# Seminaïve Evaluation for a Higher-Order Functional Language

Anonymous Author(s)

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 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.

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

## 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 *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, there are two flies in the ointment. First, 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

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

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 the Flix language [Madsen et al. 2016] and the Datafun language [Arntzenius and Krishnaswami 2016]. Conveniently for our exposition, these two languages embody two alternative design strategies.

Flix adopts the route of treating Datalog as an embedded domain-specific language [Leijen and Meijer 1999]. That is, Flix is a fairly conventional (albeit well-designed) functional programming language roughly comparable to ML or Haskell, extended with types representing Datalog predicates and programs. The evaluation of a Flix program builds a Datalog program, which is then handed off to a custom Datalog engine (albeit extended to support arbitrary semilattices). This stratification means that (a) standard Datalog implementation techniques can be used mostly off-the-shelf, but that (b) its functional programming side is mostly a very powerful macro system for Datalog. The only way Flix code runs at Datalog execution time is via the definition of new semilattices (which is functional Flix code implementing a semilattice interface), and for this Flix offers program-verification style correctness checking (including SMT-based tooling).

Like Flix, Datafun is a functional programming language, but unlike Flix, its type discipline supports tracking *monotonicity* of 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 Datalog's stratification condition. Tracking monotonicity permits a much tighter integration between the functional and logic programming 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.

Second, actually 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 *semi-naïve 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 $\emptyset, f(\emptyset), f^2(\emptyset), \ldots$ until $f^i(\emptyset) = f^{i+1}(\emptyset)$. 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 transformation we present.

- We define a program transformation to *incrementalize* well-typed Datafun programs. The translation is a compositional type-and-syntax-directed transformation, and works uniformly at all types including function types. 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, comonads, and (well-founded) fixed points.
- We establish the correctness of our transformation using a novel logical relation which simultaneously defines the changes connecting old and updated programs, as well as the optimized version of both. The fundamental lemma shows that our transformation is semantics-preserving: any closed program of first-order type has the same meaning after optimization.
- We then 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 program optimizations we found helpful for putting the optimization into practice.

## 2 Datalog and Datafun by Example

### 2.1 Datalog

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

```
parent(earendil, elrond).
parent(elrond, arwen).
parent(arwen, eldarion).

ancestor(X, Z) ← parent(X, Z).
ancestor(X, Z) ← ancestor(X, Y), parent(Y, Z).
```

This defines two binary relations, `parent` and `ancestor`. Lowercase terms like `elrond` and `arwen` are symbols *a la* Lisp, and uppercase terms like `X` and `Y` are variables.

The `parent` relation declares a set of ground *facts*; we assert that `earendil` is the parent of `elrond`, `elrond` the parent of `arwen`, and so on. The `ancestor` relation is declared as a pair of *rules*, the first saying X is an ancestor of Z if X is Z's parent, and the second saying X is an ancestor of Z if there is a intermediate person Y such that X is Y's ancestor and Y is Z's parent.

Semantically, a predicate denotes the set of tuples that satisfy it. Recursive definitions can be interpreted by viewing the whole program as a relation transformer, and taking the least fixed point of that function. Datalog imposes syntactic restrictions which ensure the relation transformer the rules define is monotone, guaranteeing a unique least fixed point.

### 2.2 Datafun

The idea behind Datafun is that since the semantics of a Datalog program is a monotone set-valued operator, its natural home is the category **Poset** of partial orders and monotone functions. Since **Poset** is bicartesian closed, it can interpret the simply-typed λ-calculus, which gives 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.

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

FIGURE 1. Datafun syntax

Datafun begins as the simply-typed λ-calculus with functions ($\lambda x.\, e$ and $e\ f$), sums ($\mathrm{in}_i\ e$ and **case** $e$ **of** ...), and products ($\langle e, f\rangle$ and $\pi_i\ e$). To this, we add a type of finite sets[1] $\{\underset{\text{eq}}{A}\}$, introduced with set literals $\{e_0, \ldots e_n\}$, and eliminated using Moggi's monadic bind syntax, **for** $(\mathit{x} \in e_1)\ e_2$, signifying the union over all $\mathit{x} \in e_1$ of $e_2$.

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

How can we express non-monotone operations while preserving the property that all functions are monotone? We square this circle by introducing the *discreteness* type constructor $\Box A$. The elements of $\Box A$ are *exactly the same* as the elements of $A$, but the order on $\Box A$ is discrete, $x \leqslant y : \Box A$ iff $x = y$. Monotonicity of a function $\Box A \to B$ is vacuous: $x = y$ always implies $f(x) \leqslant f(y)$! This lets us encode ordinary, possibly non-monotone, functions $A \to B$ as monotone functions $\Box A \to B$. Semantically, $\Box$ is a comonad, and accordingly the syntax we use for this is a variant of Pfenning and Davies [2001]'s syntax for constructive S4 modal logic. We make discrete terms with the introduction form $[e]$ and eliminate them with a pattern matching eliminator **let** $[\mathit{x}] = e$ **in** $f$. Discrete variables are colored and italicised $\mathit{x}$, while monotone variables are uncolored and upright $x$. Colored terms $e$ are restricted to only refer to discrete variables. So in the equality test $e = f$, the terms $e$ and $f$ must be discrete.

Finally, Datafun includes fixed points, *fix* $e$. The *fix* combinator takes a function $\Box(\underset{\text{fix}}{L} \to \underset{\text{fix}}{L})$ and returns its least fixed point. We impose two restrictions on the fixed point operator to ensure well-definedness and termination. First, we require that recursion occur at *semilattice types*, $L$. A join-semilattice is a partial order with a least element $\bot$ and a least-upper-bound ("join") operation $\vee$. 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 lattice has no infinite ascending chains, recursion from the bottom element is guaranteed to find the least fixed point.

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 §9. Note

[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.

$$\mathbf{bool} \xrightarrow{\text{rewrite}} \{1\} \qquad \{e \mid \varepsilon\} \xrightarrow{\text{rewrite}} \{e\}$$

$$\mathit{false} \xrightarrow{\text{rewrite}} \{\} \qquad \{e \mid p \in f, \ldots\} \xrightarrow{\text{rewrite}} \mathbf{for}\ (p \in f)\ \{e \mid \ldots\}$$

$$\mathit{true} \xrightarrow{\text{rewrite}} \{\langle\rangle\} \qquad \{e \mid f, \ldots\} \xrightarrow{\text{rewrite}} \mathbf{when}\ (f)\ \{e \mid \ldots\}$$

$$\mathbf{when}\ (e)\ f \xrightarrow{\text{rewrite}} \mathbf{for}\ (\langle\rangle \in e)\ f$$

FIGURE 2. Syntactic sugar

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.

### 2.3 Examples of Datafun Programs

In the examples, we make free use of strings and integers, ordered discretely ($x \leqslant y$ iff $x = y$). We also make use of some syntax sugar in the definitions:

(1) First, 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. Namely, the box-elimination form $\mathbf{let}\ [x] = e\ \mathbf{in}\ e'$ 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.

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

(3) Finally, we make use of set comprehension notation. The desugaring we use is based on the translation of comprehensions to the monadic operators [Wadler 1992].

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

*2.3.1 Set Operations.* Even before higher-order functions, one of the main benefits of Datafun relative to Datalog is that it offers the ability to manipulate relations as first class values. In this subsection we will show how a variety of standard operations on sets can be represented in Datafun.

*Membership Tests.* The first operation we consider is the membership test operation.

$$\mathit{mem} : \Box A \to \{A\} \to \mathbf{bool}$$
$$\mathit{mem}\ [x]\ ys = \mathbf{for}\ (y \in ys)\ x = y$$

This checks if the input $x$ is equal to any $y \in ys$. The argument $x$ to *mem* is discrete, because an element is in a set or not – the test is not monotone in $x$.

