \mathit{re} \\ &\mathit{alt}\ r_1\ r_2\ s = r_1\ s \vee r_2\ s \end{aligned}$$

The most interesting regular expression combinator is Kleene star. Thinking relationally, if we consider the set of pairs (i, j) matching some regex r, then $r{\star}$ matches its *reflexive, transitive closure*. This can be accomplished by combining *nil* and *trans*.

$$\begin{aligned} &\mathit{star} \ : \ \Box\mathit{re} \to \mathit{re} \\ &\mathit{star}\ [r]\ [s] = \mathit{nil}\ [s] \vee \mathit{trans}\ [r\ [s]] \end{aligned}$$

Note that the argument r must be discrete because *trans* uses it to compute a fixed point.[3]

3.3 Regular Expression Combinators, Take 2

The combinators in the previous section found *all* matches within a given substring, but often we are not interested in all matches: we only want to know if a string can match starting at a particular location. We can easily refactor the combinators above to work in this style, which illustrates the benefits of tightly integrating functional and relational styles of programming – we can use functions to manage strict input/output divisions, and relations to manage nondeterminism and search.

$$\textbf{type}\ \mathit{re} = \Box(\mathit{string} \times \mathit{int}) \to \{\mathit{int}\}$$

Our new type of combinators takes a string and a starting position, and returns a set of ending positions. For example, *sym* $[c]$ checks if c occurs at the start position i, yielding $\{i + 1\}$ if it does and the empty set otherwise, while *nil* simply returns the start position i.

$$\begin{aligned} &\mathit{sym} \ : \ \Box\mathit{char} \to \mathit{re} \\ &\mathit{sym}\ [c]\ [(s, i)] = \{i + 1 \mid (j, d) \in \mathit{chars}\ [s],\ i = j,\ c = d\} \end{aligned}$$

$$\begin{aligned} &\mathit{nil} \ : \ \mathit{re} \to \mathit{re} \\ &\mathit{nil}\ [(s, i)] = \{i\} \end{aligned}$$

Appending regexes *seq* r_1 r_2 simply applies r_2 starting from every ending position that r_1 can find:

$$\begin{aligned} &\mathit{seq} \ : \ \mathit{re} \to \mathit{re} \to \mathit{re} \\ &\mathit{seq}\ r_1\ r_2\ [(s, i)] = \textbf{for}\ (j \in r_1\ [(s, i)])\ r_2\ [(s, j)] \end{aligned}$$

Regex alternation *alt* is effectively unchanged:

$$\begin{aligned} &\mathit{alt} \ : \ \mathit{re} \to \mathit{re} \to \mathit{re} \\ &\mathit{alt}\ r_1\ r_2\ x = r_1\ x \vee r_2\ x \end{aligned}$$

Finally, Kleene star is implemented by recursively appending r to a set x of matches found so far:

$$\begin{aligned} &\mathit{star} \ : \ \Box\mathit{re} \to \mathit{re} \\ &\mathit{star}\ [r]\ [(s, i)] = \textbf{fix}\ x\ \textbf{is}\ (\{i\} \vee \textbf{for}\ (j \in x)\ r\ [(s, j)]) \end{aligned}$$

It's worth noting that this definition is effectively *left-recursive* – it takes the endpoints from the fixed point x, and then continues matching using the argument r. This should make clear that this is not just plain old functional programming – we are genuinely relying upon the fixed point semantics of Datafun.

[3] Technically the inclusion order on sets of integer pairs does not satisfy the ascending chain condition, so this use of *trans* is not well-typed. However, since the positions in a particular string form a finite set, semantically there is no issue. Arntzenius and Krishnaswami [2016] shows how to define bounded fixed points to handle cases like this, so we will not be scrupulous.

4 FROM SEMINAÏVE EVALUATION TO THE INCREMENTAL λ-CALCULUS

Let's return to our example Datalog program, modified to consider graphs rather than ancestry:

$$\begin{aligned} &\mathit{path}(X, Z) \leftarrow \mathit{edge}(X, Z) \\ &\mathit{path}(X, Z) \leftarrow \mathit{edge}(X, Y) \wedge \mathit{path}(Y, Z) \end{aligned}$$

Suppose *edge* denotes a linear graph, $\{(1,2), (2,3), \ldots, (n-1,n)\}$. Then *path* should denote its reachability relation, $\{(i,j) \mid 1 \leqslant i < j \leqslant n\}$. How can we compute this? The simplest approach is to begin with nothing in the *path* relation and repeatedly apply its rules until nothing more is deducible. We can make this strategy explicit by time-indexing the *path* relation:

$$\begin{aligned} &\mathit{path}_{i+1}(X, Z) \leftarrow \mathit{edge}(X, Z) \\ &\mathit{path}_{i+1}(X, Z) \leftarrow \mathit{edge}(X, Y) \wedge \mathit{path}_i(Y, Z) \end{aligned}$$

By omission $\mathit{path}_0 = \emptyset$. From this inductively $\mathit{path}_i \subseteq \mathit{path}_{i+1}$, because at step $i+1$ we re-deduce every fact known at step i. For example, suppose $\mathit{path}_i(j,k)$ holds. Then at step $i+1$ the second rule deduces $\mathit{path}_{i+1}(j-1,k)$ from $\mathit{edge}(j-1,j) \wedge \mathit{path}_i(j,k)$. But since $\mathit{path}_{i+1}(j,k)$ holds, we perform the same deduction at time $i+2$, and again at $i+3$, $i+4$, etc.

Because we append edges one at a time, path_i contains all paths of i or fewer edges. Therefore it takes n steps until we reach our fixed point $\mathit{path}_{n-1} = \mathit{path}_n$. Since step i involves $|\mathit{path}_i| \in \Theta(i^2)$ deductions, we make $\Theta(n^3)$ deductions in total. There being only $\Theta(n^2)$ paths in the final result, this is terribly wasteful; hence we term this *naïve evaluation*.

Seminaïve evaluation avoids waste by transforming the rules for *path* to find the *newly* deducible paths, dpath_i, at iteration i, and accumulating these changes to produce a final result:

$$\begin{aligned} \mathit{dpath}_0(X, Y) &\leftarrow \mathit{edge}(X, Y) \\ \mathit{dpath}_{i+1}(X, Z) &\leftarrow \mathit{edge}(X, Y) \wedge \mathit{dpath}_i(Y, Z) \\ \mathit{path}_{i+1}(X, Y) &\leftarrow \mathit{path}_i(X, Y) \vee \mathit{dpath}_i(X, Y) \end{aligned}$$

It's easy to show inductively that dpath_i contains only paths exactly $i+1$ edges long. Consequently $|\mathit{dpath}_i| \in \Theta(n-i)$ and we make $\Theta(n^2)$ deductions overall.[4]

4.1 Seminaïve Evaluation as Incremental Computation

Now let's move from Datalog to Datafun.[5] The transitive closure of *edge* is the fixed point of the monotone function *step* defined by:

$$\mathit{step}\ \mathit{path} = \mathit{edge} \cup \{(x,z) \mid (x,y) \in \mathit{edge},\ (y,z) \in \mathit{path}\}$$

The naïve way to compute *step*'s fixed point is to iterate it: start from $\mathit{path}_0 = \emptyset$ and compute $\mathit{path}_{i+1} = \mathit{step}\ \mathit{path}_i$ for increasing i until $\mathit{path}_i = \mathit{path}_{i+1}$. But as before, $\mathit{path}_i \subseteq \mathit{step}\ \mathit{path}_i$; each iteration re-computes the paths found by its predecessor. Following Datalog, we'd prefer to compute only the *change* between iterations. So consider step' defined by:

$$\mathit{step}'\ \mathit{dpath} = \{(x,z) \mid (x,y) \in \mathit{edge},\ (y,z) \in \mathit{dpath}\}$$

[4] Here we must assume the accumulation rule $\mathit{path}_{i+1}(X,Y) \leftarrow \mathit{path}_i(X,Y) \vee \mathit{dpath}_i(X,Y)$ is implemented using an union operator that is efficient when the sets being unioned are of greatly differing sizes.

[5] In this section we do not bother marking discrete variables x or expressions e, as it muddies our examples to no benefit.

Observe that

$$\begin{aligned}
& \mathit{step}\ (\mathit{path} \cup \mathit{dpath}) \\
={}& \mathit{edge} \cup \{(x,z) \mid (x,y) \in \mathit{edge},\ (y,z) \in \mathit{path} \cup \mathit{dpath}\} \\
={}& \mathit{edge} \cup \{(x,z) \mid (x,y) \in \mathit{edge},\ (y,z) \in \mathit{path}\} \cup \{(x,z) \mid (x,y) \in \mathit{edge},\ (y,z) \in \mathit{dpath}\} \\
={}& \mathit{step}\ \mathit{path} \cup \mathit{step}'\ \mathit{dpath}
\end{aligned}$$

In other words, *step′* tells us how *step* changes as its input grows. This lets us directly compute the changes dpath_i between our iterations path_i:

$$\begin{aligned}
\mathit{dpath}_0 &= \mathit{step}\ \emptyset = \mathit{edge} \\
\mathit{dpath}_{i+1} &= \mathit{step}'\ \mathit{dpath}_i = \{(x,z) \mid (x,y) \in \mathit{edge},\ (y,z) \in \mathit{dpath}_i\} \\
\mathit{path}_{i+1} &= \mathit{path}_i \cup \mathit{dpath}_i
\end{aligned}$$

These exactly mirror the derivative and accumulator rules for path_i and dpath_i we gave earlier.

The problem of seminaïve evaluation for Datafun, then, reduces to the problem of finding functions, like *step′*, which compute the change in a function's output given a change to its input. This is a problem of *incremental computation*, and since Datafun is a functional language, we turn to the *incremental λ-calculus* [Cai et al. 2014; Giarrusso et al. 2019].

4.2 Change Structures

To make precise the notion of change, an incremental λ-calculus associates every type A with a *change structure*, consisting of:[6]

(1) A type ΔA of possible changes to values of type A.
(2) A relation $dx ::_A x \rightsquigarrow y$ for $dx : \Delta A$ and $x, y : A$, read as "dx changes x into y".

Since the iterations of a fixed point grow monotonically, in Datafun we only need *increasing* changes. For example, sets change by gaining new elements:

$$\Delta\{\underset{\text{EQ}}{A}\} = \{\underset{\text{EQ}}{A}\} \qquad\qquad dx ::_{\{\underset{\text{EQ}}{A}\}} x \rightsquigarrow x \cup dx$$

Set changes may be the most significant for fixed point purposes, but to handle all of Datafun we need a change structure for every type. For products and sums, for example, the change structure is pointwise:

$$\Delta 1 = 1 \qquad \Delta(A \times B) = \Delta A \times \Delta B \qquad \Delta(A + B) = \Delta A + \Delta B$$

$$() ::_1 () \rightsquigarrow () \qquad \frac{da ::_A a \rightsquigarrow a' \qquad db ::_B b \rightsquigarrow b'}{(da, db) ::_{A \times B} (a, b) \rightsquigarrow (a', b')} \qquad \frac{dx ::_{A_i} x \rightsquigarrow y}{\mathsf{in}_i\ dx ::_{A_1 + A_2} \mathsf{in}_i\ x \rightsquigarrow \mathsf{in}_i\ y}$$

Since we only consider increasing changes, and $\Box A$ is ordered discretely, the only "change" permitted is to stay the same. Consequently, no information is necessary to indicate what changed:

$$\Delta(\Box A) = 1 \qquad\qquad () ::_{\Box A} x \rightsquigarrow x$$

Finally we come to the most interesting case: functions.

$$\Delta(A \to B) = \Box A \to \Delta A \to \Delta B \qquad\qquad \textsc{fn change}\ \frac{(\forall dx ::_A x \rightsquigarrow y)\ df\ x\ dx ::_B f\ x \rightsquigarrow g\ y}{df ::_{A \to B} f \rightsquigarrow g}$$

[6] Our notion of change structure differs significantly from that of Cai et al. [2014], although it is similar to the logical relation given in Giarrusso et al. [2019]; we discuss this in §10. Although we do not use change structures *per se* in the proof of correctness sketched in §7, they are an important source of intuition.

