DECONSTRUCTING DATALOG

by

MICHAEL ARNTZENIUS

A thesis submitted to the University of Birmingham for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
University of Birmingham
July 2021

Abstract

The deductive query language Datalog has found a wide array of uses, including static analysis (Smaragdakis and Bravenboer, 2010), business analytics (Aref et al., 2015), and distributed programming (Alvaro et al., 2010, 2011). Datalog is high-level and declarative, but simple and well-studied enough to admit efficient implementation strategies. For example, Whaley et al. found they could replace a hand-tuned C implementation of context-sensitive pointer analysis with a comparably-performing Datalog program that was 100x smaller (Whaley and Lam, 2004; Whaley et al., 2005).

However, Datalog's semantics are not stable under extensions. For instance, adding arithmetic operations breaks Datalog's termination guarantee. Despite this, nearly all practical implementations extend Datalog beyond its theoretical core to add niceties such as arithmetic, datatypes, aggregations, and so on. Moreover, pure Datalog cannot abstract over repeated code: one may express a static analysis over a *particular* program, but to express the same analysis over multiple programs, one must duplicate the analysis code for each program analyzed.

This thesis deconstructs Datalog from a categorical and type theoretic perspective to determine what makes it tick. Datalog's semantic guarantees are provided by brute syntactic restrictions, such as stratification and the absence of function symbols. In place of these, we find compositional semantic properties such as monotonicity, which we capture using types. We show that this permits integrating Datalog's features with those of typed functional languages, such as algebraic data types and higher order functions.

In particular, this thesis makes the following contributions:

1. We define and expound the semantics and metatheory of Datafun, a pure and total higher-order typed functional language capturing the essence of Datalog. Where Datalog has predicates defined by a restricted class of Horn clauses, Datafun has finite sets and set comprehensions; Datalog's bottom-up recursive queries become iterative fixed points; and Datalog's stratification condition becomes a matter of tracking monotonicity with types.

2. We show how to generalize seminaïve evaluation to handle higher-order functions. Seminaïve evaluation is a technique from the Datalog literature which improves the performance of Datalog's most distinctive feature: recursive queries. 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. We extend this transformation to handle higher-order programs written in Datafun.

3. In the process of generalizing seminaïve evaluation, we uncover a theory of incremental, monotone, higher-order computation, in which values change over time by growing larger, and programs respond incrementally to these increases.

Contents

1 Introduction **3**
- 1.1 Monotone fixed points . . . 3
- 1.2 Datalog . . . 6
  - 1.2.1 Termination and recursion . . . 7
  - 1.2.2 Stratified negation . . . 8
- 1.3 Datalog for static analysis . . . 10
- 1.4 What Datalog can't do . . . 11
  - 1.4.1 Functional abstraction . . . 11
  - 1.4.2 Semilattices other than set union . . . 12
  - 1.4.3 Arithmetic, user-defined functions, and aggregation . . . 13
  - 1.4.4 Compound data . . . 13
- 1.5 Our goal and strategy . . . 13

2 The Datafun Language **15**
- 2.1 Syntax sketch . . . 15
- 2.2 Examples . . . 17
  - 2.2.1 Set operations and relational algebra . . . 19
  - 2.2.2 Regular expression combinators . . . 20
  - 2.2.3 Regular expression combinators, take two . . . 21
  - 2.2.4 CYK parsing . . . 22
  - 2.2.5 Dataflow analysis . . . 23
- 2.3 Typing and denotational semantics . . . 25
  - 2.3.1 Typing rules . . . 25
  - 2.3.2 The category **Poset** and its structures . . . 27
  - 2.3.3 Interpretation of Datafun in **Poset** . . . 29
- 2.4 Operational semantics . . . 29
  - 2.4.1 A logical relation for termination . . . 32
  - 2.4.2 Metatheory of the logical relation . . . 34
  - 2.4.3 Proof of the fundamental theorem . . . 37

3 Seminaïve Evaluation **42**
- 3.1 Seminaïve evaluation as incremental computation . . . 42
- 3.2 Change structures for Datafun . . . 45
- 3.3 The structure of Δ**Poset** . . . 47
  - 3.3.1 Products . . . 47
  - 3.3.2 Sums . . . 48