*Set Intersection.* Using *mem*, it is possible to define set intersection, by taking the union of all the singleton sets $\{x\}$ sets where $x$ is an element of $xs$ also in the set $ys$.

$$\_ \wedge \_ : \{A\} \to \{A\} \to \{A\}$$
$$xs \wedge ys = \mathbf{for}\ (x \in xs)\ \mathbf{when}\ (\mathit{mem}\ [x]\ ys)\ \{x\}$$

Using comprehensions, this could alternately be written as:

$$xs \wedge ys = \{x \mid x \in xs,\ \mathit{mem}\ [x]\ ys\}$$

*Relational Composition.* We can also define the composition of two relations in Datafun.

$$\begin{aligned} \_ \bullet \_ &: \{A \times B\} \to \{B \times C\} \to \{A \times C\} \\ xs \bullet ys &= \textbf{for}\ (\langle a, b\rangle \in xs)\ \textbf{for}\ (\langle b', c\rangle \in ys)\ \textbf{when}\ (b = b')\ \{\langle a, c\rangle\} \end{aligned}$$

This is basically a transcription of the mathematical definition, where we build those pairs which agree on their B-typed components. It can also be written using set comprehension as:

$$\begin{aligned} \_ \bullet \_ &: \{A \times B\} \to \{B \times C\} \to \{A \times C\} \\ xs \bullet ys &= \{\langle a, c\rangle \mid \langle a, b\rangle \in xs, \langle b', c\rangle \in ys, b = b'\} \end{aligned}$$

*Transitive Closure.* Now, we define the transitive closure of a relation.

$$\begin{aligned} tc &: \square\{A \times A\} \to \{A \times A\} \\ tc\ [e] &= \textbf{fix}\ s\ \textbf{is}\ e \vee (e \bullet s) \end{aligned}$$

This definition uses recursion, just like the mathematical definition – a transitive closure is the least relation containing the original relation *xs* and the composition of *xs* with the transitive closure. However, one feature of this definition peculiar to core Datafun is that the argument type is $\square\{A \times A\}$ – 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.

2.3.2 *Combinators for Regular Expression Matching.* 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 parsing 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.

We assume the existence of a function $pos : \textbf{string} \to \{\textbf{int}\}$ which takes a string and returns the set of valid indices in that string, and assume that string indexing is written $s[n]$, as in Java or C. Having done this, we define the type of regular expression matchers.

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

A regular expression takes a string (boxed so that it can be used inside fixed point expressions), and returns a set of pairs of integers. The idea is that if the regular expression matcher is passed the string argument $s$, then if $(i, n)$ is one of the returned values, the substring $s_i, s_{i+1}, \ldots, s_{n-1}$ is matched by the regular expression. That is, it is inclusive on the left and exclusive on the right.

$$\begin{aligned} &sym : \textbf{char} \to \textbf{re} \\ &sym\ c\ [s] = \textbf{for}\ (n \in pos\ s)\ \textbf{when}\ (s[n] = c)\ \{(n, n+1)\} \end{aligned}$$

The *sym* combinator takes a character and returns a set of substrings by returning the set $(n, n+1)$ where the $n$-th element of the string $s$ is the character $c$.

$$\begin{aligned} &nil : \textbf{re} \\ &nil\ [s] = \textbf{for}\ (n \in pos\ s)\ \{(n, n)\} \end{aligned}$$

The call *nil* $[s]$ yields $(n, n)$ for each position $n$ in $s$, since an empty substring can start anywhere.

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

The *seq* combinator takes two regular expressions $r_1$ and $r_2$ as arguments, applies its argument to both, and takes the relational composition of the results. Therefore, if the range $(i, j)$ was in the result of $r_1$, and $(j, k)$ was in the result of $r_2$, then we will return $(i, k)$, just as desired.

$$
\begin{aligned}
&\mathit{bot} : \mathbf{re} \\
&\mathit{bot}\ \_ = \bot
\end{aligned}
$$

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

We get the empty set of matches from *bot*, and *alt* unions the matches of its two arguments.

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

The most interesting regular expression combinator is the Kleene star operation *star*. It uses *nil* to get the reflexive relation on positions, and then takes the transitive closure of the regular expression it received as an argument using the *tc* operation. This forces its argument to be boxed, since *tc* calculates a fixed point, and only discrete variables can occur inside of fixed point expressions.[2]

2.3.3 *Combinators for Regular Expression Matching, 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.

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

Our new type of combinators takes a string and a starting position, and returns a set of ending positions. In contrast, the earlier type took a string and returned a set of start/end pairs.

$$
\begin{aligned}
&\mathit{sym} : \mathbf{char} \to \mathbf{re} \\
&\mathit{sym}\ c\ [s, n] = \mathbf{when}\ (s[n] = c)\ \{n + 1\}
\end{aligned}
$$

*sym* $c$ checks if $c$ occurs at position $n$, returning $n + 1$ if it does, and the empty set otherwise.

$$
\begin{aligned}
&\mathit{nil} : \mathbf{re} \\
&\mathit{nil}\ [s, n] = \{n\}
\end{aligned}
$$

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

The *nil* function simply returns the same index $n$ it received as an argument, since an empty string matches starting from any position. Sequencing via *seq* $r_1\ r_2$ checks first to see the possible ending positions from matching $r_1$, and carries on with $r_2$ from there.

$$
\begin{aligned}
&\mathit{bot} : \mathbf{re} \\
&\mathit{bot}\ \_ = \bot
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{alt} : \mathbf{re} \to \mathbf{re} \to \mathbf{re} \\
&\mathit{alt}\ r_1\ r_2\ [s, n] =\ x = r_1\ x \vee r_2\ x
\end{aligned}
$$

We still get the empty set from *bot*, and *alt* still unions the two sets of end positions.

$$
\begin{aligned}
&\mathit{star} : \square\mathbf{re} \to \mathbf{re} \\
&\mathit{star}\ [r]\ [s, n] = \mathbf{fix}\ \mathit{self}\ \mathbf{is}\ \{n\} \vee \mathbf{for}\ (m \in \mathit{self})\ r\ [s, m]
\end{aligned}
$$

[2]As a technical note, sets of pairs of integers do not form a finite-height lattice, so by typing this program is not an acceptable fixed point expression. However, since the positions in a string form a finite set, semantically it is fine. The original Datafun paper shows how one can define bounded fixed points to handle cases like this, so we will not be scrupulous.

As before, the *star* combinator takes a boxed regular expression as an argument, and for the same reason – we are implementing sequencing with a fixed point. One thing worth noting about this definition is that it is *left-recursive* – the definition takes the endpoints from the fixed point *self*, 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 From Seminaïve Evaluation to the Incremental λ-Calculus

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

```
path(X, Z) ← edge(X, Z).
path(X, Z) ← edge(X,Y), path(Y, Z).
```

Let's suppose that edge denotes a linear graph, $\{(a_1, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n)\}$. Then path should denote its reachability relation, $\{(a_i, a_j) \mid 1 \leqslant i < j \leqslant n\}$. How can we compute this relation? The naïve approach is to begin with nothing in the path relation and repeatedly apply its rules until nothing more is deducible. We can make this precise by explicitly time-indexing our rules:

$$path_{i+1}(X, Z) \leftarrow edge(X, Z) \qquad path_{i+1}(X, Z) \leftarrow path_i(X, Y) \wedge edge(Y, Z)$$

By omission $path_0 = \emptyset$. From this it's easy to see that inductively $path_i \subseteq path_{i+1}$. Consequently, at step $i + 1$ we re-deduce every fact known at step $i$. For example, suppose $path_i(a_j, a_k)$. Then at step $i + 1$ we apply the second rule to $edge(a_{j-1}, a_j)$ and deduce $path_{i+1}(a_{j-1}, a_k)$. But since we also have $path_{i+1}(a_j, a_k)$, at time $i + 2$ we deduce the very same thing again, and again at $i + 3$, $i + 4$, and so on.