Observe that a function change df takes two arguments: a base point $x : \Box A$ and a change $\mathrm{dx} : \Delta A$. To understand why we need both, consider incrementalizing function application: we wish to know how $f\ x$ changes as both f and x change. Supposing $\mathrm{df} :: f \rightsquigarrow g$ and $\mathrm{dx} :: x \rightsquigarrow y$, how do we find a change $f\ x \rightsquigarrow g\ y$ that updates both function and argument?

If changes were given pointwise, taking only a base point, we'd stipulate that $\mathrm{df} :: f \rightsquigarrow g$ iff $(\forall x)\ \mathrm{df}\ x :: f\ x \rightsquigarrow g\ x$. But this only gets us to $g\ x$, not $g\ y$: we've accounted for the change in the function, but not the argument. We can account for both by giving df an additional parameter: not just the base point x, but also the change dx to it. Then by inverting FN CHANGE we have $\mathrm{df}\ x\ \mathrm{dx} :: f\ x \rightsquigarrow g\ y$ as desired.

Note also the mixture of monotonicity and non-monotonicity in the type $\Box A \to \Delta A \to \Delta B$. Since our functions are monotone (increasing inputs yield increasing outputs), function changes are monotone with respect to input changes ΔA: a larger increase in the input yields a larger increase in the output. However, there's no reason to expect the change in the output to grow as the base point increases – hence the use of $\Box$.

4.3 Zero Changes, Derivatives, and Faster Fixed Points

If $\mathrm{dx} ::_A x \rightsquigarrow x$, we call dx a *zero change* to x. Usually zero changes are boring – for example, a zero change to a set $x : \{\underset{\text{EQ}}{A}\}$ is any $\mathrm{dx} \subseteq x$, and so $\emptyset$ is always a zero change. However, there is one very interesting exception: function zero changes. Suppose $\mathrm{df} ::_{A \to B} f \rightsquigarrow f$. Then inverting FN CHANGE implies that

$$\mathrm{dx} ::_A x \rightsquigarrow y \implies \mathrm{df}\ x\ \mathrm{dx} ::_B f\ x \rightsquigarrow f\ y$$

In other words, df yields the change in the output of f given a change to its input. This is exactly the property of *step′* that made it useful for seminaïve evaluation – indeed, *step′* is a zero change to *step*, modulo not taking the base point x as an argument:

$$\mathrm{dx} ::_{\{\underset{\text{EQ}}{A}\}} x \rightsquigarrow y \implies \mathit{step}'\ \mathrm{dx} ::_{\{\underset{\text{EQ}}{A}\}} \mathit{step}\ x \rightsquigarrow \mathit{step}\ y$$
$$\text{i.e.}$$
$$x \cup \mathrm{dx} = y \implies \mathit{step}\ x \cup \mathit{step}'\ \mathrm{dx} = \mathit{step}\ y$$

Function zero changes are so important we give them a special name: *derivatives*. We now have enough machinery to prove correct a general *seminaïve fixed point strategy*. First, observe that:

LEMMA 4.1. *At every semilattice type* L, *we have* $\Delta L = L$ *and* $\mathrm{dx} ::_L x \rightsquigarrow y \iff (x \vee \mathrm{dx}) = y$.

This holds by a simple induction on semilattice types L. Now, given a monotone map $f : L \to L$ and its derivative $f' : \Box L \to L \to L$, we can find f's fixed-point as the limit of the sequence x_i defined:

$$x_0 = \bot \qquad x_{i+1} = x_i \vee \mathrm{dx}_i$$
$$\mathrm{dx}_0 = f\ \bot \qquad \mathrm{dx}_{i+1} = f'\ x_i\ \mathrm{dx}_i$$

Let $\mathit{semifix}\ (f, f') = \bigvee_i x_i$. By induction and the derivative property, we have $\mathrm{dx}_i :: x_i \rightsquigarrow f\ x_i$ and so $x_i = f^i\ x$, and therefore $\mathit{semifix}\ (f, f') = \mathit{fix}\ f$. Moreover, if L has no infinite ascending chains, we will reach our fixed point $x_i = x_{i+1}$ in a finite number of iterations.

This leads directly to our strategy for seminaïve Datafun. Cai et al. [2014] defines a static transformation $\mathcal{D}\mathit{erive}\ e$ which computes the change in e given the change in its free variables; it *incrementalizes* e. Our goal is not to incrementalize Datafun *per se*, but to find fixed points faster. Consequently, we define two mutually recursive transformations: ϕe, which computes e faster by replacing fixed points with calls to *semifix*; and δe, which incrementalizes ϕe just enough that we can compute the derivative of fixed point functions. In order to define ϕ and δ and show them correct, however, we first need a fuller account of Datafun's type system and semantics.

$$\begin{array}{rrcl} \text{contexts} & \Gamma & ::= & \varepsilon \mid \Gamma, H \\ \text{hypotheses} & H & ::= & x : A \mid x :: A \end{array} \qquad \begin{array}{lcl} \lceil \varepsilon \rceil & = & \varepsilon \\ \lceil \Gamma, x : A \rceil & = & \lceil \Gamma \rceil \\ \lceil \Gamma, x :: A \rceil & = & \lceil \Gamma \rceil, x :: A \end{array}$$

$$\textsc{var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{dvar}\ \frac{x :: A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{lam}\ \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.\ e : A \to B} \qquad \textsc{app}\ \frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash f : A}{\Gamma \vdash e\ f : B} \qquad \textsc{unit}\ \frac{}{\Gamma \vdash () : 1}$$

$$\textsc{pair}\ \frac{(\Gamma \vdash e_i : A_i)_i}{\Gamma \vdash (e_1, e_2) : A_1 \times A_2} \qquad \textsc{prj}\ \frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \pi_i\ e : A_i} \qquad \textsc{inj}\ \frac{\Gamma \vdash e : A_i}{\Gamma \vdash \mathsf{in}_i\ e : A_1 + A_2}$$

$$\textsc{case}\ \frac{\Gamma \vdash e : A_1 + A_2 \qquad (\Gamma, x_i : A_i \vdash f_i : B)_i}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ (\mathsf{in}_i\ x_i \to f_i)_i : B} \qquad \textsc{box}\ \frac{\lceil \Gamma \rceil \vdash e : A}{\Gamma \vdash [e] : \Box A} \qquad \textsc{letbox}\ \frac{\Gamma \vdash e : \Box A \qquad \Gamma, x :: A \vdash f : B}{\Gamma \vdash \textbf{let}\ [x] = e\ \textbf{in}\ f : B}$$

$$\textsc{bot}\ \frac{}{\Gamma \vdash \bot : \underset{\text{EQ}}{L}} \qquad \textsc{join}\ \frac{(\Gamma \vdash e_i : \underset{\text{EQ}}{L})_i}{\Gamma \vdash e_1 \vee e_2 : \underset{\text{EQ}}{L}} \qquad \textsc{set}\ \frac{(\lceil \Gamma \rceil \vdash e_i : \underset{\text{EQ}}{A})_i}{\Gamma \vdash \{e_i\}_i : \{\underset{\text{EQ}}{A}\}} \qquad \textsc{for}\ \frac{\Gamma \vdash e : \{A\} \qquad \Gamma, x :: A \vdash f : \underset{\text{EQ}}{L}}{\Gamma \vdash \textbf{for}\ (x \in e)\ f : \underset{\text{EQ}}{L}}$$

$$\textsc{eq}\ \frac{(\lceil \Gamma \rceil \vdash e_i : \underset{\text{EQ}}{A})_i}{\Gamma \vdash e_1 = e_2 : \mathit{bool}} \qquad \textsc{empty?}\ \frac{\lceil \Gamma \rceil \vdash e : \{1\}}{\Gamma \vdash \mathit{empty?}\ e : 1 + 1} \qquad \textsc{split}\ \frac{\Gamma \vdash e : \Box(A + B)}{\Gamma \vdash \mathit{split}\ e : \Box A + \Box B} \qquad \textsc{fix}\ \frac{\Gamma \vdash e : \Box(\underset{\text{FIX}}{L} \to \underset{\text{FIX}}{L})}{\Gamma \vdash \mathit{fix}\ e : \underset{\text{FIX}}{L}}$$

Fig. 3. Datafun typing rules

5 TYPES AND SEMANTICS

The syntax of core Datafun is given in figure 1 and its typing rules in figure 3. Contexts are lists of hypotheses H; a hypothesis gives the type of either a monotone variable $x : A$ or a discrete variable $x :: A$. The stripping operation $\lceil \Gamma \rceil$ drops all monotone hypotheses from the context Γ, leaving only the discrete ones. The typing judgement $\Gamma \vdash e : A$ may be read as "under hypotheses Γ, the term e has type A".

The VAR and DVAR rules say that both monotone hypotheses $x : A$ and discrete hypotheses $x :: A$ justify ascribing the variable x the type A. The LAM rule is the familiar rule for λ-abstraction. However, note that we introduce the argument variable $x : A$ as a *monotone* hypothesis, not a discrete one. (This is the "right" choice because in **Poset** the exponential object is the poset of monotone functions.) The application rule APP is standard, as are the rules UNIT, PAIR, PRJ, INJ, and CASE. As with LAM, the variables $x_i : A_i$ bound in the case branches f_i are monotone.

BOX says that $[e]$ has type $\Box A$ when e has type A in the stripped context $\lceil \Gamma \rceil$. This restricts e to refer only to discrete variables, ensuring we don't smuggle any information we must treat monotonically into a discretely-ordered $\Box$ expression. The elimination rule LETBOX for ($\textbf{let}\ [x] = e\ \textbf{in}\ f$) allows us to "cash in" a boxed expression $e : \Box A$ by binding its result to a discrete variable $x :: A$ in the body f.

At this point, our typing rules correspond to standard constructive S4 modal logic [Pfenning and Davies 2001]. We get to Datafun by adding a handful of domain-specific types and operations. First, SPLIT provides an operator $\mathit{split} : \Box(A + B) \to \Box A + \Box B$ to distribute box across sum types.[7] The other direction, $\Box A + \Box B \to \Box(A + B)$, is already derivable, as is the isomorphism $\Box A \times \Box B \cong \Box(A \times B)$. This is used implicitly by box pattern-matching – e.g., in the pattern $[(\mathrm{in}_1\, x, \mathrm{in}_2\, y)]$, the variables x and y are both discrete, which is information we propagate via these conversions. Semantically, all of these operations are the identity, as we shall see shortly.

This leaves only the rules for manipulating sets and other semilattices. BOT and JOIN tell us that $\bot$ and $\vee$ are valid at any semilattice type L, that is, at sets and products of semilattice types. The rule for set-elimination, FOR, is *almost* monadic bind. However, we generalize it by allowing **for** $(x \in e)\ f$ to eliminate into any semilattice type, not just sets, denoting a "big semilattice join" rather than a "big union". Finally, the introduction rule SET is says that $\{e_i\}_{i \in I}$ has type $\{\underset{\text{EQ}}{A}\}$ when each of the e_i has type $\underset{\text{EQ}}{A}$. Just as in BOX, each e_i has to typecheck in a stripped context; constructing a set is a discrete operation, since $1 \leqslant 2$ but $\{1\} \not\subseteq \{2\}$.

Likewise discrete is equality comparison $e_1 = e_2$, whose rule EQ is otherwise straightforward; and EMPTY?, which requires more explanation. The idea is that *empty?* e determines whether $e : \{1\}$ is empty, returning $\mathrm{in}_1\,()$ if it is, and $\mathrm{in}_2\,()$ if it isn't. This lets us turn "booleans" (sets of units) into values we can **case**-analyse. This is, however, not monotone, because while booleans are ordered *false* $<$ *true*, sum types are ordered disjointly; $\mathrm{in}_1\,()$ and $\mathrm{in}_2\,()$ are simply incomparable.

Finally, the rule FIX for fixed points *fix* e takes a function $e : \Box(\underset{\text{FIX}}{L} \to \underset{\text{FIX}}{L})$ and yields an expression of type $\underset{\text{FIX}}{L}$. The restriction to "fixtypes" ensures $\underset{\text{FIX}}{L}$ has no infinite ascending chains, guaranteeing the recursion will terminate.

5.1 Semantics

The syntax of core Datafun can be interpreted in **Poset**, the category of partially ordered sets and monotone maps. That is, an object of **Poset** is a pair $(A, \leqslant_A)$ consisting of a set A and a reflexive, transitive, antisymmetric relation $\leqslant_A \subseteq A \times A$, while a morphism $f : A \to B$ is a function such that $x \leqslant_A y \implies f(x) \leqslant_B f(y)$.