3.3.3 Exponentials . . . 49
3.3.4 Semilattice change structures and seminaïve fixed points . . . 52
3.3.5 Fixed points and discreteness comonads . . . 53
3.4 The Φ and δ transforms . . . 54
3.4.1 Typing Φ and δ . . . 55
3.4.2 Fixed points . . . 59
3.4.3 Variables, λ-abstraction, and application . . . 59
3.4.4 The discreteness comonad, $\Box$ . . . 60
3.4.5 Case analysis, split, and dummy . . . 61
3.4.6 Semilattices and comprehensions . . . 61
3.4.7 Leftovers . . . 62
3.5 Proving the seminaïve transformation correct . . . 62

4 Implementation and Efficiency **65**
4.1 Applying the seminaïve transformation to transitive closure . . . 65
4.2 Implementation . . . 67
4.2.1 The compiler structure . . . 68
4.2.2 Compiling transitive closure . . . 70
4.2.3 Benchmarking seminaïve evaluation . . . 71
4.3 Change minimization . . . 77

5 Related Work **81**
5.1 Logic, higher-order abstraction, and semilattices . . . 81
5.1.1 Flix . . . 82
5.2 Incremental computation . . . 83
5.2.1 The incremental λ-calculus . . . 84
5.2.2 The monoidal approach to change . . . 89

6 Looking Back and Forward **92**
6.1 Directions forward . . . 93
6.2 Lessons and surprises . . . 95
6.3 Successes summarized . . . 96

A Proofs omitted from main text **98**
A.1 Datafun . . . 98
A.2 Seminaïve evaluation . . . 99

Chapter 1

Introduction

1.1 Monotone fixed points

A remarkable number of computational problems can be expressed as finding the least fixed point of a monotone map on a semilattice satisfying the ascending chain condition. The utility of Datalog is explained by the fact that it captures this pattern, albeit restricted to the semilattice of finite sets under union. To understand this pattern better, let's consider three examples of increasing complexity: (1) reachability in a graph; (2) single-source shortest paths; and (3) analyzing which variable assignments may reach a given line in a simple imperative program (called "reaching definitions").

Reachability Consider a graph and suppose we wish to find all nodes reachable from some designated start node. We proceed as follows: first, we put a check mark next to the start node; then, repeatedly, we pick a node and put a check next to it if any of its neighbors is checked. Once there are no nodes which we can mark this way – in particular, when there are no edges between checked and unchecked nodes – we are done; the reachable nodes are exactly the checked nodes.

Shortest paths Now suppose each edge e in the graph has an associated non-negative length d_e, and we wish to find the minimum distance to each reachable node. We use a small modification of the previous procedure: instead of a check mark, we annotate nodes v with the length d_v of the shortest path to them we've discovered so far. Initially we mark the start node with 0 and every other node with ∞ (representing "no known path"). Then, whenever an edge provides a shorter path to a node, we update its annotation – that is, for any edge $v \xrightarrow{e} u$ we may update $d_u := \min(d_u, d_v + d_e)$. Once no shortening edges exist, the annotations d_v cannot change, and we are done.[1]

Reaching definitions Finally, let's consider something seemingly completely different: statically analyzing a simple imperative program (figure 1.1). In particular, we wish to determine which assignments may reach a given program line; for example, the `print` on line 2 will receive only the value of `x` assigned on line 1, while the `print` on line 4 may receive the values assigned on both lines 1 and 5.

[1] The classic algorithm for single-source shortest paths, Dijkstra's algorithm, can be seen as a version of the algorithm we've described, but with a crucial optimization: it prioritizes which edges to consider next in a way that guarantees it never needs to revisit a node.

```
1  x := 0
2  print x
3  while true do
4    print x
5    x := x + 1
```

FIGURE I.I *Example program*

We determine this by propagating information along the control flow graph of our program. At each line we maintain a set of assignments (line-number/variable pairs) that we know can reach that line. Each line collects the assignments from all lines that can transfer control to it – usually the immediately preceding line, but loops and conditionals complicate this. However, lines which assign to a variable add themselves to this set, and discard other assignments to the same variable from incoming lines.

Once there is no line whose corresponding assignment-set is changing, the analysis is finished. See figure 1.2 for a step-by-step diagram of this process.