Because we append edges one at a time, $path_i$ contains exactly paths of $i$ or fewer edges. Therefore it takes $n$ steps until we reach our fixed point $path_{n-1} = path_n$. Since step $i$ involves $|path_i| \in \Theta(i^2)$ deductions, we make $\Theta(n^3)$ deductions in total. This seems wasteful, since there are only $\Theta(n^2)$ paths in the final result.

Seminaïve evaluation avoids this 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:

$$\begin{aligned} dpath_0(X, Y) &\leftarrow edge(X, Y) \\ dpath_{i+1}(X, Z) &\leftarrow edge(X, Y) \wedge dpath_i(Y, Z) \\ path_{i+1}(X, Y) &\leftarrow path_i(X, Y) \vee dpath_i(X, Y) \end{aligned}$$

It's easy to show inductively that $dpath_i$ contains only paths *exactly* $i + 1$ edges long. Consequently $|dpath_i| \in \Theta(n - i)$ and we make $\Theta(n^2)$ deductions overall.[3]

### 3.1 Seminaïve evaluation as incremental computation

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

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

The naïve way to compute *step*'s fixed point is to iterate it: start with $path_0 = \emptyset$ and compute $path_{i+1} = step\ path_i$ for increasing $i$ until $path_i = path_{i+1}$. But since $path_i \subseteq step\ path_i$, each iteration re-computes every path found by the previous iteration. Following Datalog, we'd prefer to compute only the *change* between iterations. So consider $step'$ defined by:

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

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

Observe that

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

In other words, $step'$ tells us how $step$ changes as its input grows. Using this property, we can directly compute the changes $dpath_i$ between our iterations $path_i$:

$$\begin{aligned}
dpath_0 &= step\ \emptyset = edge \\
dpath_{i+1} &= step'\ dpath_i = \{(x, z) \mid (x, y) \in edge, (y, z) \in dpath_i\} \\
path_{i+1} &= path_i \cup dpath_i
\end{aligned}$$

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].

### 3.2 Change structures

To make precise the notion of change, an incremental λ-calculus associates every type $A$ with a *change structure*, consisting of:[4]

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

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

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

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:

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

Finally we come to the most interesting case: functions.

[4] 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 §9. Although we do not use change structures *per se* in our proofs, they are an important source of intuition.

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

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

If changes were given pointwise, taking only a base point, we'd stipulate that $df :: f \leadsto g$ iff $(\forall x)\ df\ x :: f\ x \leadsto g\ x$. 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 FnChange we have $df\ x\ dx :: f\ x \leadsto g\ y$ 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 also monotone on input changes $\Delta A$ — 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 $\Box$.

### 3.3 Zero-changes, derivatives, and faster fixed points

If $dx ::_A x \leadsto x$, we call $dx$ a *zero-change* to $x$. Usually zero-changes are rather boring — for example, a zero change to a set $x : \{\underset{\text{eq}}{A}\}$ is any $dx \subseteq x$, and so $\emptyset$ is always a zero change. However, there is one very interesting exception: function zero changes. Suppose $df ::_{A \to B} f \leadsto f$. This implies that

$$dx ::_A x \leadsto y \implies df\ x\ dx ::_B f\ x \leadsto f\ y \tag{1}$$

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:

$$dx ::_{\{\underset{\text{eq}}{A}\}} x \leadsto y \implies \mathit{step}'\ dx ::_{\{\underset{\text{eq}}{A}\}} \mathit{step}\ x \leadsto \mathit{step}\ y$$

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 3.1. *At every semilattice type* $L$*, we have* $\Delta L = L$ *and* $dx ::_L x \leadsto y \iff (x \vee dx) = y$.

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:

$$x_0 = \bot \qquad x_{i+1} = x_i \vee dx_i$$
$$dx_0 = f\ \bot \qquad dx_{i+1} = f'\ x_i\ dx_i$$

Let *semifix* $(f, f') = \bigvee_i x_i$. By induction and the derivative property, we have $dx_{i+1} :: x_i \leadsto f\ x_i$ and so $x_i = f^i\ x$, and therefore *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 $\mathcal{Derive}$ $e$ 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: $\phi e$, which computes $e$ faster by replacing fixed points with calls to *semifix*; and $\delta e$, which incrementalizes $\phi e$ so that we can

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

$$\textsc{Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{DVar}\ \frac{x :: A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Lam}\ \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.\ e : A \to B} \qquad \textsc{App}\ \frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash f : A}{\Gamma \vdash e\ f : B} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash \langle\rangle : 1}$$

$$\textsc{Pair}\ \frac{(\Gamma \vdash e_i : A_i)_i}{\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \times A_2} \qquad \textsc{Prj}\ \frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \pi_i\ e : A_i} \qquad \textsc{Inj}\ \frac{\Gamma \vdash e : A_i}{\Gamma \vdash \mathsf{in}_i\ e : A_1 + A_2}$$

$$\textsc{Case}\ \frac{\Gamma \vdash e : A_1 + A_2 \qquad (\Gamma, x_i : A_i \vdash f_i : B)_i}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ (\mathsf{in}_i\ x_i \to f_i)_i : B} \qquad \textsc{BoxI}\ \frac{[\Gamma] \vdash e : A}{\Gamma \vdash [e] : \square A} \qquad \textsc{BoxE}\ \frac{\Gamma \vdash e : \square A \qquad \Gamma, x :: A \vdash f : B}{\Gamma \vdash \mathbf{let}\ [x] = e\ \mathbf{in}\ f : B}$$

$$\textsc{Bot}\ \frac{}{\Gamma \vdash \bot : \underline{L}_{\mathrm{eq}}} \qquad \textsc{Join}\ \frac{(\Gamma \vdash e_i : \underline{L}_{\mathrm{eq}})_i}{\Gamma \vdash e_1 \vee e_2 : \underline{L}_{\mathrm{eq}}} \qquad \textsc{Set}\ \frac{([\Gamma] \vdash e_i : \underline{A}_{\mathrm{eq}})_i}{\Gamma \vdash \{e_i\}_i : \{\underline{A}_{\mathrm{eq}}\}} \qquad \textsc{SetFor}\ \frac{\Gamma \vdash e : \{A\} \qquad \Gamma, x :: A \vdash f : \underline{L}_{\mathrm{eq}}}{\Gamma \vdash \mathbf{for}\ (x \in e)\ f : \underline{L}_{\mathrm{eq}}}$$

$$\textsc{Eq}\ \frac{([\Gamma] \vdash e_i : \underline{A}_{\mathrm{eq}})_i}{\Gamma \vdash e_1 = e_2 : \mathbf{bool}} \qquad \textsc{IsEmpty}\ \frac{[\Gamma] \vdash e : \{1\}}{\Gamma \vdash \mathit{empty?}\ e : 1 + 1} \qquad \textsc{Split}\ \frac{\Gamma \vdash e : \square(A + B)}{\Gamma \vdash \mathit{split}\ e : \square A + \square B} \qquad \textsc{Fix}\ \frac{\Gamma \vdash e : \square(\underline{L}_{\mathrm{fix}} \to \underline{L}_{\mathrm{fix}})}{\Gamma \vdash \mathit{fix}\ e : \underline{L}_{\mathrm{fix}}}$$

FIGURE 3. Datafun core syntax and typing rules

compute the derivative of fixed point functions. In order to define $\phi$ and $\delta$ and show them correct, however, we first need a fuller account of Datafun's type system and semantics.