5.1.1 Bicartesian Structure. The bicartesian closed structure of **Poset** is largely the same as in **Set**. The product and sum sets are constructed the same way, and ordered pointwise:

$$\begin{aligned} (a, b) \leqslant_{A \times B} (a', b') &\iff a \leqslant_A a' \wedge b \leqslant_B b' \\ \mathrm{in}_i\, x \leqslant_{A_1 + A_2} \mathrm{in}_j\, y &\iff i = j \wedge x \leqslant_{A_i} y \end{aligned}$$

Projections π_i, injections in_i, tupling $\langle f, g \rangle$ and case-analysis $[f, g]$ are all the same as in **Set**, pausing only to note that all these operations preserve monotonicity, as we need.

The exponential $A \Rightarrow B$ consists of only the *monotone* maps $f : A \to B$, again ordered pointwise:

$$f \leqslant_{A \Rightarrow B} g \iff (\forall x \leqslant_A y)\ f\, x \leqslant_B g\, y$$

Currying λ and evaluation are the same as in **Set**. Supposing $f : A \times B \to C$, then:

$$\begin{aligned} &\lambda(f)\ :\ A \to (B \Rightarrow C) &\qquad& \mathit{eval}_{A,B}\ :\ (A \Rightarrow B) \times A \to B \\ &\lambda(f) = x \mapsto y \mapsto f(x, y) && \mathit{eval}_{A,B} = (g, x) \mapsto g(x) \end{aligned}$$

Monotonicity here follows from the monotonicity of f and g and the pointwise ordering of $A \Rightarrow B$.

5.1.2 The Discreteness Comonad. Given a poset $(A, \leqslant_A)$ we define the discreteness comonad $\Box(A, \leqslant_A)$ as $(A, \leqslant_{\Box A})$, where $a \leqslant_{\Box A} a' \iff a = a'$. That is, the discrete order preserves the underlying elements, but reduces the partial order to mere equality. This forms a rather boring

[7] An alternative syntax, pursued in Arntzenius and Krishnaswami [2016], would be to give two rules for **case**, depending on whether or not the scrutinee could be typechecked in a stripped context.

comonad whose functorial action $\square(f)$, extraction $\varepsilon_A : \square A \to A$, and duplication $\delta_A : \square A \to \square\square A$ are all identities on the underlying sets:

$$\square(f) = f \qquad\qquad \varepsilon_A = a \mapsto a \qquad\qquad \delta_A = a \mapsto a$$

This makes the functor and comonad laws trivial. Monotonicity holds in each case because *all* functions are monotone with respect to $\leqslant_{\square A}$. It is also immediate that $\square$ is monoidal with respect to *both* products and coproducts. That is, $\square(A \times B) \cong \square A \times \square B$ and $\square(A + B) \cong \square A + \square B$. In both cases the isomorphism is witnessed by identity on the underlying elements. These lift to n-ary products and sums as well, which we write as $\mathit{dist}^{\times}_{\square} : \prod_i \square A_i \to \square \prod_i A_i$ and $\mathit{dist}^{\square}_{+} : \square \sum_i A_i \to \sum_i \square A_i$.

5.1.3 Sets and Semilattices. Given a poset $(A, \leqslant_A)$ we define the finite powerset poset $P(A, \leqslant_A)$ as $(P_{\text{fin}}\, A, \subseteq)$, with finite subsets of A as elements, ordered by subset inclusion. Note that the subset ordering completely ignores the element ordering $\leqslant_A$. Finite sets admit a pair of useful morphisms:

$$\mathit{singleton} \;:\; \square A \to PA \qquad\qquad \mathit{isEmpty} \;:\; \square PA \to 1 + 1$$

$$\mathit{singleton} = a \mapsto \{a\} \qquad\qquad \mathit{isEmpty} = X \mapsto \begin{cases} \mathrm{in}_1\, () & \text{when } X = \emptyset \\ \mathrm{in}_2\, () & \text{otherwise} \end{cases}$$

The *singleton* function takes a value and makes a singleton set out of it. The domain must be discrete, as otherwise the map will not be monotone (sets are ordered by inclusion, and set membership relies on equality, not the partial order). Similarly, the emptiness test *isEmpty* also takes a discrete set-valued argument, because otherwise the boolean test would not be monotone.

Sets also form a semilattice, with the least element given by the empty set, and join given by union. For this and other semilattices $L \in \mathbf{Poset}$, in particular products of semilattices, we will write $\mathit{join}^L_n : L^n \to L$ to denote the n-ary semilattice join (least upper bound). Also, if $f : A \times \square B \to L$, we can define a morphism $\mathit{collect}(f) : A \times PB \to L$ as follows:

$$\mathit{collect}(f) = (a, X) \mapsto \bigvee_{b \in X} f(a, b)$$

We will use this to interpret **for**-loops. However, it is worth noting that the discreteness restrictions on *singleton* mean that finite sets do not quite form a monad in **Poset**.

5.1.4 Equality. Every object $A \in \mathbf{Poset}$ admits an equality-test morphism *eq*:

$$\mathit{eq} \;:\; \square A \times \square A \to P1$$

$$\mathit{eq} = (x, y) \mapsto \begin{cases} \{()\} & \text{if } x = y \\ \emptyset & \text{otherwise} \end{cases}$$

The domain must be discrete, since $x = y$ and $y \leqslant z$ certainly doesn't imply $x = z$.

5.1.5 Fixed Points. Given a semilattice $L \in \mathbf{Poset}$ without infinite ascending chains, we can define a fixed point operation $\mathit{fix} : (L \to L) \to L$ as follows:

$$\mathit{fix} \;= f \mapsto \bigvee_{n \in \mathbb{N}} f^n(\bot)$$

A routine inductive argument shows this must yield a least fixed point.

5.1.6 Interpretation. The semantic interpetation (defined over typing derivations) is given in figure 4. The interpretation itself mostly follows the usual interpretation for constructive S4 [Alechina et al. 2001], with what novelty there is occuring in the interpretation of sets and fixed points. Even there, the semantics is straightforward, making fairly direct use of the combinators defined above.

TYPE AND CONTEXT DENOTATIONS

$$\begin{array}{rclcrcl}
\llbracket 1 \rrbracket & = & 1 & \qquad & \llbracket A \to B \rrbracket & = & \llbracket A \rrbracket \Rightarrow \llbracket B \rrbracket \\
\llbracket \{\underset{\text{EQ}}{A}\} \rrbracket & = & P\llbracket \underset{\text{EQ}}{A} \rrbracket & & \llbracket A \times B \rrbracket & = & \llbracket A \rrbracket \times \llbracket B \rrbracket \\
\llbracket \Box A \rrbracket & = & \Box \llbracket A \rrbracket & & \llbracket A + B \rrbracket & = & \llbracket A \rrbracket + \llbracket B \rrbracket
\end{array}$$

$$\llbracket \Gamma \rrbracket = \prod_{H \in \Gamma} \llbracket H \rrbracket \qquad \llbracket x : A \rrbracket = \llbracket A \rrbracket \qquad \llbracket \mathit{x} :: A \rrbracket = \Box \llbracket A \rrbracket \qquad \llbracket \Gamma \vdash A \rrbracket = \mathbf{Poset}(\llbracket \Gamma \rrbracket, \llbracket A \rrbracket)$$

TERM DENOTATIONS

$$\begin{array}{lcll}
\llbracket \Gamma \vdash \mathit{x} : A \rrbracket & = & \pi_{\mathit{x}} \cdot \varepsilon & (\text{for } \mathit{x} :: A \in \Gamma) \\
\llbracket \Gamma \vdash x : A \rrbracket & = & \pi_x & (\text{for } x : A \in \Gamma) \\
\llbracket \Gamma \vdash \lambda x.\ e : A \to B \rrbracket & = & \lambda \llbracket \Gamma, x : A \vdash e : B \rrbracket & \\
\llbracket \Gamma \vdash f\ e : B \rrbracket & = & \langle \llbracket \Gamma \vdash f : A \to B \rrbracket, \llbracket \Gamma \vdash e : A \rrbracket \rangle \cdot \mathit{eval} & \\
\llbracket \Gamma \vdash (e_1, e_2) : A_1 \times A_2 \rrbracket & = & \langle \llbracket \Gamma \vdash e_1 : A_1 \rrbracket, \llbracket \Gamma \vdash e_2 : A_2 \rrbracket \rangle & \\
\llbracket \Gamma \vdash \pi_i\ e : A_i \rrbracket & = & \llbracket \Gamma \vdash e : A_1 \times A_2 \rrbracket \cdot \pi_i & \\
\llbracket \Gamma \vdash [e] : \Box A \rrbracket & = & \mathit{box}_\Gamma(\llbracket \lceil \Gamma \rceil \vdash e : A \rrbracket) & \\
\llbracket \Gamma \vdash \mathbf{let}\ [x] = e\ \mathbf{in}\ f : B \rrbracket & = & \langle \mathit{id}_{\llbracket \Gamma \rrbracket}, \llbracket \Gamma \vdash e : \Box A \rrbracket \rangle \cdot \llbracket \Gamma, \mathit{x} :: A \vdash f : B \rrbracket & \\
\llbracket \Gamma \vdash \bot : L \rrbracket & = & \langle \rangle \cdot \mathit{join}_0^L & \\
\llbracket \Gamma \vdash e \vee f : L \rrbracket & = & \langle \llbracket \Gamma \vdash e : L \rrbracket, \llbracket \Gamma \vdash f : L \rrbracket \rangle \cdot \mathit{join}_2^L & \\
\llbracket \Gamma \vdash \mathit{empty?}\ e : 1 + 1 \rrbracket & = & \mathit{box}_\Gamma(\llbracket \lceil \Gamma \rceil \vdash e : \{1\} \rrbracket) \cdot \mathit{isEmpty} & \\
\llbracket \Gamma \vdash \mathit{split}\ e : \Box A + \Box B \rrbracket & = & \llbracket \Gamma \vdash e : \Box(A + B) \rrbracket \cdot \mathit{dist}_+^\Box & \\
\llbracket \Gamma \vdash e_1 = e_2 : \mathit{bool} \rrbracket & = & \langle \mathit{box}_\Gamma(\llbracket \lceil \Gamma \rceil \vdash e_1 : \underset{\text{EQ}}{A} \rrbracket), \mathit{box}_\Gamma(\llbracket \lceil \Gamma \rceil \vdash e_2 : \underset{\text{EQ}}{A} \rrbracket) \rangle \cdot \mathit{eq} & \\
\llbracket \Gamma \vdash \mathit{fix}\ e : \underset{\text{FIX}}{L} \rrbracket & = & \llbracket \Gamma \vdash e : \Box(\underset{\text{FIX}}{L} \to \underset{\text{FIX}}{L}) \rrbracket \cdot \varepsilon \cdot \mathit{fix} & \\
\llbracket \Gamma \vdash \{e_i\}_i : \{\underset{\text{EQ}}{A}\} \rrbracket & = & \langle \mathit{box}_\Gamma(\llbracket \lceil \Gamma \rceil \vdash e_i : \underset{\text{EQ}}{A} \rrbracket) \cdot \mathit{singleton} \rangle_i \cdot \mathit{join}^L & \\
\llbracket \Gamma \vdash \mathbf{for}\ (\mathit{x} \in e)\ f : L \rrbracket & = & \langle \mathit{id}_{\llbracket \Gamma \rrbracket}, \llbracket \Gamma \vdash e : \{\underset{\text{EQ}}{A}\} \rrbracket \rangle \cdot \mathit{collect}(\llbracket \Gamma, \mathit{x} :: \underset{\text{EQ}}{A} \vdash f : L \rrbracket) & \\
\llbracket \Gamma \vdash \mathrm{in}_i\ e : A_1 + A_2 \rrbracket & = & \llbracket \Gamma \vdash e : A_i \rrbracket \cdot \mathrm{in}_i & \\
\llbracket \Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ (\mathrm{in}_i\ x_i \to f_i)_i : B \rrbracket & = & \langle \mathit{id}_{\llbracket \Gamma \rrbracket}, \llbracket \Gamma \vdash e : A_1 + A_2 \rrbracket \rangle \cdot \mathit{dist}_+^\times \cdot \left[\llbracket \Gamma, x_i : A_i \vdash f_i : B \rrbracket \right]_i &
\end{array}$$

AUXILLIARY OPERATIONS

$$\begin{array}{lcl}
\mathit{dist}_+^\times \ :\ A \times (B_1 + B_2) \to (A \times B_1) + (A \times B_2) & \qquad & \mathit{box}_\Gamma \ :\ \llbracket \lceil \Gamma \rceil \vdash A \rrbracket \to \llbracket \Gamma \vdash \Box A \rrbracket \\
\mathit{dist}_+^\times = \langle \pi_2 \cdot [\lambda(\langle \pi_2, \pi_1 \rangle \cdot \mathrm{in}_i)]_i, \pi_1 \rangle \cdot \mathit{eval} & & \mathit{box}_\Gamma(f) = \langle \pi_{\mathit{x}} \cdot \delta \rangle_{\mathit{x} :: A \in \Gamma} \cdot \mathit{dist}_\Box^\times \cdot \Box(f)
\end{array}$$

Fig. 4. Semantics of Datafun

We give the interpretation in combinatory style, and to increase readability, we freely use n-ary products to elide the book-keeping associated with reassociating binary products.