How do these three examples fit into our proposed pattern: finding the least fixed point of a monotone map on a semilattice satisfying the ascending chain condition? Let's break down each point in turn:

Fixed points In each example, we maintained some state – check marks or distance annotations on nodes, sets of reaching assignments on lines – that changed over time, and we terminated when there was no action we could take – no node, edge, or line we could examine – which would change this state. In other words, we halted once our state was *fixed* under our transition function.

Monotone Although our state changed over time, it did not change in arbitrary ways: there was a direction to it. Nodes went from unchecked to checked; distances to nodes decreased; and sets of reaching assignments grew. We can formalize this by giving our states a partial order representing the direction they change as computation progresses and information increases – an order with respect to which our state increases *monotonically* over time.

For example, in graph reachability a node's state is a boolean flag; we'll regard it as true if the node is checked, false otherwise. Since nodes go from unchecked to checked but not vice-versa, we say that false $<$ true. In single-source shortest-path our distances change toward zero, so we order them inversely, $\infty < \ldots < 3 < 2 < 1 < 0$. And in reaching definitions our assignment-sets change by gaining new elements, so we order them by the subset relation, $s \leqslant t \iff s \subseteq t$.

Semilattices In each example, we had some way of combining information from multiple sources. In graph reachability, we marked a node if *any* of its neighboring nodes were marked; in shortest paths, when there were multiple edges/paths into a node, we took the *minimum* among these competing options; and in reaching definitions, when a line could receive control from multiple lines, we took the *union* of their reaching assignment sets.

Not coincidentally, these operations – boolean disjunction, minimum, and union respectively – are the *least upper bound* operators for the partial orders we imposed on

		reaching assignments
1	`x := 0`	(x,1)
2	`print x`	
3	`while true do`	
4	`  print x`	
5	`  x := x + 1`	

STEP 1 The assignment on line 1 adds itself.

		reaching assignments
1	`x := 0`	(x,1)
2	`print x`	(x,1)
3	`while true do`	
4	`  print x`	
5	`  x := x + 1`	

STEP 2 Information propagates to line 2.

1	`x := 0`	(x,1)
2	`print x`	(x,1)
3	`while true do`	(x,1)
4	`  print x`	
5	`  x := x + 1`	

STEP 3 Information propagates to line 3.

1	`x := 0`	(x,1)
2	`print x`	(x,1)
3	`while true do`	(x,1)
4	`  print x`	(x,1)
5	`  x := x + 1`	

STEP 4 Information propagates to line 4.

1	`x := 0`	(x,1)
2	`print x`	(x,1)
3	`while true do`	(x,1)
4	`  print x`	(x,1)
5	`  x := x + 1`	(x,5)

STEP 5 The assignment on line 5 adds itself, discarding the assignment from line 1.

1	`x := 0`	(x,1)
2	`print x`	(x,1)
3	`while true do`	(x,1) (x,5)
4	`  print x`	(x,1) (x,5)
5	`  x := x + 1`	(x,5)

STEP 6 Information propagates from line 5 to line 3 because of the loop.

1	`x := 0`	(x,1)
2	`print x`	(x,1)
3	`while true do`	(x,1) (x,5)
4	`  print x`	(x,1) (x,5)
5	`  x := x + 1`	(x,5)

STEP 7 Information propagates to line 4.

1	`x := 0`	(x,1)
2	`print x`	(x,1)
3	`while true do`	(x,1) (x,5)
4	`  print x`	(x,1) (x,5)
5	`  x := x + 1`	(x,5)

STEP 8 No changes – done!

FIGURE 1.2 *Reaching definitions, step-by-step*

our states, making those partial orders into *join-semilattices*. In general, we write $x \vee y$ for the least upper bound/semilattice join of x and y. If our partial order represents the direction of increasing information, $x \vee y$ is a natural way to combine information: it includes all information from (is greater than) both x and y, but does not jump to unnecessary conclusions – it is the *least*, most conservative, upper bound.