## 4 Typing and Semantics of Core Datafun

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 $[\Gamma]$ drops all monotone hypotheses from the context $\Gamma$, leaving only the discrete ones. The typing judgement $\Gamma \vdash e : A$ may be glossed as "under hypotheses $\Gamma$, the term $e$ has the 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 $\lambda$-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.

BoxI says that $[e]$ has type $\square A$ when $e$ has type $A$ in the stripped context $[\Gamma]$. This restricts $e$ to refer only to discrete variables, ensuring we don't smuggle any information we must treat monotonically into a discretely-ordered $\square$ expression. The elimination rule BoxE for $(\mathbf{let}\ [x] = e\ \mathbf{in}\ f)$ allows us to "cash in" a boxed expression $e : \square A$ 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 rule says that *split* $e : \Box A + \Box B$ whenever $e : \Box(A + B)$, distributing box across sum types.[5] The other direction, $\Box A + \Box B \to \Box(A + B)$, is already derivable, as is the isomorphism $\Box A \times \Box B \cong \Box(A \times B)$. In the examples we give, we use all these operations implicitly, to propagate discreteness onto the variables bound by a box pattern – in the pattern $[\langle \mathrm{in}_1\ x, \mathrm{in}_2\ y\rangle]$, we expect both the variables $x$ and $y$ to be discrete, which is information we propagate using these coercions. Semantically, all of these operations are the identity, as we will see in the following subsection.

This leaves only the rules for manipulating sets and other semilattices. Recall that sets form a semilattice in the inclusion order, with the empty set as least element and set union as join, and also that products of semilattices are semilattices (with order and operations given pointwise). So we take the *semilattice types* L to be the set type $\{A\}$, as well as units 1 and products of lattice types $L_1 \times L_2$. Then, the Bot rule says that the term $\bot$ is the least element of any semilattice type L, and the Join rule gives the type of joins, saying that $e_1 \vee e_2$ is of semilattice type L when each $e_i$ has type L. These constructors work for any semilattice L, but there are rules specialized to sets as well. The typing rule for eliminating sets, SetFor, is *almost* the monadic bind. However, we generalize it by not requiring the term **for** $(x \in e)\ e'$ to eliminate into a set type. Instead, we permit $e'$ to be *any* semilattice type, not just sets. Similarly, the introduction form Set is specific to set types, saying that $\{e_i\}_{i \in I}$ has type $\{A\}$ when each of the $e_i$ has type A. Furthermore, each $e_i$ has to typecheck in the discrete context, since membership relies on the non-monotonic property of equality. This can be seen in the Eq rule, which checks equality of two terms $e = f$ only when they are discrete, checking in stripped contexts. Finally, the rule Fix for fixed points *fix* $e$, takes an endofunction $e$ of type $\Box(L \to L)$ and yields an expression of type L. In addition to being of semilattice type, we also demand here that L is of finite height, ensuring that the recursion will terminate.

### 4.1 Semantics

The syntax of core Datafun can be interpreted in **Poset**, the category of partially ordered sets and monotone functions between them. That is, an object A of **Poset** is a pair $(A, \leqslant_A)$ consisting of a set A of elements, and a partial order relation $\leqslant_A \subseteq A \times A$ which is reflexive, transitive, and antisymmetric. A morphism $f : A \to B$ in **Poset** is a function on sets $f : A \to B$ which is monotone. That is, for all $a$ and $a'$ such that $a \leqslant_A a'$ we have $f(a) \leqslant_B f(a')$. The identity morphism is the identity function on the underlying sets, and composition of morphisms is just ordinary function composition. We will typically write composition in diagramamtic or "pipeline" order: given $f : A \to B$ and $g : B \to C$ we write $(f\,;g) : A \to C$.

*4.1.1 Cartesian Closed Structure.* The cartesian closed structure of **Poset** is largely the same as in **Set**.

*Products.* Given partial orders $(A, \leqslant_A)$ and $(B, \leqslant_B)$, we define the poset $(A, \leqslant_A) \times (B, \leqslant_B)$ to be $(A \times B, \leqslant_{A \times B})$ with the order defined as:

$$(a, b) \leqslant_{A \times B} (a', b') \iff a \leqslant_A a' \text{ and } b \leqslant_B b'$$

The verification that this forms a partial order is routine. To form the product object, we equip $A \times B$ with the projection maps $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$, which are defined by the first and second projection on the underlying sets respectively. Given two maps $f : A \to B$ and $g : A \to C$, the universal property of products is witnessed by $\langle f, g\rangle : A \to B \times C$, where

$$\langle f, g\rangle \triangleq a \mapsto \langle f(a), g(a)\rangle$$

[5] 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.

Type and Context Denotations

$$
\begin{array}{rclcrcl}
[\![1]\!] & = & 1 & \qquad & [\![A \to B]\!] & = & [\![A]\!] \Rightarrow [\![B]\!] \\
[\![\{\underset{\text{eq}}{A}\}]\!] & = & P[\![\underset{\text{eq}}{A}]\!] & & [\![A \times B]\!] & = & [\![A]\!] \times [\![B]\!] \\
[\![\Box A]\!] & = & \Box[\![A]\!] & & [\![A + B]\!] & = & [\![A]\!] + [\![B]\!]
\end{array}
$$

$$
[\![\Gamma]\!] = \prod_{H \in \Gamma} [\![H]\!] \qquad [\![x : A]\!] = [\![A]\!] \qquad [\![\color{green}{x} :: A]\!] = \Box[\![A]\!] \qquad [\![\Gamma \vdash A]\!] = \mathbf{Poset}([\![\Gamma]\!], [\![A]\!])
$$

Term Denotations

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

Auxilliary Operations

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

$$
\begin{array}{lcl}
\mathit{join}_0^L & : & 1 \to L \\
\mathit{join}_0^1 & = & !_1 \\
\mathit{join}_0^{L_1 \times L_2} & = & \langle \mathit{join}_0^{L_1}, \mathit{join}_0^{L_2} \rangle \\
\mathit{join}_0^{PA} & = & \mathit{join}_0^{PA}
\end{array}
\qquad
\begin{array}{lcl}
\mathit{join}_2^L & : & L \times L \to L \\
\mathit{join}_2^1 & = & !_{1 \times 1} \\
\mathit{join}_2^{L_1 \times L_2} & = & \alpha_{L_1, L_2} \,;\, \mathit{join}_2^{L_1} \times \mathit{join}_2^{L_2} \\
\mathit{join}_2^{PA} & = & \mathit{join}_2^{PA}
\end{array}
$$

$$
\begin{array}{lcl}
\alpha_{L,M} & : & (L \times M) \times (L \times M) \to (L \times L) \times (M \times M) \\
\alpha_{L,M} & = & ((l_1, m_1), (l_2, m_2)) \mapsto ((l_1, l_2), (m_1, m_2))
\end{array}
$$

Figure 4. Semantics of Datafun

The verification that these maps are monotone and that the relevant diagrams commute is routine.

*Exponentials.* Given partial orders $(A, \leqslant_A)$ and $(B, \leqslant_B)$, we define the poset $(A, \leqslant_A) \to (B, \leqslant_B)$ to be $(A \to B, \leqslant_{A\to B})$ with the order defined as:

$$f \leqslant_{A\to B} g \iff \forall a \in A.\ f(a) \leqslant_B g(a)$$

That this gives a partial order is immediate. We form the exponential object by equipping it with the evaluation map $\mathit{eval}_{A,B} \in \mathbf{Poset}((A \to B) \times A, B)$, defined via evaluation on sets:

$$\mathit{eval} \triangleq (f, x) \mapsto f(x)$$

Given $f : C \times A \to B$, the "exponential transpose" (i.e., currying operation) is defined as:

$$\begin{aligned} \lambda(f) &: C \to (A \to B) \\ \lambda(f) &\triangleq c \mapsto (\lambda a.\ f(c, a)) \end{aligned}$$

Verifying that these maps are monotone and that the relevant diagrams commute is straightforward.