Regarding notation, we write composition in diagrammatic or "pipeline" order with a simple centered dot, letting $f \cdot g : A \to C$ mean $f : A \to B$ followed by $g : B \to C$. If $f_i : A \to B_i$ then we write $\langle f_i \rangle_i : A \to \prod_i B_i$ for the "tupling map" such that $\langle f_i \rangle_i \cdot \pi_j = f_j$. In particular, $\langle \rangle$ is the map into the terminal object. Dually, if $g_i : A_i \to B$ then we write $[g_i]_i : \sum_i A_i \to B$ for the "case-analysis map" such that $\mathrm{in}_j \cdot [g_i]_i = g_j$.

$$\frac{}{\varepsilon \sqsubseteq \varepsilon}\ \textsc{empty} \qquad \frac{\Gamma \sqsubseteq \Delta}{\Gamma, H \sqsubseteq \Delta, H}\ \textsc{cons} \qquad \frac{\Gamma \sqsubseteq \Delta}{\Gamma \sqsubseteq \Delta, H}\ \textsc{drop} \qquad \frac{\Gamma \sqsubseteq \Delta}{\Gamma, x : A \sqsubseteq \Delta, x :: A}\ \textsc{disc}$$

Fig. 5. Weakening relation

5.2 Metatheory

If we were presenting core Datafun in isolation, the usual thing to do would be to prove the soundness of syntactic substitution, show that syntactic and semantic substitution agree, and then establish the equational theory. However, that is not our goal in this paper. We want to prove the correctness of the seminaïve translation, which we will do with a logical relations argument. Since we can harvest almost all the properties we need from the logical relation, only a small residue of metatheory needs to be established manually – indeed, the only thing we need to prove at this stage is the type-correctness of weakening, which we will need to show the type-correctness of the seminaïve transformation.

We define the weakening relation $\Gamma \sqsubseteq \Delta$ in figure 5. This says that Δ is a weakening of Γ, either because it has extra hypotheses (DROP), or because a hypothesis in Γ becomes discrete in Δ (DISC). The idea is that making a hypothesis discrete only increases the number of places it can be used.

Lemma 5.1. *If* $\Gamma \vdash e : A$ *and* $\Gamma \sqsubseteq \Delta$ *then* $\Delta \vdash e : A$.

This follows by the usual induction on typing derivations.

6 THE ϕ AND δ TRANSFORMATIONS

We use two static transformations, ϕ and δ, defined in figures 7 and 8 respectively. Rather than dive into the gory details immediately, we first build some intuition.

The speed-up transform ϕe computes fixed points seminaïvely by replacing *fix* f by *semifix* (f, f'). But to find the derivative f' of f we'll need a second transform, called δe. Since a derivative is a zero change, can δe simply find a zero change to e? Unfortunately, this is not strong enough. For example, the derivative of $\lambda x.\ e$ depends on how e changes as its free variable x changes – which is not necessarily a zero change. To compute derivatives, we need to solve the general problem of computing *changes*. So, modelled on the incremental λ-calculus' *Derive* [Cai et al. 2014], δe will compute how ϕe changes as its free variables change.

However, to speed up *fix* f we don't want the change to f; we want its derivative. Since derivatives are zero changes, function changes and derivatives coincide if *the function cannot change*. This is why the typing rule for *fix* f requires that $f : \Box(L_{\text{FIX}} \to L_{\text{FIX}})$: the use of $\Box$ prevents f from changing! So the key strategy of our speed-up transformation is to **decorate expressions of type □A with their zero changes.** This makes derivatives available exactly where we need them: at *fix* expressions.

6.1 Typing ϕ and δ

In order to decorate expressions with extra information, ϕ also needs to decorate their types. In figure 6 we give a type translation ΦA capturing this. In particular, if $e : \Box A$ then ϕe will have type $\Phi(\Box A) = \Box(\Phi A \times \Delta\Phi A)$. The idea is that evaluating ϕe will produce a pair $[(x, dx)]$ where $x : \Phi A$ is the sped-up result and $dx : \Delta\Phi A$ is a zero change to x. Thus, if $e : \Box(L_{\text{FIX}} \to L_{\text{FIX}})$, then ϕe will compute $[(f, f')]$, where f' is the derivative of f.

On types other than $\Box A$, there is no information we need to add, so Φ simply distributes. In particular, source programs and sped-up programs agree on the shape of first-order data:

$$
\begin{aligned}
\Phi 1 &= 1 & \Delta 1 &= 1\\
\Phi\{\underset{\text{EQ}}{A}\} &= \{\Phi \underset{\text{EQ}}{A}\} \quad \text{(see lemma 6.1)} & \Delta\{\underset{\text{EQ}}{A}\} &= \{\underset{\text{EQ}}{A}\}\\
\Phi(\Box A) &= \Box(\Phi A \times \Delta\Phi A) & \Delta(\Box A) &= 1\\
\Phi(A \times B) &= \Phi A \times \Phi B & \Delta(A \times B) &= \Delta A \times \Delta B\\
\Phi(A + B) &= \Phi A + \Phi B & \Delta(A + B) &= \Delta A + \Delta B\\
\Phi(A \to B) &= \Phi A \to \Phi B & \Delta(A \to B) &= \Box A \to \Delta A \to \Delta B
\end{aligned}
$$

Fig. 6. Δ and Φ type transformations

$$
\begin{aligned}
\phi x &= x & \phi x &= x\\
\phi(\lambda x.\ e) &= \lambda x.\ \phi e & \phi(e\ f) &= \phi e\ \phi f\\
\phi(e_i)_i &= (\phi e_i)_i & \phi(\pi_i\ e) &= \pi_i\ \phi e\\
\phi(\mathrm{in}_i\ e) &= \mathrm{in}_i\ \phi e & \phi(\mathbf{case}\ e\ \mathbf{of}\ (\mathrm{in}_i\ x \to f_i)_i) &= \mathbf{case}\ \phi e\ \mathbf{of}\ (\mathrm{in}_i\ x \to \phi f_i)_i\\
\phi\bot &= \bot & \phi(e \vee f) &= \phi e \vee \phi f\\
\phi(\{e_i\}_i) &= \{\phi e_i\}_i & \phi(\mathbf{for}\ (x \in e)\ f) &= \mathbf{for}\ (x \in \phi e)\ \mathbf{let}\ [dx] = [\mathbf{0}\ x]\ \mathbf{in}\ \phi f\\
\phi[e] &= [(\phi e, \delta e)] & \phi(\mathbf{let}\ [x] = e\ \mathbf{in}\ f) &= \mathbf{let}\ [(x, dx)] = \phi e\ \mathbf{in}\ \phi f\\
\phi(e = f) &= (\phi e = \phi f) & \phi(\mathit{empty?}\ e) &= \mathit{empty?}\ \phi e\\
\phi(\mathit{fix}\ e) &= \mathit{semifix}\ \phi e & \phi(\mathit{split}\ e) &= \mathbf{case}\ \phi e\ \mathbf{of}\\
& & &\quad ([(\mathrm{in}_i\ x, \mathrm{in}_i\ dx)] \to \mathrm{in}_i\ [(x, dx)])_i\\
& & &\quad ([(\mathrm{in}_i\ x, \mathrm{in}_j\ \_)] \to \mathrm{in}_i\ [(x, \mathit{dummy}\ x)])_{i \neq j}
\end{aligned}
$$

Fig. 7. Seminaïve speed-up translation, ϕ

$$
\delta\bot = \delta\{e_i\}_i = \delta(e = f) = \delta(\mathit{fix}\ e) = \bot
$$

$$
\begin{aligned}
\delta x &= dx & \delta x &= dx\\
\delta(\lambda x.\ e) &= \lambda[x].\ \lambda dx.\ \delta e & \delta(e\ f) &= \delta e\ [\phi e]\ \delta f\\
\delta(e_i)_i &= (\delta e_i)_i & \delta(\pi_i\ e) &= \pi_i\ \delta e\\
\delta(\mathrm{in}_i\ e) &= \mathrm{in}_i\ \delta e & \delta(e \vee f) &= \delta e \vee \delta f\\
\delta[e] &= () & \delta(\mathbf{let}\ [x] = e\ \mathbf{in}\ f) &= \mathbf{let}\ [(x, dx)] = \phi e\ \mathbf{in}\ \delta f\\
\delta(\mathit{empty?}\ e) &= \mathit{empty?}\ \phi e & \delta(\mathit{split}\ e) &= \mathbf{case}\ \phi e\ \mathbf{of}\ ([(\mathrm{in}_i\ \_, \_)] \to \mathrm{in}_i\ ())_i
\end{aligned}
$$

$$
\begin{aligned}
\delta(\mathbf{case}\ e\ \mathbf{of}\ (\mathrm{in}_i\ x \to f_i)_i) &= \mathbf{case}\ \mathit{split}\ [\phi e],\ \delta e\ \mathbf{of}\\
&\quad (\mathrm{in}_i\ [x],\ \mathrm{in}_i\ dx \to \delta f_i)_i\\
&\quad (\mathrm{in}_i\ [x],\ \mathrm{in}_j\ \_ \to \mathbf{let}\ dx = \mathit{dummy}\ x\ \mathbf{in}\ \delta f_i)_{i \neq j}\\
\delta(\mathbf{for}\ (x \in e)\ f) &= (\mathbf{for}\ (x \in \delta e)\ \mathbf{let}\ [dx] = \mathbf{0}\ x\ \mathbf{in}\ \phi f)\\
&\quad \vee (\mathbf{for}\ (x \in \phi e \vee \delta e)\ \mathbf{let}\ [dx] = \mathbf{0}\ x\ \mathbf{in}\ \delta f)
\end{aligned}
$$

Fig. 8. Seminaïve derivative translation, δ

Lemma 6.1. $\Phi \underset{\text{EQ}}{A} = \underset{\text{EQ}}{A}$.

This is easily seen by induction on $\underset{\text{EQ}}{A}$.

As we'll see in §6.3 and 6.4, ϕ and δ are mutually recursive. To make this work, δe must find the change to ϕe rather than e. So if $e : A$ then $\phi e : \Phi A$ and $\delta e : \Delta\Phi A$. However, so far we have neglected to say what ϕ and δ do to typing contexts. To understand this, it's helpful to look at what Φ and $\Delta\Phi$ do to functions and to $\square$. This is because expressions denote functions of their free variables. Moreover, in Datafun free variables come in two flavors, monotone and discrete, and discrete variables are semantically $\square$-ed.