Ascending chain condition Finally, in each case there was a limit on how much information we could possibly learn, and thus how many transitions we could take. For instance, in graph reachability, there were finitely many nodes, and each node could only transition from unchecked to checked once. This argument can be formalized by showing our partial order on states obeys the *ascending chain condition* (ACC), which asserts that there are no infinite strictly ascending chains, $x_0 < x_1 < x_2 < \ldots$; consequently, any process producing an increasing state-sequence must halt. We leave it as an exercise for the reader to convince themselves each of our state-posets satisfies this property.[2]

The essence of our computational pattern, then, is this: we follow rules which accumulate information *monotonically* until there is nothing left to learn (a *fixed point*). We usually need a way to combine information from multiple sources (our *semilattice*); and if we are to finish, we cannot keep learning forever (the *ascending chain condition*). As this thesis progresses, we will see more examples of this pattern and develop a more precise mathematical understanding of what it consists in. To do this, it will help to examine a language which is at once an instance of this pattern for us to deconstruct and a vehicle for expressing further instances of it: Datalog.

1.2 Datalog

Datalog may be seen either as a restricted logic programming language or an expressive database query language. To start with, we consider the former view, explaining Datalog in terms of deduction. Here is a simple Datalog program:

$$\mathsf{parent}(\mathsf{alice}, \mathsf{bob}).$$
$$\mathsf{parent}(\mathsf{bob}, \mathsf{charlie}).$$
$$\mathsf{grandparent}(X, Z) \leftarrow \mathsf{parent}(X, Y),\ \mathsf{parent}(Y, Z).$$

We can see each line, or clause, of this program as an inference rule. The first two lines are axioms, or inference rules with no premises; the last line is a rule with two premises. In inference rule notation we might write this:

$$\frac{}{\mathsf{parent}(\mathsf{alice}, \mathsf{bob})} \qquad \frac{}{\mathsf{parent}(\mathsf{bob}, \mathsf{charlie})} \qquad \frac{\mathsf{parent}(X, Y) \qquad \mathsf{parent}(Y, Z)}{\mathsf{grandparent}(X, Z)}$$

More formally, a Datalog program is a sequence of *clauses* terminated by periods. Each clause is an implication with one conclusion and an optional list of premises, written conclusion-first, "$B \leftarrow A_1, A_2, \ldots, A_n$."; or if there are no premises, simply "B.". The premises and conclusion

[2] Although for simplicity's sake we have presented our partial orders as ranging over booleans, distances, and finite sets respectively, our states are actually *maps* from nodes or lines into booleans, distances, or finite sets. Fortunately, if the domain is finite, maps into a poset satisfying ACC also satisfy ACC.

are *atoms* of first-order logic: a predicate applied to a sequence of terms, $P(T_1, ..., T_n)$, or a negation of the same, $\neg P(T_1, ..., T_n)$. Moreover, the conclusion of a clause must be positive, not negated.

As is the convention when interpreting inference rules, in Datalog the variables in a clause (for which we use capital letters X, Y, Z) are considered to be universally quantified: the logical interpretation of our third line, for example, is $(\forall X, Y, Z)\ \mathsf{parent}(X, Y) \wedge \mathsf{parent}(Y, Z) \implies \mathsf{grandparent}(X, Z)$.

The intended interpretation of a Datalog program is the set of all facts deducible from its clauses. We access this set via *queries* such as $\mathsf{grandparent}(\mathsf{alice}, X)$, which asks for a list of all X such that $\mathsf{grandparent}(\mathsf{alice}, X)$ is deducible. In general we allow conjunctive queries: lists of conjoined atoms, for instance the unlikely query $\mathsf{parent}(X, Y), \mathsf{parent}(Y, X)$, which asks for a pair of people each the parent of the other (or, in the case $X = Y$, a single person who is their own parent). Finally, a query without any variables, such as $\mathsf{grandparent}(\mathsf{charlie}, \mathsf{alice})$, amounts to asking a yes-or-no question: is the query deducible or not?

Computing the set of all deducible facts fits neatly into our description of monotone fixed points: applying inference rules is a monotone process, adding but never removing knowledge; we desire the fixed point of this process, where everything that can be deduced has been; our semilattice is sets of atoms under union. The ascending chain condition, however, does not obviously hold – perhaps there are infinitely many deducible facts? To answer this question, we must consider what differentiates Datalog from other logic programming languages.