*Coproducts.* Given partial orders $(A, \leqslant_A)$ and $(B, \leqslant_B)$, we define the poset $(A, \leqslant_A) + (B, \leqslant_B)$ to be $(A + B, \leqslant_{A+B})$ with the order defined by (with the understanding that if no clause matches there is no order relationship):

$$\begin{aligned} \mathrm{in}_1\, a \leqslant_{A+B} \mathrm{in}_1\, a' &\iff a \leqslant_A a' \\ \mathrm{in}_2\, b \leqslant_{A+B} \mathrm{in}_2\, b' &\iff b \leqslant_B b' \end{aligned}$$

It is easy to check this yields a partial order. We can construct the coproduct object by equipping this poset with the injections $\mathrm{in}_1 : A \to A + B$ and $\mathrm{in}_2 : B \to A + B$, which are given by the injections on the underlying sets. Given two maps $f : A \to C$ and $g : B \to C$, the universal property of coproducts is witnessed by $[f, g] : A + B \to C$, where

$$[f, g] \triangleq x \mapsto \begin{cases} f(a) & \text{when } x = \mathrm{in}_1\, a \\ g(b) & \text{when } x = \mathrm{in}_2\, b \end{cases}$$

It is easy to check the defined maps are monotone and that the relevant diagrams commute.

*4.1.2 The Discreteness Comonad.* Given a poset $(A, \leqslant_A)$ we define the discreteness operator $\Box(A, \leqslant_A)$ 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. Given a morphism $f : A \to B$, the functorial action is defined as:

$$\Box(f) \triangleq f$$

That is, the action on morphisms just gives back the same underlying function on sets. Checking that this preserves monotonicity is trivial, as is the preservation of identities and composition. This functor forms a comonad, with the extraction $\varepsilon_A : \Box A \to A$ and duplication $\delta_A : \Box A \to \Box\Box A$ given by identities on the underlying sets:

$$\begin{aligned} \varepsilon_A &: \Box A \to A & \delta_A &: \Box A \to \Box\Box A \\ \varepsilon_A &= a \mapsto a & \delta_A &= a \mapsto a \end{aligned}$$

Once we check that the monotonicity requirements are satisfied, this makes checking the comonad diagrams easy. It is also immediate that $\Box$ is a comonad monoidal with respect to *both* products and coproducts. That is, $\Box(A \times B) \simeq \Box A \times \Box B$, and also $\Box(A + B) \simeq \Box A + \Box B$. In both cases the isomorphism is witnessed in both directions by identity on the underlying elements. We will write $dist^{\Box}_{\times}$ to name the map witnessing distributivity of $\Box$ over products, and $dist^{\Box}_{+}$ to name the map witnessing distributivity of $\Box$ over coproducts.

*4.1.3 Sets and Lattices.* Given a poset $(A, \leqslant_A)$ we define the finite powerset poset $P(A, \leqslant_A)$ as $(PA, \subseteq)$, with subsets of $A$ as elements ordered by subset inclusion. This is obviously a semilattice, with the least element $join_0^{PA}$ given by the empty set, and binary join $join_2^{PA}$ given by union. If $A$ itself is finite, then this is also a *complete* semilattice. We can define the following collection of useful morphisms on sets:

$$\begin{array}{lcllcl} join_0^{PA} & : & 1 \to P(A) & join_2^{PA} & : & P(A) \times P(A) \to P(A) \\ join_0^{PA} & = & \langle\rangle \mapsto \emptyset & join_2^{PA} & : & \langle X, Y\rangle \mapsto X \cup Y \\ \\ singleton & : & \Box A \to PA & isEmpty & : & \Box PA \to 1 + 1 \\ singleton & = & a \mapsto \{a\} & isEmpty & = & X \mapsto \begin{cases} in_1\ \langle\rangle & \text{when } X = \emptyset \\ in_2\ \langle\rangle & \text{otherwise} \end{cases} \end{array}$$

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.

Finally, if $L$ is a complete lattice, and $f : A \times \Box B \to L$, then we can define we can also define the morphism $collect(f) : A \times PA \to L$ as follows:

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

We will use this to interpret for-comprehensions, since all the definable lattice types (sets, and products of lattices) in core Datafun are complete lattices. However, it is worth noting that the discreteness restrictions on *singleton* mean that powersets do not quite form a monad in **Poset**.

*4.1.4 Fixed Points.* Given a finite-height semilattice $L$, we can define a fixed point operation $fix$ : $\Box(L \to L) \to L$ as follows:

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

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

*4.1.5 Interpretation.* The semantic interpetation (defined over typing derivations) is given in figure 4. 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. 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.

## 4.2 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.

$$\textsc{Empty}\ \frac{}{\varepsilon \sqsubseteq \varepsilon} \qquad \textsc{Cons}\ \frac{\Gamma \sqsubseteq \Delta}{\Gamma, H \sqsubseteq \Delta, H} \qquad \textsc{Drop}\ \frac{\Gamma \sqsubseteq \Delta}{\Gamma \sqsubseteq \Delta, H} \qquad \textsc{Disc}\ \frac{\Gamma \sqsubseteq \Delta}{\Gamma, x : A \sqsubseteq \Delta, x :: A}$$

FIGURE 5. Weakening relation

$$\begin{aligned}
\Phi 1 &= 1 & \Delta 1 &= 1 \\
\Phi\{\underset{\text{eq}}{A}\} &= \{\Phi \underset{\text{eq}}{A}\} \quad \text{(see lemma 5.1)} & \Delta\{\underset{\text{eq}}{A}\} &= \{\underset{\text{eq}}{A}\} \\
\Phi(\Box A) &= \Box(\Phi A \times \Delta\Phi A) & \Delta(\Box A) &= 1 \\
\Phi(A \times B) &= \Phi A \times \Phi B & \Delta(A \times B) &= \Delta A \times \Delta B \\
\Phi(A + B) &= \Phi A + \Phi B & \Delta(A + B) &= \Delta A + \Delta B \\
\Phi(A \to B) &= \Phi A \to \Phi B & \Delta(A \to B) &= \Box A \to \Delta A \to \Delta B
\end{aligned}$$

FIGURE 6. $\Delta$ and $\Phi$ type transformations

We define the weakening relation $\Gamma \sqsubseteq \Delta$ in figure 5. This says that $\Delta$ is a weakening of $\Gamma$, either because (as usual) it has extra hypotheses (in rule Drop), or (additionally!) because a hypothesis in $\Gamma$ becomes discrete in $\Delta$ (rule Disc). The idea is that making a hypothesis discrete only increases the number of places it can be used.

LEMMA 4.1. *If* $\Gamma \vdash e : A$ *and* $\Gamma \sqsubseteq \Delta$ *then* $\Delta \vdash e : A$.

This follows by the usual induction on typing derivations.

## 5 The $\phi$ and $\delta$ Transformations

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

The speed-up transform $\phi e$ 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 $\delta e$. Since a derivative is a zero change, can $\delta e$ 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 $\lambda$-calculus' $\mathcal{Derive}$ [Cai et al. 2014], $\delta e$ will compute how $\phi e$ changes as its free variables change.

However, to speed up *fix* $e$ we don't want the change to $e$; 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* $e$ requires that $e : \Box(\underset{\text{fix}}{L} \to \underset{\text{fix}}{L})$: the use of $\Box$ prevents $e$ from changing! So the key strategy of our speed-up transformation is to **decorate expressions of type $\Box A$ with their zero-changes.** This makes derivatives available exactly where we need them: at *fix* expressions.