Viewed as functions of their free variables, δe denotes the *derivative* of ϕe. And just as the derivative of a unary function $f\ x$ has *two* arguments, $df\ x\ dx$, the derivative of an expression e with n variables $x_1, \ldots, x_n$ will have $2n$ variables: the original $x_1, \ldots, x_n$ and their changes $dx_1, \ldots, dx_n$.[8] However, this says nothing yet about monotonicity or discreteness. To make this precise, we'll use three context transformations, named according to the analogous type operators $\square$, Φ, and Δ:

$$
\begin{aligned}
\square(x : A) &= x :: A & \qquad \square(x :: A) &= x :: A \\
\Phi(x : A) &= x : \Phi A & \qquad \Phi(x :: A) &= x :: \Phi A, dx :: \Delta\Phi A \\
\Delta(x : A) &= dx : \Delta A & \qquad \Delta(x :: A) &= \varepsilon \quad \text{(the empty context)}
\end{aligned}
$$

Otherwise all three operators distribute; e.g. $\square\varepsilon = \varepsilon$ and $\square(\Gamma_1, \Gamma_2) = \square\Gamma_1, \square\Gamma_2$. Intuitively, $\square\Gamma$, $\Phi\Gamma$, and $\Delta\Gamma$ mirror the effect of $\square$, Φ, and Δ on the semantics of Γ:

$$
[\![\square\Gamma]\!] \cong \square[\![\Gamma]\!] \qquad
\begin{aligned}
[\![\Phi(x : A)]\!] &\cong [\![\Phi A]\!] \\
[\![\Phi(x :: A)]\!] &\cong [\![\Phi\square A]\!]
\end{aligned}
\qquad
\begin{aligned}
[\![\Delta(x : A)]\!] &\cong [\![\Delta A]\!] \\
[\![\Delta(x :: A)]\!] &\cong [\![\Delta\square A]\!]
\end{aligned}
$$

These defined, we can state the types of ϕe and δe:

Theorem 6.2 (well-typedness). *If $\Gamma \vdash e : A$, then ϕe and δe have the following types:*

$$
\begin{gathered}
\Phi\Gamma \vdash \phi e : \Phi A \\
\square\Phi\Gamma, \Delta\Phi\Gamma \vdash \delta e : \Delta\Phi A
\end{gathered}
$$

As expected, if we view expressions as functions of their free variables, and pretend Γ is a type, these correspond to $\Phi(\Gamma \to A)$ and $\Delta\Phi(\Gamma \to A)$ respectively:

$$
\Phi(\Gamma \to A) = \Phi\Gamma \to \Phi A \qquad\qquad \Delta\Phi(\Gamma \to A) = \square\Phi\Gamma \to \Delta\Phi\Gamma \to \Delta\Phi A
$$

To get the hang of these context and type transformations, suppose $x :: A, y : B \vdash e : C$. Then theorem 6.2 tells us:

$$
\begin{gathered}
x :: \Phi A,\ dx :: \Delta\Phi A,\ y : \Phi B \vdash \phi e : \Phi C \\
x :: \Phi A,\ dx :: \Delta\Phi A,\ y :: \Phi B, dy : \Delta\Phi B \vdash \delta e : \Delta\Phi C
\end{gathered}
$$

Along with the original program's variables, ϕe requires zero change variables dx for every discrete source variable x. Meanwhile, δe requires changes for *every* source program variable (for discrete variables these will be zero changes), and moreover is *discrete* with respect to the source program variables (the "base points").

We now have enough information to tackle the definitions of ϕ and δ given in figures 7 and 8. In the remainder of this section, we'll examine the most interesting and important parts of these definitions in detail.

[8] For notational convenience we assume that source programs contain no variables starting with the letter *d*.

6.2 Fixed Points

The whole purpose of ϕ and δ is to speed up fixed points, so let's start there. In a fixed point expression *fix* e, we know $e : \Box(L_{\text{FIX}} \to L_{\text{FIX}})$. Consequently the type of ϕe is

$$\begin{aligned}
\Phi(\Box(L_{\text{FIX}} \to L_{\text{FIX}})) &= \Box(\Phi(L_{\text{FIX}} \to L_{\text{FIX}}) \times \Delta\Phi(L_{\text{FIX}} \to L_{\text{FIX}}))\\
&= \Box((\Phi L_{\text{FIX}} \to \Phi L_{\text{FIX}}) \times (\Box\Phi L_{\text{FIX}} \to \Delta\Phi L_{\text{FIX}} \to \Delta\Phi L_{\text{FIX}}))\\
&= \Box((L_{\text{FIX}} \to L_{\text{FIX}}) \times (\Box L_{\text{FIX}} \to \Delta L_{\text{FIX}} \to \Delta L_{\text{FIX}}) && \text{by lemma 6.1, } \Phi L_{\text{FIX}} = L_{\text{FIX}}\\
&= \Box((L_{\text{FIX}} \to L_{\text{FIX}}) \times (\Box L_{\text{FIX}} \to L_{\text{FIX}} \to L_{\text{FIX}}) && \text{by lemma 4.1, } \Delta L_{\text{FIX}} = L_{\text{FIX}}
\end{aligned}$$

The behavior of ϕe is to compute a boxed pair $[(f, f')]$, where $f : L_{\text{FIX}} \to L_{\text{FIX}}$ is a sped-up function and $f' : \Box L_{\text{FIX}} \to L_{\text{FIX}} \to L_{\text{FIX}}$ is its derivative. This is exactly what we need to call *semifix*. Therefore $\phi(\textit{fix}\ e) = \textit{semifix}\ \phi e$. However, if we're going to use *semifix* in the output of ϕ, we ought to give it a typing rule and semantics:

$$\frac{\Gamma \vdash e : \Box((L_{\text{FIX}} \to L_{\text{FIX}}) \times (\Box L_{\text{FIX}} \to L_{\text{FIX}} \to L_{\text{FIX}})}{\Gamma \vdash \textit{semifix}\ e : L_{\text{FIX}}} \qquad \begin{aligned}[\![\textit{semifix}\ e]\!]\ \gamma &= \textit{semifix}\ (f, f')\\ \text{where } (f, f') &= [\![e]\!]\ \gamma\end{aligned}$$

As for $\delta(\textit{fix}\ e)$, since e can't change (having $\Box$ type), neither can *fix* e (or *semifix* ϕe). All we need is a zero change at type L_{FIX}; by lemma 4.1, $\bot$ suffices.

6.3 Variables, λ, and Application

At the core of a functional language are variables, λ, and application. The ϕ translation leaves these alone, simply distributing over subexpressions. On variables, δ yields the corresponding change variables. On functions and application, δ is more interesting:

$$\Delta\Phi(A \to B) = \Box\Phi A \to \Delta\Phi A \to \Delta\Phi B \qquad \delta(\lambda x.\ e) = \lambda[x].\ \lambda dx.\ \delta e \qquad \delta(e\ f) = \delta e\ [\phi f]\ \delta f$$

The intuition behind $\delta(\lambda x.\ e) = \lambda[x].\ \lambda dx.\ \delta e$ is that a function change takes two arguments, a base point x and a change dx, and yields the change in the result of the function, δe. However, we are given an argument of type $\Box\Phi A$, but consulting theorem 6.2 for the type of δe, we need a discrete variable $x :: \Phi A$, so we use pattern-matching to unbox our argument.

The intuition behind $\delta(e\ f) = \delta e\ [\phi f]\ \delta f$ is much the same: δe needs two arguments, the original input ϕf and its change δf, to return the change in the function's output. Moreover, it's discrete in its first argument, so we need to box it, $[\phi f]$.

One might wonder why this type-checks, since ϕe and δe don't use the same typing context. We're even boxing ϕf, hiding all monotone variables; consequently, it gets the context $\lceil \Box\Phi\Gamma, \Delta\Phi\Gamma \rceil$. However, $\Box$ makes every variable discrete, and $\lceil - \rceil$ leaves discrete variables alone, so this includes *at least* $\Box\Phi\Gamma$. The context ϕf needs is $\Phi\Gamma$. Since $\Box$ only makes a context stronger (recalling our definition of weakening from §5.2), we're safe. The same argument applies (all the more easily) when ϕe is used in a monotone rather than a discrete position.

6.4 The Discreteness Comonad, □

Our strategy hinges on decorating expressions of type $\Box A$ with their zero changes, so the translations of $[e]$ and $(\textbf{let}\ [x] = e\ \textbf{in}\ f)$ are of particular interest. The most trivial of these is $\delta[e] = ()$; this follows from $\Delta\Phi\Box A = 1$, since boxed values cannot change.

Next, consider $\phi[e] = [(\phi e, \delta e)]$. The intuition here is straightforward: ϕ needs to decorate e with its zero change; since e is discrete and cannot change, we use δe. However! In general, one cannot use δ inside the ϕ translation and expect the result to be well-typed; ϕ and δ require different typing contexts. To see this, let's apply theorem 6.2 to singleton contexts:

$$\begin{aligned}
\mathit{dummy}_{\{A\}_{\mathrm{EQ}}}\ \_ &= \bot & \mathit{dummy}_{A \times B}\ (x, y) &= (\mathit{dummy}\ x, \mathit{dummy}\ y) \\
\mathit{dummy}_{1}\ () &= () & \mathit{dummy}_{A+B}\ (\mathrm{in}_i\ x) &= \mathrm{in}_i\ (\mathit{dummy}\ x) \\
\mathit{dummy}_{\Box A}\ [x] &= [\mathit{dummy}\ x] & \mathit{dummy}_{A \to B}\ f &= \lambda x.\ \mathit{dummy}\ (f\ x)
\end{aligned}$$

Fig. 9. The function $\mathit{dummy}_A : A \to \Delta A$

Γ (context of e)	$\Phi\Gamma$ (context of ϕe)	$\Box\Phi\Gamma, \Delta\Phi\Gamma$ (context of δe)
$x : A$	$x : \Phi A$	$x :: \Phi A, dx : \Delta\Phi A$
$x :: A$	$x :: \Phi A, dx :: \Delta\Phi A$	$x :: \Phi A, dx :: \Delta\Phi A$

Luckily, although $\Phi\Gamma$ and $\Box\Phi\Gamma, \Delta\Phi\Gamma$ differ on monotone variables, they agree on discrete ones. And since e is discrete, it *has* no free monotone variables, justifying the use of δe in $\phi[e] = [(\phi e, \delta e)]$.

Next we come to $(\mathbf{let}\ [x] = e\ \mathbf{in}\ f)$, whose ϕ and δ translations are very similar:

$$\begin{aligned}
\phi(\mathbf{let}\ [x] = e\ \mathbf{in}\ f) &= \mathbf{let}\ [(x, dx)] = \phi e\ \mathbf{in}\ \phi f \\
\delta(\mathbf{let}\ [x] = e\ \mathbf{in}\ f) &= \mathbf{let}\ [(x, dx)] = \phi e\ \mathbf{in}\ \delta f
\end{aligned}$$

Since x is a discrete variable, both ϕf and δf need access to its zero change dx. Luckily, $\phi e : \Box(\Phi A \times \Delta\Phi A)$ provides it, so we simply unpack it. We don't use δe in δf, but this is unsurprising when you consider that its type is $\Delta\Phi\Box A = 1$.

6.5 Case Analysis, *split*, and *dummy*

The derivative of case-analysis, $\delta(\mathbf{case}\ e\ \mathbf{of}\ (\mathrm{in}_i\ x_i \to f_i)_i)$, is complex. Suppose ϕe evaluates to $\mathrm{in}_i\ x$ and its change δe evaluates to $\mathrm{in}_j\ dx$. Since δe is a change to ϕe, the change structure on sums tells us that $i = j$! (This is because sums are ordered disjointly; the value x can increase, but the tag in_i must remain the same.) So the desired change $\delta(\mathbf{case}\ e\ \mathbf{of} \dots)$ is given by δf_i in a context supplying a discrete base point x (the value x) and the change dx. To bind x discretely, we need to use $[\phi e] : \Box(\Phi A + \Phi B)$; to pattern-match on this, we need *split* to distribute the $\Box$.

This handles the first two cases, $(\mathrm{in}_i\ [x], \mathrm{in}_i\ dx \to \delta f_i)_i$. Since we know the tags on ϕe and δe agree, these are the only possible cases. However, to appease our type-checker we must handle the *impossible* case that $i \neq j$. This case is dead code: it needs to typecheck, but is otherwise irrelevant. It suffices to generate a dummy change $dx : \Delta\Phi A_i$ from our base point $x :: \Phi A_i$. We do this using a simple function $\mathit{dummy}_A : A \to \Delta A$ (figure 9).