1.2.1 Termination and recursion

Thus far our description of Datalog has not distinguished it from its ancestor Prolog; this is because the difference lies not so much in their syntax as in their semantics. Without further restrictions, whether a proposition is deducible from a collection of clauses is in general undecidable. Prolog's solution is to specify its proof search strategy. This lets Prolog programmers reason about the execution of their programs, but it can mean that some logically sensible recursive programs fail to terminate. For example:

$$\mathsf{reachable}(Y) \leftarrow \mathsf{reachable}(X), \mathsf{edge}(X, Y).$$
$$\mathsf{reachable}(\mathsf{start}).$$

These rules encode our graph reachability example: a node is reachable if one of its neighbors is, or if it is the start node. In Prolog, however, any query to reachable will loop.

This is because Prolog uses backward chaining (also called goal-directed or top-down) depth-first search: we start from a goal and reason backward, applying rules that might prove it. These rules are applied in the order they occur in the program, so to solve the query $\mathsf{reachable}(\mathsf{st\text{-}louis})$, Prolog will apply the first rule and try recursively to solve $\mathsf{reachable}(X)$ for unknown X. This in turn will apply the same rule, solving $\mathsf{reachable}(X_2)$ for unknown X_2, which will solve $\mathsf{reachable}(X_3)$ for unknown X_3, and so on and on interminably.[3]

[3] One natural approach to this problem is to keep backward chaining, but use a complete search strategy instead of depth-first search; this is the approach adopted by miniKanren (Friedman et al., 2005). This restores some declarativeness to logic programming; in particular, reordering rules can no longer cause unproductive infinite looping. However, introducing a "redundant" rule like $\mathsf{reachable}(X) \leftarrow \mathsf{reachable}(X)$ will still cause proof search to continue indefinitely (although it won't prevent any proofs from being found). And while this example is

Datalog takes a different tack: rather than fix a proof search strategy, it imposes limitations that keep proof search decidable. In particular, ignoring for now the issue of negation, it imposes two restrictions which keep all relations finite:

1. Clauses are *range-restricted:* all variables in the conclusion of a clause must occur positively in its premises. For example, the premiseless clause "$\mathsf{equal}(X, X)$." is disallowed; while logically sensible (it asserts equal is reflexive), it leaves the variable X unconstrained, which would generate an infinite relation.

2. Programs are *constructor-free:* predicate arguments are either atomic terms or variables. This prevents the introduction of new terms that don't already appear in the program, as this could also result in an infinite relation; for example, the relation containing all digit-lists (the use of a 'constructor' is underlined and red):

$$\mathsf{digits}(\mathsf{nil}).$$
$$\mathsf{digits}(\mathsf{cons}(X, Xs)) \leftarrow \mathsf{digit}(X), \mathsf{digits}(Xs).$$

Range-restriction and constructor-freedom together ensure that relations are finite and thus enforce the ascending chain condition. This permits the most common Datalog implementation strategy, *forward chaining*. In backward chaining we start from a goal ("can we reach St. Louis?") and reason backward, applying rules that might prove it. In forward chaining, we start from what we know ("we can reach Chicago, and there's an edge from Chicago to St. Louis") and apply rules whose premises are satisfied.

The weakness of forward chaining is that it's undirected: it deduces everything it can! If all you want to know is whether you can reach St. Louis, this is wasteful. On the other hand, it's much easier to know when to stop: when there is no rule whose application yields a new fact. This is the operational justification for range-restriction and constructor-freedom: by ensuring all predicates are finite, we guarantee forward-chaining deduction terminates.

1.2.2 Stratified negation

Negation and deduction have an interesting relationship. Consider the following program:

$$\mathsf{undeducible}() \leftarrow \neg\mathsf{undeducible}().$$

In classical logic, an implication $B \leftarrow A$ is equivalent to $B \vee \neg A$. Applying this, the above is equivalent to $\mathsf{undeducible}() \vee \neg\neg\mathsf{undeducible}()$, and thus simply to $\mathsf{undeducible}()$. Regarded as a rule of inference, however – as a strategy for deducing new facts from ones already known – this clause makes little sense: we cannot invoke it unless we have proved its conclusion is false!

To avoid this sort of gap between the logical meaning of a program and its interpretation as inference rules, Datalog allows only programs where uses of negation can be *stratified:* a recursively defined predicate (or mutually recursive group of predicates) cannot use its own negation in its definition.