### 5.1 Typing $\phi$ and $\delta$

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

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

Lemma 5.1. $\Phi \underset{\text{eq}}{A} = \underset{\text{eq}}{A}$.

Proof. Induct on $\underset{\text{eq}}{A}$. $\square$

As we'll see in §5.3 and 5.4, $\phi$ and $\delta$ are mutually recursive. To make this work, $\delta e$ must find the change to $\phi e$ 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 $\phi$ and $\delta$ do to typing contexts. To understand this, it's helpful to look at what $\Phi$ 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.

If we view expressions as functions of their free variables, $\delta e$ will denote the *derivative* of the function $\phi e$ denotes. 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 $dx_1, \ldots, dx_n$.[6] 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 $\square$, $\Phi$, and $\Delta$:

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

(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 $\square$, $\Phi$, and $\Delta$ on the semantics of $\Gamma$:

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

These defined, we can state the types of $\phi e$ and $\delta e$:

Theorem 5.2 (Well-typedness). *If* $\Gamma \vdash e : A$, *then*

$$\Phi\Gamma \vdash \phi e : \Phi A$$
$$\square\Phi\Gamma, \Delta\Phi\Gamma \vdash \delta e : \Delta\Phi A$$

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

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

To get the hang of these context and type transformations, suppose $x :: A, y : B \vdash e : C$. Then theorem 5.2 tells us:

$$x :: \Phi A,\ dx :: \Delta\Phi A,\ y : \Phi B \vdash \phi e : \Phi C$$
$$x :: \Phi A,\ dx :: \Delta\Phi A,\ y :: \Phi B,\ dy : \Delta\Phi B \vdash \delta e : \Delta\Phi C$$

Along with the original program's variables, $\phi e$ requires zero change variables $dx$ for every discrete source variable $x$. Meanwhile, $\delta e$ 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").

[6] We assume throughout the paper as a matter of notational convenience that source programs contain no variables starting with the letter *d*.

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

FIGURE 7. Seminaïve speed-up translation, $\phi$

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

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

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

FIGURE 8. Seminaïve derivative translation, $\delta$

We now have enough information to tackle the definitions of $\phi$ and $\delta$ 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.

### 5.2 Fixed points

The whole purpose of $\phi$ and $\delta$ is to speed up fixed points, so let's start there. In a fixed point expression *fix* $e$, we know $e : \Box(\underset{\mathrm{fix}}{L} \to \underset{\mathrm{fix}}{L})$. Consequently the type of $\phi e$ is

$$
\begin{aligned}
\Phi(\Box(\underset{\text{fix}}{L} \to \underset{\text{fix}}{L})) &= \Box(\Phi(\underset{\text{fix}}{L} \to \underset{\text{fix}}{L}) \times \Delta\Phi(\underset{\text{fix}}{L} \to \underset{\text{fix}}{L})) \\
&= \Box((\Phi\underset{\text{fix}}{L} \to \Phi\underset{\text{fix}}{L}) \times (\Box\Phi\underset{\text{fix}}{L} \to \Delta\Phi\underset{\text{fix}}{L} \to \Delta\Phi\underset{\text{fix}}{L})) \\
&= \Box((\underset{\text{fix}}{L} \to \underset{\text{fix}}{L}) \times (\Box\underset{\text{fix}}{L} \to \Delta\underset{\text{fix}}{L} \to \Delta\underset{\text{fix}}{L}) && \text{by lemma 5.1, } \Phi\underset{\text{fix}}{L} = \underset{\text{fix}}{L} \\
&= \Box((\underset{\text{fix}}{L} \to \underset{\text{fix}}{L}) \times (\Box\underset{\text{fix}}{L} \to \underset{\text{fix}}{L} \to \underset{\text{fix}}{L}) && \text{by lemma 3.1, } \Delta\underset{\text{fix}}{L} = \underset{\text{fix}}{L}
\end{aligned}
$$

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

$$
\frac{\Gamma \vdash e : \Box((\underset{\text{fix}}{L} \to \underset{\text{fix}}{L}) \times (\Box\underset{\text{fix}}{L} \to \underset{\text{fix}}{L} \to \underset{\text{fix}}{L})}{\Gamma \vdash \mathit{semifix}\ e : \underset{\text{fix}}{L}}
\qquad
\begin{aligned}
&[\![\mathit{semifix}\ e]\!]\ \gamma = \mathit{semifix}\ (f, f') \\
&\quad \text{where } (f, f') = [\![e]\!]\ \gamma
\end{aligned}
$$

As for $\delta(\mathit{fix}\ e)$, since $e$ can't change (having $\Box$ type), neither can $\mathit{fix}\ e$ (or $\mathit{semifix}\ \phi e$). All we need is a zero change at type $\underset{\text{fix}}{L}$; by lemma 3.1, $\perp$ suffices.

### 5.3 Variables, $\lambda$, and application

At the core of a functional language are variables, $\lambda$, and application. The $\phi$ translation leaves these alone, simply distributing over subexpressions. On variables, $\delta$ yields the corresponding change variables. On functions and application, $\delta$ 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, $\delta e$. However, we are given an argument of type $\Box\Phi A$, but consulting theorem 5.2 for the type of $\delta e$, 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: $\delta e$ needs two arguments, the original input $\phi f$ and its change $\delta f$, 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 $\phi e$ and $\delta e$ don't use the same typing context. We're even boxing $\phi f$, 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 $\phi f$ needs is $\Phi\Gamma$. Since $\Box$ only makes a context stronger, we're safe. To emphasize this, we've marked all *discrete* uses of $\phi e$ inside $\delta e$ in pink. The same argument applies (all the more easily) when $\phi e$ is used in a monotone rather than a discrete position.

### 5.4 The discreteness comonad, $\Box$

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 $\delta[e] = \langle\rangle$; this follows from $\Delta\Phi\Box A = 1$, since boxed values cannot change.

Next, consider $\phi[e] = [\langle \phi e, \delta e \rangle]$. The intuition here is straightforward: $\phi$ needs to decorate $e$ with its zero change; since $e$ is discrete and cannot change, we use $\delta e$. However! In general, one cannot use $\delta$ inside the $\phi$ translation and expect the result to be well-typed; $\phi$ and $\delta$ require different typing contexts. To see this, let's apply theorem 5.2 to singleton contexts:

| $\Gamma$ | $\Phi\Gamma$ | $\Box\Phi\Gamma, \Delta\Phi\Gamma$ |
|---|---|---|
| $x : A$ | $x : \Phi A$ | $x :: \Phi A,\ dx : \Delta\Phi A$ |
| $x :: A$ | $x :: \Phi A,\ dx :: \Delta\Phi A$ | $x :: \Phi A,\ dx :: \Delta\Phi A$ |

$$
\begin{array}{ll}
\mathit{dummy}_{\{A\}_{\mathrm{eq}}}\ \_ = \{\} & \mathit{dummy}_{A \times B}\ \langle x, y\rangle = \langle \mathit{dummy}\ x, \mathit{dummy}\ y\rangle \\
\mathit{dummy}_1\ \langle\rangle = \langle\rangle & \mathit{dummy}_{A+B}\ (\mathrm{in}_i\ x) = \mathrm{in}_i\ (\mathit{dummy}\ x) \\
\mathit{dummy}_{\Box A}\ [x] = [\mathit{dummy}\ x] & \mathit{dummy}_{A \to B}\ f = \lambda x.\ \mathit{dummy}\ (f\ x)
\end{array}
$$

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

Luckily, although $\Phi\Gamma$ and $\Box\Phi\Gamma, \Delta\Phi\Gamma$ differ on monotone variables, they agree on discrete variables. And since $e$ is discrete, there *are* no monotone variables in $e$, justifying the use of $\delta e$ in $\phi[e] = [\langle \phi e, \delta e\rangle]$.