We also need *dummy* in the definition of $\phi(\mathit{split}\ e)$. In effect $\mathit{split} : \Box(A + B) \to \Box A + \Box B$. Observe that

$$\begin{aligned}
\Phi(\Box(A + B)) &= \Box((\Phi A + \Phi B) \times (\Delta\Phi A + \Delta\Phi B)) \\
\Phi(\Box A + \Box B) &= \Box(\Phi A \times \Delta\Phi A) + \Box(\Phi B \times \Delta\Phi B)
\end{aligned}$$

So while ϕe yields a boxed pair of tagged values, $[(\mathrm{in}_i\ x, \mathrm{in}_j\ dx)]$, we need $\phi(\mathit{split}\ e)$ to yield a tagged boxed pair, $\mathrm{in}_i\ [(x, dx)]$. Again we use *dummy* to handle the impossible case $i \neq j$.

Finally, observe that $\delta(\mathit{split}\ e)$ has type $\Delta\Phi(\Box A + \Box B) = \Delta\Phi\Box A + \Delta\Phi\Box B = 1 + 1$. All it must do is return $(\mathrm{in}_i\ ())$ with a tag that matches $\phi(\mathit{split}\ e)$ and ϕe; **case**-analysing ϕe suffices.

6.6 Semilattices and Comprehensions

The translation $\phi(e \vee f) = \phi e \vee \phi f$ is as simple as it seems. However, $\delta(e \vee f) = \delta e \vee \delta f$ is mildly clever. Restricting to sets, suppose that dx changes x into x' and dy changes y to y'. In particular, suppose these changes are *precise*: that $dx = x' \setminus x$ and $dy = y' \setminus y$. Then the precise change from $x \cup y$ into $x' \cup y'$ is:

$$(x' \cup y') \setminus (x \cup y) = (x' \setminus x \setminus y) \cup (y' \setminus y \setminus x) = (dx \setminus y) \cup (dy \setminus x)$$

This suggests letting $\delta(e \cup f) = (\delta e \setminus \phi f) \cup (\delta f \setminus \phi e)$. This is a valid derivative, but it involves recomputing ϕe and ϕf, and our goal is to avoid recomputation. So instead, we *overapproximate* the derivative: $\delta e \cup \delta f$ might contain some unnecessary elements, but we expect it to be cheaper to include these than to recompute ϕe and ϕf. This overapproximation agrees with seminaïve evaluation in Datalog: Datalog implicitly unions the results of different rules for the same predicate (e.g. those for *path* in §4), and the seminaïve translations of these rules do not include negated premises to compute a more precise difference.

Now let's consider **for** $(x \in e)\ f$. Its ϕ-translation is straightforward, with one hitch: because $x :: \underset{\text{EQ}}{A}$ is a discrete variable, the inner loop ϕf needs access to its zero change $dx :: \Delta\underset{\text{EQ}}{A}$. Conveniently, at eqtypes (although not in general), the *dummy* function computes zero changes:

Lemma 6.3. *If* $x : \underset{\text{EQ}}{A}$ *then* $\mathit{dummy}\ x ::_{\underset{\text{EQ}}{A}} x \rightsquigarrow x$.

For clarity, we write **0** rather than *dummy* when we use it to produce zero changes; we only call it *dummy* in dead code.

Finally, we come to $\delta($**for** $(x \in e)\ f)$, the computational heart of the seminaïve transformation, as **for** is what enables embedding relational algebra (the right-hand-sides of Datalog clauses) into Datafun. Here there are two things to consider, corresponding to the two **for**-clauses generated by $\delta($**for** $(x \in e)\ f)$. First, if the set ϕe we're looping over gains new elements $x \in \delta e$, we need to compute ϕf over these new elements. Second, if the inner loop ϕf changes, we need to add in its changes δf for every element, new or old, in the looped-over set, $\phi e \vee \delta e$. Just as in the ϕ-translation, we use **0**/*dummy* to calculate zero changes to set elements.

6.7 Leftovers

The ϕ rules we haven't yet discussed simply distribute ϕ over subexpressions. The remaining δ rules mostly do the same, with a few exceptions. In the case of $\delta(\{e_i\}_i) = \delta(e = f) = \bot$, the sub-expressions are discrete and cannot change, so we produce a zero change $\bot$. This is also the case for $\delta(\mathit{empty?}\ e) = \mathit{empty?}\ \phi e$, but as with $\delta(\mathit{split}\ e)$, the zero change here is at type $1 + 1$, so to get the tag right we use ϕe.

7 PROVING THE SEMINAÏVE TRANSFORMATION CORRECT

We have given two program transformations: ϕe, which optimizes e by computing fixed points seminaïvely; and δe, which finds the change in ϕe under a change in its free variables. To state the correctness of ϕ and δ, we need to show that ϕe preserves the meaning of e and that δe correctly updates ϕe with respect to changes in its variable bindings. Since our transformations modify the types of higher-order expressions to include the extra information needed for seminaïve evaluation, we cannot directly prove that the semantics is preserved. Instead, we formalize the relationship between e, ϕe, and δe using a logical relation, and use this relation to prove an *adequacy theorem* saying that the semantics is preserved for closed, first-order programs.

So, inductively on types A, letting $a, b \in [\![A]\!]$, $x, y \in [\![\Phi A]\!]$, and $dx \in [\![\Delta\Phi A]\!]$, we define a five place relation $dx ::_A x \lightning a \to y \lightning b$, meaning roughly "$x, y$ speed up a, b respectively, and dx changes x into y". The full definition is in figure 10.

At product, sum, and function types this is essentially a more elaborate version of the change structures given in §4.2. At set types, changes are still a set of values added to the initial value, but we additionally insist that the "slow" a, b and "speedy" x, y are equal. This is because we have engineered the definitions of Φ and ϕ to preserve behavior on equality types. Finally, since $\square A$ represents values which cannot change, dx is an uninformative empty tuple and the original and

$$
\begin{aligned}
() ::_{1} () \lightning () \to () \lightning () &\iff \top \\
\vec{dx} ::_{A_1 \times A_2} \vec{x} \lightning \vec{a} \to \vec{y} \lightning \vec{b} &\iff (\forall i)\ dx_i ::_{A_i} x_i \lightning a_i \to y_i \lightning b_i \\
\mathsf{in}_i\, dx ::_{A_1 + A_2} \mathsf{in}_j\, x \lightning \mathsf{in}_k\, a \to \mathsf{in}_l\, y \lightning \mathsf{in}_m\, b &\iff i = j = k = l = m \wedge dx ::_{A_i} x \lightning a \to y \lightning b \\
df ::_{A \to B} f_\phi \lightning f \to g_\phi \lightning g &\iff (\forall dx ::_A x \lightning a \to y \lightning b) \\
&\ \ df\ x\ dx ::_B f_\phi\ x \lightning f\ a \to g_\phi\ y \lightning g_\phi\ b \\
dx ::_{\{\underset{\text{EQ}}{A}\}} x \lightning a \to y \lightning b &\iff (x, y, x \cup dx) = (a, b, y) \\
() ::_{\Box A} (x, dx) \lightning a \to (y, dy) \lightning b &\iff (a, x, dx) = (b, y, dy) \wedge dx ::_A x \lightning a \to y \lightning b
\end{aligned}
$$

Fig. 10. Definition of the logical relation

updated values are identical. However, the "speedy" values are now *pairs* of a value and its zero change. This ensures that at a boxed function type, we will always have a derivative (a zero change) available.

The logical relation is defined on simple values, and so before we can state the fundamental theorem, we have to extend it to contexts Γ and substitutions, letting $\rho, \rho' \in [\![\Gamma]\!]$, $\gamma, \gamma' \in [\![\Phi\Gamma]\!]$, and $d\gamma \in [\![\Delta\Phi\Gamma]\!]$:

$$
\begin{aligned}
d\gamma ::_\Gamma \gamma \lightning \rho \to \gamma' \lightning \rho' \iff\ & (\forall x : A \in \Gamma)\ d\gamma_{dx} ::_A \gamma_x \lightning \rho_x \to \gamma'_x \lightning \rho'_x \\
& \wedge (\forall x :: A \in \Gamma)\ () ::_{\Box A} (\gamma_{dx}, \gamma_x) \lightning \rho_x \to (\gamma'_{dx}, \gamma'_x) \lightning \rho'_x
\end{aligned}
$$

With that in place, we can state the fundamental theorem, showing that ϕ and δ generate expressions which satisfy this logical relation:

Theorem 7.1 (fundamental property). *If* $\Gamma \vdash e : A$ *and* $d\gamma ::_\Gamma \gamma \lightning \rho \to \gamma' \lightning \rho'$ *then*

$$[\![\delta e]\!]\ (\gamma, d\gamma) ::_A [\![\phi e]\!]\ \gamma \lightning [\![e]\!]\ \rho \to [\![\phi e]\!]\ \gamma' \lightning [\![e]\!]\ \rho'$$

This theorem follows by a structural induction on typing derivations as usual, but a number of lemmas need to be proved in order to establish the fundamental theorem.

By and large, these lemmas generalize or build on results stated earlier in this paper regarding the simpler change structures from §4.2. For example, we build on lemmas 6.1 and 6.3 to characterize the logical relation at equality types $\underset{\text{EQ}}{A}$ and the behavior of *dummy*:

Lemma 7.2 (equality changes). *If* $dx ::_{\underset{\text{EQ}}{A}} x \lightning a \to y \lightning b$ *then* $x = a$ *and* $y = b$.

Lemma 7.3 (dummy is zero at eqtypes). *If* $x \in [\![\underset{\text{EQ}}{A}]\!]$ *then dummy* $x ::_{\underset{\text{EQ}}{A}} x \lightning x \to x \lightning x$.

Lemma 7.2 tells us that at equality types, the sped-up version of a value is the value itself. This is used later to prove our adequacy theorem. Lemma 7.3 is an analogue of lemma 6.3, showing that *dummy* function computes zero changes at equality types. This is used in the proof of the fundamental theorem for **for**-loops, in whose ϕ and δ translations $\mathbf{0}$ is implemented by *dummy*.

Next, we generalize lemma 4.1 to characterize changes at semilattice type:

Lemma 7.4 (semilattice changes). *At any semilattice type* L, *we have* $\Delta L = L$, *and moreover* $dx ::_L x \lightning a \to y \lightning b$ *iff* $x = a$ *and* $y = b = x \vee_L dx$

This follows by induction on semilattice types L, and from lemma 7.2 (noting that every semilattice type is an equality type). We require this lemma in the proofs of the fundamental theorem in all the cases involving semilattice types – namely $\perp$, $\vee$, **for**-loops, and *fix*.

Since typing rules that involve discreteness (such as the $\Box$ rules) manipulate the context, we need some lemmas regarding these manipulations. First, we show that all valid changes for a context with only discrete variables send substitutions to themselves, recalling that $\lceil\Gamma\rceil$ contains only the discrete variables from Γ.

Lemma 7.5 (discrete contexts don't change). *If* $() ::_{\lceil\Gamma\rceil} \gamma \lightning \rho \to \gamma' \lightning \rho'$ *then* $\gamma = \gamma'$ *and* $\rho = \rho'$.

We use this lemma in combination with the next, which says that any valid context change gives rise to a valid change on a stripped context:

Lemma 7.6 (context stripping). *If* $\mathrm{d}\gamma ::_\Gamma \gamma \lightning \rho \to \gamma' \lightning \rho'$ *then*

$$() ::_{\lceil\Gamma\rceil} \mathit{strip}_{\Phi\Gamma}(\gamma) \lightning \mathit{strip}_{\Gamma}(\rho) \to \mathit{strip}_{\Phi\Gamma}(\gamma') \lightning \mathit{strip}_{\Gamma}(\rho')$$

where $\mathit{strip}_\Gamma = \langle \pi_x \rangle_{x::A\in\Gamma}$ *keeps only the discrete variables from a substitution.*

Jointly, these two lemmas ensure that a valid change to any context is an identity on the discrete part. We use these in all the cases of the fundamental theorem involving discrete expressions – equality $e_1 = e_2$, set literals $\{e_i\}_i$, emptiness tests $\mathit{empty?}\ e$, and box introduction $[e]$.

Once the fundamental theorem has been established, we can specialize it to closed terms and equality types, Then, the equality changes lemma implies adequacy – that first-order closed programs compute the same result when ϕ-translated:

Theorem 7.7 (adequacy). *If* $\varepsilon \vdash e : \underset{\text{EQ}}{A}$ *then* $[\![e]\!] = [\![\phi e]\!]$.

8 APPLYING THE SEMINAÏVE TRANSFORMATION TO TRANSITIVE CLOSURE

Let's try applying the seminaïve transform to a simple Datafun program: the transitive closure function *trans* from §3.1:

$$\begin{aligned}
&\mathit{trans}\ [e] = \textbf{fix}\ \mathrm{p}\ \textbf{is}\ e \cup (e \bullet \mathrm{p})\\
&\mathrm{s} \bullet \mathrm{t} = \textbf{for}\ ((x, y_1) \in \mathrm{s})\ \textbf{for}\ ((y_2, z) \in \mathrm{t})\ \textbf{when}\ (y_1 = y_2)\ \{(x, z)\}
\end{aligned}$$

In the process we'll discover that besides ϕ itself we need a few simple optimisations to actually speed up our program: most importantly, we need to propagate $\bot$ expressions.

In our experience, performing ϕ and δ by hand is easiest when you work inside-out. At the core of transitive closure is a relation composition, $(e \bullet \mathrm{p})$, and at the core of relation composition is a **when**-expression. Let's take a look at its ϕ and δ translations:

$$\begin{aligned}
\phi(\textbf{when}\ (y_1 = y_2)\ \{(x,z)\}) &= \phi(\textbf{for}\ (() \in y_1 = y_2)\ \{(x,z)\}) && \text{desugar } \textbf{when}\\
&= \textbf{for}\ (() \in y_1 = y_2)\ \phi\{(x,z)\} && \text{apply } \phi\text{, omitting an unused } \textbf{let}\\
&= \textbf{when}\ (y_1 = y_2)\ \{(x,z)\} && \text{resugar}
\end{aligned}$$

Frequently, as in this case, ϕ does nothing interesting. For brevity we'll skip such no-op translations.

$$\begin{aligned}
&\delta(\textbf{when}\ (y_1 = y_2)\ \{(x,z)\})\\
&= \delta(\textbf{for}\ (() \in y_1 = y_2)\ \{(x,z)\}) && \text{desugar } \textbf{when}\\
&= \quad \textbf{for}\ (() \in \delta(y_1 = y_2))\ \phi\{(x,z)\}\\
&\quad \cup\ \textbf{for}\ (() \in \phi(y_1 = y_2) \cup \delta(y_1 = y_2))\ \delta\{(x,z)\} && \text{apply } \phi\text{, omitting unused } \textbf{let}\text{s}\\
&= \textbf{for}\ (() \in \bot)\ \{(x,z)\} \cup \textbf{for}\ (() \in \phi(y_1 = y_2) \cup \bot)\ \bot && \text{apply } \phi(y_1 = y_2) \text{ and } \delta\{(x,z)\}\\
&= \bot && \text{propagate } \bot
\end{aligned}$$

The core insight here is that neither $y_1 = y_2$ nor $\{(x,z)\}$ can change. Propagating this information – for example, rewriting $(\textbf{for}\ (\ldots)\ \bot)$ to $\bot$ – can simplify derivatives and eliminate expensive **for**-loops.

Now let's pull out and examine **for** $((y_2, z) \in \mathrm{t})$ **when** $(y_1 = y_2)$ $\{(x, z)\}$. The ϕ translation is again a no-op.

$$\begin{aligned}
&\delta(\textbf{for}\ ((y_2, z) \in \mathrm{t})\ \textbf{when}\ (y_1 = y_2)\ \{(x, z)\}) \\
=\ &\ \textbf{for}\ ((y_2, z) \in \mathrm{dt})\ \phi(\textbf{when}\ (y_1 = y_2)\ \{(x, z)\}) && \text{apply } \delta\text{, omitting some unused } \textbf{let}\text{s} \\
&\cup\ \textbf{for}\ ((y_2, z) \in \mathrm{t} \cup \mathrm{dt})\ \delta(\textbf{when}\ (y_1 = y_2)\ \{(x, z)\}) \\
=\ &\textbf{for}\ ((y_2, z) \in \mathrm{dt})\ \textbf{when}\ (y_1 = y_2)\ \{(x, z)\} && \text{applying prior work, propagating } \bot
\end{aligned}$$

Tackling the outermost **for** loop:

$$\begin{aligned}
&\delta(\textbf{for}\ ((x, y_1) \in \mathrm{s})\ \textbf{for}\ ((y_2, z) \in \mathrm{t})\ \textbf{when}\ (y_1 = y_2)\ \{(x, z)\}) \\
=\ &\textbf{for}\ ((x, y_1) \in \mathrm{ds})\ \phi(\textbf{for}\ ((y_2, z) \in \mathrm{t})\ \textbf{when}\ (y_1 = y_2)\ \{(x, z)\}) && \text{definition of } \delta(\textbf{for} \ldots) \\
\cup\ &\textbf{for}\ ((x, y_1) \in \mathrm{s} \cup \mathrm{ds})\ \delta(\textbf{for}\ ((y_2, z) \in \mathrm{t})\ \textbf{when}\ (y_1 = y_2)\ \{(x, z)\}) \\
=\ &\textbf{for}\ ((x, y_1) \in \mathrm{ds})\ \textbf{for}\ ((y_2, z) \in \mathrm{t})\ \textbf{when}\ (y_1 = y_2)\ \{(x, z)\} && \text{applying prior work} \\
\cup\ &\textbf{for}\ ((x, y_1) \in \mathrm{s} \cup \mathrm{ds})\ \textbf{for}\ ((y_2, z) \in \mathrm{dt})\ \textbf{when}\ (y_1 = y_2)\ \{(x, z)\} \\
=\ &(\mathrm{ds} \bullet \mathrm{t}) \cup ((\mathrm{s} \cup \mathrm{ds}) \bullet \mathrm{dt}) && \text{rewriting in terms of } \bullet
\end{aligned}$$

This, then, is the derivative $\delta(\mathrm{s} \bullet \mathrm{t})$ of relation composition. With a bit of rewriting, this is equivalent to $(\mathrm{ds} \bullet \mathrm{t}) \cup (\mathrm{s} \bullet \mathrm{dt}) \cup (\mathrm{ds} \bullet \mathrm{dt})$, which is perhaps the derivative a human would give.

Let's use this to figure out $\phi(\mathit{trans}\ [e])$. Working inside out, we start with the derivative of the loop body, $\delta(e \cup (e \bullet \mathrm{p}))$:

$$\begin{aligned}
\delta(e \cup (e \bullet \mathrm{p})) &= \delta e \cup \delta(e \bullet \mathrm{p}) \\
&= \delta e \cup (\delta e \bullet \mathrm{p}) \cup ((e \cup \delta e) \bullet \mathrm{dp}) \\
&= \bot \cup (\bot \bullet \mathrm{p}) \cup ((e \cup \bot) \bullet \mathrm{dp}) && \delta e \text{ is a zero change; insert } \bot \\
&= e \bullet \mathrm{dp} && \text{propagate } \bot
\end{aligned}$$

The penultimate step requires a new optimization. By definition $\delta e = de$, but since e is discrete we know de is a zero change, so we may safely replace it by $\bot$.

Putting everything together, we have:

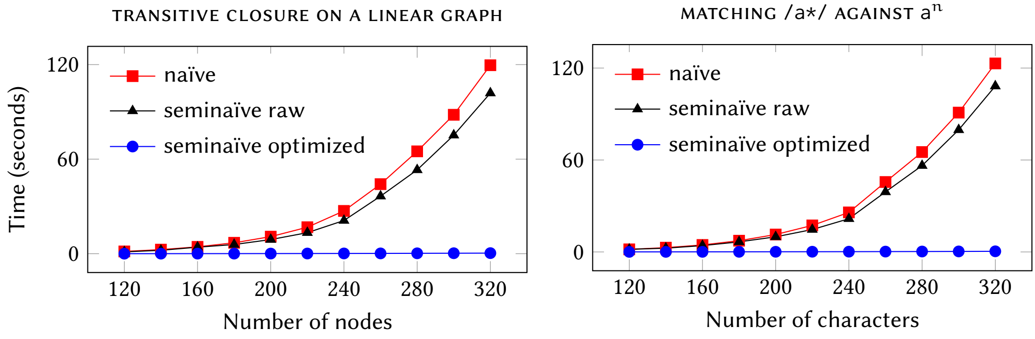
$$\begin{aligned}
\phi(\textbf{fix}\ \mathrm{p}\ \textbf{is}\ e \cup (e \bullet \mathrm{p})) &= \phi(\mathit{fix}\ [\lambda \mathrm{p}.\ e \cup (e \bullet \mathrm{p})]) && \text{desugaring} \\
&= \mathit{semifix}\ \phi[\lambda \mathrm{p}.\ e \cup (e \bullet \mathrm{p})] \\
&= \mathit{semifix}\ [(\phi(\lambda \mathrm{p}.\ e \cup (e \bullet \mathrm{p})),\ \delta(\lambda \mathrm{p}.\ e \cup (e \bullet \mathrm{p})))] \\
&= \mathit{semifix}\ [((\lambda \mathrm{p}.\ e \cup (e \bullet \mathrm{p})),\ (\lambda[p].\ \lambda \mathrm{dp}.\ e \bullet \mathrm{dp}))] && \text{previous work}
\end{aligned}$$

Examining the recurrence produced by this use of *semifix*, we recover the seminaïve transitive closure algorithm from §4.1:

$$\begin{aligned}
\mathrm{x}_0 &= \bot & \mathrm{x}_{i+1} &= \mathrm{x}_i \cup \mathrm{dx}_i \\
\mathrm{dx}_0 &= (\lambda \mathrm{p}.\ e \cup (e \bullet \mathrm{p}))\ \bot = e & \mathrm{dx}_{i+1} &= (\lambda[p].\ \lambda \mathrm{dx}.\ e \bullet \mathrm{dp})\ [\mathrm{x}_i]\ \mathrm{dx}_i = e \bullet \mathrm{dx}_i
\end{aligned}$$

9 IMPLEMENTATION AND OPTIMIZATION

We have implemented a compiler from a fragment of Datafun (omitting sum types) to Haskell, available at https://github.com/rntz/datafun/tree/popl20/v4-fastfix. We use Haskell's `Data.Set` to represent Datafun sets, and typeclasses to implement Datafun's notions of equality and semilattice types. We do no query planning and implement no join algorithms; relational joins, written in Datafun as nested **for**-loops, are compiled into nested loops. Consequently our performance is



	GRAPH SIZE / STRING LENGTH										
	120	140	160	180	200	220	240	260	280	300	320
REGEX SEARCH, NAÏVE	1.683	2.786	4.549	7.324	11.357	17.304	25.788	45.634	65.174	90.934	123.023
TRANSITIVE CLOSURE, NAÏVE	1.446	2.599	4.356	6.933	10.840	16.803	27.159	44.136	64.953	88.154	119.604
REGEX SEARCH, SEMINAÏVE RAW	1.687	2.454	4.134	6.573	9.854	14.611	21.661	39.171	56.345	79.687	108.236
TRANSITIVE CLOSURE, SEMINAÏVE RAW	1.187	2.163	4.154	5.835	8.982	13.350	21.069	36.512	53.197	75.209	101.933
REGEX SEARCH, SEMINAÏVE OPTIMIZED	0.028	0.037	0.054	0.075	0.102	0.133	0.171	0.220	0.269	0.331	0.401
TRANSITIVE CLOSURE, SEMINAÏVE OPTIMIZED	0.026	0.037	0.056	0.072	0.099	0.130	0.170	0.204	0.259	0.312	0.377

Fig. 11. Naïve vs seminaïve evaluation of transitive closure and regex matching in Datafun

worse than any real Datalog engine. However, we do implement the ϕ translation, along with the following optimizations:

(1) Propagating $\bot$; for example, rewriting $(e \vee \bot) \rightsquigarrow e$ and $(\textbf{for}\ (x \in e)\ \bot) \rightsquigarrow \bot$.
(2) Inserting $\bot$ in place of semilattice-valued zero changes (for example, changes to discrete variables δx). This makes $\bot$-propagation more effective.
(3) Recognising complex zero change expressions; for example, $\delta e\ \boxed{\phi f}\ \delta f$ is a zero change if δe and δf are. This allows more zero changes to be replaced by $\bot$, especially in higher-order code such as our regular expression example.