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

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

Since $x$ is a discrete variable, both $\phi f$ and $\delta f$ 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 $\delta e$ in $\delta f$, but this is unsurprising when you consider that its type is $\Delta\Phi\Box A = 1$.

### 5.5 Case analysis, *split*, and *dummy*

The derivative of case-analysis, $\delta(\textbf{case}\ e\ \textbf{of}\ (\mathrm{in}_i\ x_i \to f_i)_i)$, is complex. Suppose $\phi e$ evaluates to $\mathrm{in}_i\ x$ and its change $\delta e$ evaluates to $\mathrm{in}_j\ dx$. Since $\delta e$ is a change to $\phi e$, 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 $\mathrm{in}_i$ must remain the same.) So the desired change $\delta(\textbf{case}\ e\ \textbf{of} \ldots)$ is given by $\delta f_i$ 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 $\phi e$ and $\delta e$ 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 $\phi e$ yields a boxed pair of tagged values, $[\langle \mathrm{in}_i\ x, \mathrm{in}_j\ dx\rangle]$, we need $\phi(\mathit{split}\ e)$ to yield a tagged boxed pair, $\mathrm{in}_i\ [\langle x, dx\rangle]$. 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 $(\mathrm{in}_i\ \langle\rangle)$ with a tag that matches $\phi(\mathit{split}\ e)$ and $\phi e$; **case**-analysing $\phi e$ suffices.

### 5.6 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 slightly cleverer. In particular, let's restrict to sets, and suppose that $dx$ changes $x$ into $x'$ and $dy$ changes $y$ to $y'$. In particular, let's 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 $\phi e$ and $\phi f$, 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 $\phi e$ and $\phi f$. 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 §3), 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 $\phi$-translation is straightforward, with one hitch: because $x :: \underset{\text{eq}}{A}$ is a discrete variable, the inner loop $\phi f$ needs access to its zero change $dx :: \Delta\underset{\text{eq}}{A}$. And at eqtypes (although not in general), the *dummy* function computes zero changes:

Lemma 5.3. *dummy* $x \mathrel{::}_{\underset{\text{eq}}{A}} x \rightsquigarrow x$ *for any* $x : \underset{\text{eq}}{A}$.

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 $\delta($**for**$(x \in e)\ f)$ generates. First, if the set $\phi e$ we're looping over gains new elements $x \in \delta e$, we need to compute $\phi f$ over these new elements. Second, if the inner loop $\phi f$ changes, we need to add in its changes $\delta f$ for every element, new or old, in the looped-over set, $\phi e \vee \delta e$. Just as in the $\phi$-translation, we use **0**/*dummy* to calculate zero-changes to set elements.

### 5.7 Leftovers

The $\phi$ rules we haven't yet discussed simply distribute $\phi$ over subexpressions. The remaining $\delta$ 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 $\delta($*empty?* $e) =$ *empty?* $\phi e$, but as with $\delta($*split* $e)$, the zero-change here is at type $1 + 1$, so to get the tag right we use $\phi e$.

## 6 Proving the Seminaïve Transformation Correct

We formalize the intended behavior of $\phi e$ and $\delta e$ using a logical relation. Inductively on types $A$, letting $a, b \in [\![A]\!]$, $x, y \in [\![\Phi A]\!]$, and $dx \in [\![\Delta\Phi A]\!]$, we define $dx \mathrel{::}_A x \lightning a \to y \lightning b$, which may be glossed as "$x, y$ speed up $a, b$ respectively, and $dx$ changes $x$ into $y$", as follows:

$$
\begin{aligned}
\langle\rangle \mathrel{::}_1 \langle\rangle \lightning \langle\rangle \to \langle\rangle \lightning \langle\rangle &\iff \top \\
dx \mathrel{::}_{\{\underset{\text{eq}}{A}\}} x \lightning a \to y \lightning b &\iff (x, y, x \cup dx) = (a, b, y) \\
\langle\rangle \mathrel{::}_{\Box A} (x, dx) \lightning a \to (y, dy) \lightning b &\iff (a, x, dx) = (b, y, dy) \wedge dx \mathrel{::}_A x \lightning a \to y \lightning b \\
\vec{dx} \mathrel{::}_{A_1 \times A_2} \vec{x} \lightning \vec{a} \to \vec{y} \lightning \vec{b} &\iff (\forall i)\ dx_i \mathrel{::}_{A_i} x_i \lightning a_i \to y_i \lightning b_i \\
\mathrm{in}_i\ dx \mathrel{::}_{A_1 + A_2} \mathrm{in}_j\ x \lightning \mathrm{in}_k\ a \to \mathrm{in}_l\ y \lightning \mathrm{in}_m\ b &\iff i = j = k = l = m \wedge dx \mathrel{::}_{A_i} x \lightning a \to y \lightning b \\
df \mathrel{::}_{A \to B} f \lightning f_s \to g \lightning g_s &\iff (\forall dx \mathrel{::}_A x \lightning a \to y \lightning b) \\
&\phantom{\iff}\ df\ x\ dx \mathrel{::}_B f\ x \lightning f_s\ a \to g\ y \lightning g_s\ b
\end{aligned}
$$

This extends to contexts $\Gamma$, letting $\rho, \rho' \in [\![\Gamma]\!]$, $\gamma, \gamma' \in [\![\Phi\Gamma]\!]$, and $d\gamma \in [\![\Delta\Phi\Gamma]\!]$:

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

Our fundamental result is that $\phi$ and $\delta$ generate expressions which preserve this logical relation:

Theorem 6.1 (Fundamental). *If* $\Gamma \vdash e : A$ *and* $d\gamma ::_\Gamma \gamma \between \rho \to \gamma' \between \rho'$ *then*

$$[\![\delta e]\!]\ \langle \gamma, d\gamma \rangle ::_A [\![\phi e]\!]\ \gamma \between [\![e]\!]\ \rho \to [\![\phi e]\!]\ \gamma' \between [\![e]\!]\ \rho'$$

At eqtypes, it's easy to show inductively that $dx ::_{\underset{\text{eq}}{A}} x \between a \to y \between b$ implies $x = a$. Consequently, first-order closed programs compute the same result when $\phi$-translated:

Corollary 6.2 (First-order Correctness). *If* $\varepsilon \vdash e : \underset{\text{eq}}{A}$ *then* $[\![e]\!] = [\![\phi e]\!]$.

## 7 Applying the Seminaïve Transformation to Transitive Closure

Let's try applying the seminaïve transform to a simple Datafun program: the transitive closure function *tc* from §2.3.1:

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

In the process we'll discover that besides $\phi$ 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 $\phi$ and $\delta$ by hand is easiest when you work inside-out. At the core of transitive closure is a relation composition, $(e \bullet p)$, and at the core of relation composition is a **when**-expression. Let's take a look at its $\phi$ and $\delta$ translations:

$$\begin{aligned} \phi(\textbf{when}\ (y_1 = y_2)\ \{\langle x, z \rangle\}) &= \phi(\textbf{for}\ (\langle\rangle \in y_1 = y_2)\ \{\langle x, z \rangle\}) && \text{desugaring} \\ &= \textbf{for}\ (\langle\rangle \in y_1 = y_2)\ \phi\{\langle x, z \rangle\} && \text{omitting a needless } \textbf{let}\text{-binding} \\ &= \textbf{when}\ (y_1 = y_2)\ \{\langle x, z \rangle\} && \text{resugaring} \end{aligned}$$

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

$$\begin{aligned} &\delta(\textbf{when}\ (y_1 = y_2)\ \{\langle x, z \rangle\}) \\ &= \delta(\textbf{for}\ (\langle\rangle \in y_1 = y_2)\ \{\langle x, z \rangle\}) && \text{desugaring } \textbf{when} \\ &= \textbf{for}\ (\langle\rangle \in \delta(y_1 = y_2))\ \phi\{\langle x, z \rangle\} && \text{omitting needless } \textbf{let}\text{-bindings} \\ &\cup\ \textbf{for}\ (\langle\rangle \in \phi(y_1 = y_2) \cup \delta(y_1 = y_2))\ \delta\{\langle x, z \rangle\} \\ &= \textbf{for}\ (\langle\rangle \in \bot)\ \{\langle x, z \rangle\} \cup \textbf{for}\ (\langle\rangle \in \phi(y_1 = y_2) \cup \bot)\ \bot && \text{rules for } \phi(e = f) \text{ and } \delta\{e_i\}_i \\ &= \bot && \text{propagating } \bot \end{aligned}$$

The core insight here is that $y_1 = y_2$ can't change, and neither can $\{\langle x, z \rangle\}$. By propagating this information — for example, rewriting $(\textbf{for}\ (x \in \bot)\ e)$ to $\bot$ — we can simplify our derivative down to nothing. Now let's pull out and examine $\textbf{for}\ (\langle y_2, z \rangle \in t)\ \textbf{when}\ (y_1 = y_2)\ \{\langle x, z \rangle\}$. The $\phi$ translation is again a no-op.

$$\begin{aligned} &\delta(\textbf{for}\ (\langle y_2, z \rangle \in t)\ \textbf{when}\ (y_1 = y_2)\ \{\langle x, z \rangle\}) \\ &= \textbf{for}\ (\langle y_2, z \rangle \in dt)\ \phi(\textbf{when}\ (y_1 = y_2)\ \{\langle x, z \rangle\}) && \text{omitting needless } \textbf{let}\text{-bindings} \\ &\cup\ \textbf{for}\ (\langle y_2, z \rangle \in t \cup dt)\ \delta(\textbf{when}\ (y_1 = y_2)\ \{\langle x, z \rangle\}) \\ &= \textbf{for}\ (\langle y_2, z \rangle \in dt)\ \textbf{when}\ (y_1 = y_2)\ \{\langle x, z \rangle\} && \text{propagating } \bot \end{aligned}$$

Tackling the outermost **for** loop:

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

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 $\phi(tc\ [e])$. Working inside out, we start with the derivative of the loop body, $\delta(e \cup (e \bullet \mathrm{p}))$:

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

This requires a new optimization: by definition, $\delta e = de$. However, since $e$ is discrete we know it's not changing, and since it's of set type, $de$ may as well be the empty set. So we replace $\delta e$ with $\bot$ instead. Finally, putting everything together:

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

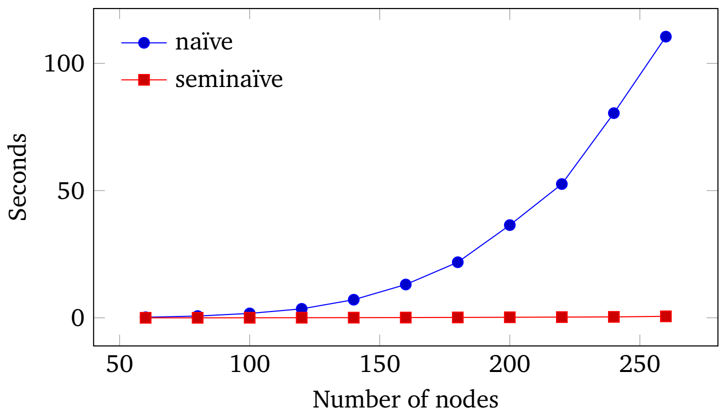
Examining the recurrence produced by this use of *semifix*, we recover exactly the seminaïve transitive closure algorithm we gave in §3.1:

$$
\begin{aligned}
x_0 &= \bot & x_{i+1} &= x_i \cup dx_i \\
dx_0 &= (\lambda \mathrm{p}.\ e \cup (e \bullet \mathrm{p}))\ \bot = e & dx_{i+1} &= (\lambda [p].\ \lambda dx.\ e \bullet \mathrm{dp})\ [x_i]\ dx_i = e \bullet dx_i
\end{aligned}
$$

## 8 Implementation and Optimization

To test whether the $\phi$ translation can produce the asymptotic performance gains we claim, we have implemented a compiler from a fragment of Datafun (omitting sum types) to Haskell. 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; relational joins, written in Datafun as nested **for**-loops, are compiled into nested loops. Consequently our performance is worse than any real Datalog engine. However, we do implement the $\phi$ translation, along with the following optimizations:

- Propagating $\bot$; for example, rewriting $(e \vee \bot) \rightsquigarrow e$ and $(\textbf{for}\ (x \in e)\ \bot) \rightsquigarrow \bot$.
- Replacing lattice-valued expressions that produce zero changes (for example, changes to discrete variables $\delta x$) by $\bot$. This makes $\bot$-propagation more effective.
- Recognising complex zero change expressions; for example, $\delta e\ \phi f\ \delta f$ is a zero change if $\delta e$ and $\delta f$ are. This allows more zero changes to be replaced by $\bot$, especially in higher-order code such as our regular expression example.




| Graph size | Naïve | Semi-naïve |
|---|---|---|
| 60 | 0.23 | 0.01 |
| 80 | 0.70 | 0.01 |
| 100 | 1.74 | 0.03 |
| 120 | 3.53 | 0.05 |
| 140 | 7.12 | 0.08 |
| 160 | 13.09 | 0.11 |
| 180 | 21.85 | 0.16 |
| 200 | 36.44 | 0.22 |
| 220 | 52.57 | 0.29 |
| 240 | 80.43 | 0.37 |
| 260 | 110.49 | 0.59 |

FIGURE 10. Naïve and seminaïve transitive closure on a linear graph

We benchmarked the transitive closure function *tc* from §2.3.1, compiled both naïvely and seminaïvely (i.e. omitting or using $\phi$), against the linear graph $\{(i, i+1) \mid 0 \leqslant i \leqslant n\}$. The results (figure 10) are consistent with our expectation that $\phi$-translated code, with appropriate optimisation, can perform asymptotically better than naïve evaluation.

## 9 Discussion and Related Work

*Nested fixed points.* The typing rule for *fix* $e$ requires $e : \Box(L_{\text{fix}} \to L_{\text{fix}})$. The $\phi$ translation takes advantage of this $\Box$, decorating expressions of type $\Box A$ with their zero-changes. However, it also prevents an otherwise valid idiom: in a nested fixed-point expression **fix** $x$ **is** $\ldots$ (**fix** $y$ **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 $\delta(\textit{fix}\ f) = \bot$: now that $f$ can change, so can *fix* $f$. Luckily, there is a simple and correct solution: $\delta(\textit{fix}\ f) = \textit{fix}\ [\delta f\ [\textit{fix}\ f]]$ [Arntzenius 2017]. However, to compute this new fixed point seminaïvely, we need a *second derivative*: the zero-change to $\delta f\ [\textit{fix}\ f]$. Indeed, for a program with fixed points nested $n$ deep, we need $n^{\text{th}}$ derivatives. We leave this to future work.

*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 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 examples of this line of work was to optimize bulk collection operations, and benchmarks showing asymptotic performance improvements were demonstrated. 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. 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 must have the freedom to overapproximate. In §5.6, we 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 expensive recomputations to construct set differences, and without this freedom it is doubtful that seminaïve evaluation would even be useful at all!

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.

Daan Leijen and Erik Meijer. 1999. Domain Specific Embedded Compilers. In *Proceedings of the 2Nd Conference on Domain-specific Languages (DSL '99)*. ACM, 109–122. https://doi.org/10.1145/331960.331977 event-place: Austin, Texas, USA.

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