To test whether the ϕ translation can produce the asymptotic performance gains we claim, we benchmark two example Datafun programs:

(1) Finding the transitive closure of a linear graph using the *trans* function from §3.1. We chose this example because, as discussed in §4, it has a well understood asymptotic speed-up under seminaïve evaluation. This means that if we've failed to capture the essence of seminaïve evaluation, it should be highly visible.
(2) Finding all matches of the regular expression /a*/ in the string a^n, using the regex combinators from §3.2. Finding all matches for /a*/ amounts to finding the reflexive, transitive closure of the matches of /a/, and on a^n these form a linear graph. Thus it is a close computational analogue of our first example, written in a higher-order style. We chose this example to test whether our extension of seminaïve evaluation properly handles Datafun's distinctive feature: higher-order programming.

We compiled each program in three distinct ways: *naïve*, without the ϕ transform (but with $\bot$-propagation); *seminaïve raw*, with the ϕ transform but without further optimization; and *seminaïve optimized*, with the ϕ transform followed by all three optimizations listed previously. The results are shown in figure 11. The measured times are substantially similar for transitive closure and regex search across all three optimization levels, suggesting that higher-order code does not pose a particular problem for our optimizations. However, compared to *naïve*, the ϕ transform alone (*seminaïve raw*) provides only a small speed-up, roughly 10–20%. Only when followed by other optimizations (*seminaïve optimized*) does it provide the expected asymptotic speedup.[9]

We believe this is because both $\phi(\textbf{for}\ (x \in e)\ ...)$ and $\delta(\textbf{for}\ (x \in e)\ ...)$ produce loops that iterate over at least every $x \in \phi e$. Consulting our logical relation at set type, we see that in this case e and ϕe will be identical, and so the number of iterations never shrinks. However, as demonstrated in §8, if the body can be simplified to $\bot$, then we can eliminate the loop entirely by rewriting $(\textbf{for}\ (x \in e)\ \bot)$ to $\bot$, which allows for asymptotic improvement.

As in Datalog, we do not expect seminaïve evaluation to be useful on *all* recursive programs. Under naïve evaluation, each iteration towards a fixed point is more expensive than the last, so as a rule of thumb, seminaïve evaluation is more valuable the more iterations required.

10 DISCUSSION AND RELATED WORK

Nested fixed points. The typing rule for *fix* e requires $e : \square(L_{\text{FIX}} \to L_{\text{FIX}})$. The ϕ translation takes advantage of this $\square$, decorating expressions of type $\square A$ with their zero changes. However, it also prevents an otherwise valid idiom: in a nested fixed-point expression $\textbf{fix}\ x\ \textbf{is}\ \ldots(\textbf{fix}\ y\ \textbf{is}\ e)\ldots$, the inner fixed point body e cannot use the monotone variable x! This restriction is not present in Arntzenius and Krishnaswami [2016]; its addition brings Datafun closer to Datalog, whose syntax cannot express this sort of nested fixed point.

We suspect it is possible to lift this restriction without losing seminaïve evaluation, by decorating *all* expressions and variables (not just discrete ones) with zero changes. However, this also invalidates $\delta(\mathit{fix}\ f) = \bot$: now that f can change, so can *fix* f. Luckily, there is a simple and correct solution: $\delta(\mathit{fix}\ f) = \mathit{fix}\ [\delta f\ [\mathit{fix}\ f]]$ [Arntzenius 2017]. However, to compute this new fixed point seminaïvely, we need a *second derivative*: the zero change to $\delta f\ [\mathit{fix}\ f]$. Indeed, for a program with fixed points nested n deep, we need n^{th} derivatives. We leave this to future work.

Self-maintainability. In the incremental λ-calculus, a function f is *self-maintainable* if its derivative f' depends only upon the change dx to the argument and not upon the base point x. This is a crucial property, because it lets us compute the change in the function's result without recomputing the original input, which might be expensive. So it's reasonable to ask whether lack of self-maintainability is ever an issue in Datafun. We suspect (without proof) that due to the limited way seminaïve evaluation uses incremental computation, it usually isn't. For example, consider a variant definition of transitive closure as the fixed point of $f = \lambda \mathit{path}.\ \mathit{edge} \cup (\mathit{path} \bullet \mathit{path})$. This is not self-maintainable; its derivative is:

$$f'\ \mathit{path}\ \mathit{dpath} = (\mathit{path} \bullet \mathit{dpath}) \vee (\mathit{dpath} \bullet \mathit{path}) \vee (\mathit{dpath} \bullet \mathit{dpath})$$

However, this is not a problem when computing its fixed point seminaïvely, because both *path* and *dpath* are available from the previous iteration. Thus non-self-maintainable fixed points do not appear to be forced into doing extensive recomputation.

Related work. The incremental lambda calculus was introduced by Cai et al. [2014], as a static program transformation which associated a type of *changes* to each base type, along with operations to update a value based on a change. Then, a program transformation on the simply-typed lambda

[9] We also tried following the ϕ transform with only $\bot$-propagation, dropping the other two optimisations. This produced essentially the same results as *seminaïve optimized*, so we have omitted it from figure 11. It is unclear whether inserting $\bot$ or recognizing complex zero changes are ever necessary to achieve an asymptotic speed-up.

It's also worth addressing the asymptotic performance of *seminaïve optimized*. On regex search, for example, doubling the string length from 160 to 320 produces a slowdown factor of $\frac{.401}{.054} \approx 7.42$! However, since there are quadratically many matches and we find all of them, the best possible runtime is $O(n^2)$. Moreover, our nested-loop joins are roughly a factor of n slower than optimal, so we expect $O(n^3)$ behavior or worse. This back-of-the-envelope estimation predicts a slowdown of $2^3 = 8$, reasonably close to 7.42. Phew!

calculus with base types and functions was defined, which rewrote lambda terms into incremental functions which propagated changes as needed to reduce recomputation. The fundamental idea of the incremental function type taking two arguments (a base point and a change) is one we have built on, though we have extended the transformation to support many more types like sums, sets, modalities, and fixed points. Subsequently, Giarrusso et al. [2019] extended this work to support the *untyped* lambda calculus, additionally also extending the incremental transform to support additional *caching*. In this work, the overall correctness of change propagation was proven using a step-indexed logical relation, which defined which changes were valid in a fashion very similar to our own.

The motivating example of this line of work was to optimize bulk collection operations. However, all of the intuitions were phrased in terms of calculus – a change structure can be thought of as a space paired with its tangent space, a zero change on functions is a derivative, and so on. However, the idea of a derivative as a linear approximation is taken most seriously in the work on the differential lambda calculus [Ehrhard and Regnier 2003]. These calculi have the beautiful property that the *syntactic* linearity in the lambda calculus corresponds to the *semantic* notion of linear transformation.

Unfortunately, the intuition of a derivative has its limits. A function's derivative is *unique*, a property which models of differential lambda calculi have gone to considerable length to enforce [Blute et al. 2006]. This is problematic from the point of view of seminaïve evaluation, since we make use of the freedom to overapproximate. In §6.6, we followed common practice from Datalog and took the derivative $\delta(e \vee f)$ to be $\delta(e) \vee \delta(f)$, which may overapproximate the change to $e \vee f$. This spares us from having to do certain recomputations to construct set differences; it is not clear to what extent seminaïve evaluation's practical utility depends on this approximation.

Alvarez-Picallo et al. [2019] offer an alternative formulation of change structures, by requiring changes to form a monoid, and representing the change itself with a monoid action. They use change actions to prove the correctness of seminaïve evaluation for Datalog, and express the hope that it could apply to Datafun. Unfortunately, it does not seem to – the natural notion of function change in their setting is pointwise, which does not seem to lead to the derivatives we want in the examples we considered.

Overall, there seems to be a lot of freedom in the design space for incremental calculi, and the tradeoffs different choices are making remain unclear. Much further investigation is warranted!

REFERENCES

Foto Afrati and Christos H. Papadimitriou. 1993. The Parallel Complexity of Simple Logic Programs. *J. ACM* 40, 4 (Sep 1993), 891–916. https://doi.org/10.1145/153724.153752

Natasha Alechina, Michael Mendler, Valeria de Paiva, and Eike Ritter. 2001. Categorical and Kripke Semantics for Constructive S4 Modal Logic. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings (Lecture Notes in Computer Science)*, Laurent Fribourg (Ed.), Vol. 2142. Springer, 292–307. https://doi.org/10.1007/3-540-44802-0_21

Mario Alvarez-Picallo, Alex Eyers-Taylor, Michael Peyton Jones, and C.-H. Luke Ong. 2019. Fixing Incremental Computation - Derivatives of Fixpoints, and the Recursive Semantics of Datalog. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science)*, Luís Caires (Ed.), Vol. 11423. Springer, 525–552. https://doi.org/10.1007/978-3-030-17184-1_19

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. 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.* 249–260.

Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. 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.* 1371–1382.

Michael Arntzenius. 2017. Static differentiation of monotone fixed points. http://www.rntz.net/files/fixderiv.pdf. Unpublished note.

Michael Arntzenius and Neelakantan R. Krishnaswami. 2016. Datafun: A Functional Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 214–227. https://doi.org/10.1145/2951913.2951948

François Bancilhon. 1986. Naive Evaluation of Recursively Defined Relations. In *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*, Michael L Brodie and John Mylopoulos (Eds.). Springer-Verlag New York, Inc., New York, NY, USA, 165–178. http://dl.acm.org/citation.cfm?id=8789.8804

Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1986. 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)*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/6012.15399

Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. 2010. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security* 18, 4 (2010), 619–665.

R. F. Blute, J. R. B. Cockett, and R. A. G. Seely. 2006. Differential categories. *Mathematical Structures in Computer Science* 16, 6 (2006), 1049–1083. https://doi.org/10.1017/S0960129506005676

Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. 2014. A Theory of Changes for Higher-order Languages: Incrementalizing λ-calculi by Static Differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 145–155. https://doi.org/10.1145/2594291.2594304

Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. Knowl. Data Eng.* 1, 1 (1989), 146–166.

Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. 2001. Complexity and Expressive Power of Logic Programming. *Comput. Surveys* 33, 3 (Sep 2001), 374–425. https://doi.org/10.1145/502807.502810

Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. 2007. .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*. 78–133.

Thomas Ehrhard and Laurent Regnier. 2003. The differential lambda-calculus. *Theoretical Computer Science* 309, 1 (2003), 1 – 41. https://doi.org/10.1016/S0304-3975(03)00392-X

George Fourtounis and Yannis Smaragdakis. 2019. Deep Static Modeling of invokedynamic. In *ECOOP (LIPIcs)*, Vol. 109. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Paolo G. Giarrusso, Yann Régis-Gianas, and Philipp Schuster. 2019. Incremental \lambda -Calculus in Cache-Transfer Style - Static Memoization by Program Transformation. In *ESOP (Lecture Notes in Computer Science)*, Vol. 11423. Springer, 553–580.

Rich Hickey, Stuart Halloway, and Justin Gehtland. 2012. Datomic: The fully transactional, cloud-ready, distributed database. http://www.datomic.com. Accessed: 5 July 2019.

Martin Hofmann. 1997. A Mixed Modal/Linear Lambda Calculus with Applications to Bellantoni-Cook Safe Recursion. In *CSL (Lecture Notes in Computer Science)*, Vol. 1414. Springer, 275–294.

Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On Synthesis of Program Analyzers. In *CAV (2) (Lecture Notes in Computer Science)*, Vol. 9780. Springer, 422–430.

Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 194–208. https://doi.org/10.1145/2908080.2908096

Frank Pfenning and Rowan Davies. 2001. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science* 11, 4 (2001), 511–540. https://doi.org/10.1017/S0960129501003322

Max Schäfer and Oege de Moor. 2010. 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*. 145–156.

Yannis Smaragdakis and George Balatsouras. 2015. *Pointer Analysis*. Now Foundations and Trends. https://ieeexplore.ieee.org/document/8186778

Philip Wadler. 1992. Comprehending Monads. *Mathematical Structures in Computer Science* 2, 4 (1992), 461–493.

John Whaley. 2007. *Context-Sensitive Pointer Analysis using Binary Decision Diagrams*. Ph.D. Dissertation. Stanford University.

John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, William Pugh and Craig Chambers (Eds.). ACM, 131–144. https://doi.org/10.1145/996841.996859