---

contrived, the problem in general is fundamental: without further limitations on clauses, proof search is only semi-decidable, so while complete search strategies can guarantee finding all proofs, they cannot guarantee they'll *halt* after doing so. This is particularly important for the handling of negation-as-failure; see §1.2.2.

This restriction also avoids the need to make arbitrary choices. For example, this is disallowed:

$$\text{marry-rochester}() \leftarrow \neg\text{marry-st-john}().$$
$$\text{marry-st-john}() \leftarrow \neg\text{marry-rochester}().$$

This is classically equivalent to $\text{marry-rochester}() \vee \text{marry-st-john}()$. While sensible logically, this means answering simple yes-or-no queries requires making an arbitrary choice: if we query the propositions $\text{marry-rochester}()$ and $\text{marry-st-john}()$ we may consistently answer either *Rochester* or *St John* or *both*. The symmetry of our program makes answering either *Rochester* or *St John* unprincipled, and *both* is simply not deducible from the given rules.

Thus, unlike the preceding restrictions, stratified negation is not about finiteness or decidability; rather, it is motivated by interpreting a program as a set of rules for deduction and not merely a set of propositions. In other words, in Datalog *truth* is identified with *deducibility*.

Following this principle, we regard anything not deducible as false. This conforms with the programmer's expectation that anything not explicitly declared to be true is false. For example, returning to our graph reachability example:

$$\text{reachable}(Y) \leftarrow \text{reachable}(X), \text{edge}(X, Y).$$
$$\text{reachable}(\text{start}).$$

Regarded as mere propositions, these do not rule out the possibility that $\text{reachable}(X)$ is true for all vertices X, regardless of the edge relation (in other words, one model of these propositions makes reachable the entire vertex set). But this is clearly not the programmer's intent, which is to capture reachability: not every graph is completely connected!

The principle of regarding anything not deducible as false is known as *negation as failure:* to deduce $\neg\text{reachable}(X)$ it suffices to attempt to deduce $\text{reachable}(X)$ and fail.[4] Forward-chaining provides a natural implementation strategy for negation-as-failure: once we have deduced all facts of the form $\text{reachable}(X)$, if a particular such fact was not deduced, for example $\text{reachable}(\alpha\text{-centauri})$, we regard this as proof of its negation.

However, because a forward-chaining system must wait until all facts $\text{reachable}(X)$ are deduced before handling negative queries $\neg\text{reachable}(X)$, it cannot handle such negative queries in reachable's own definition. This is the operational justification for stratified negation: we must be able to stratify our Datalog program into layers, each of which may only use the negation of predicates defined in the preceding layers.

Returning to our computational pattern, if range-restriction and constructor-freedom are about establishing the ascending chain condition, stratification is about establishing *monotonicity* within each stratum – each recursively-defined relation or group of relations. These strata correspond to individual fixed point computations. But negation is non-monotone: as its input grows toward truth, its result decreases to falsehood. This makes applying a rule with a negated premise $\neg P(\ldots)$ dangerous: if as our knowledge grows we learn that $P(\ldots)$ holds after all, we must retract our conclusion because it is not deducible. But this violates the condition that our state – the set of deduced atoms – grows over time!

[4] The closely related *closed-world assumption* states that whatever is true is known to be true; conversely, anything not known to be true is false. If we take "known" to mean "deducible", then the closed-world assumption justifies negation as failure.

1.3 Datalog for static analysis

Datalog has been successfully applied in various domains: for business analytics (Aref et al., 2015), as a general purpose database query language (Hickey et al., 2012), in network protocols (Alvaro et al., 2010; Loo et al., 2009), and for implementing distributed systems algorithms (Alvaro et al., 2011; Conway et al., 2012). But probably Datalog's most significant real-world adoption has been as a tool for scalable static analysis.

Datalog makes defining simple static analyses almost trivial. For instance, the essence of reaching definitions analysis can be expressed as follows:

$$\text{reaches}(L, V, L) \leftarrow \text{assigns}(L, V).$$
$$\text{reaches}(L_{\text{dest}}, V, L_{\text{src}}) \leftarrow \neg\text{assigns}(L_{\text{dest}}, V),\ \text{reaches}(L_{\text{prev}}, V, L_{\text{src}}),\ \text{flows}(L_{\text{prev}}, L_{\text{dest}}).$$

If $\text{flows}(L_1, L_2)$ means that line L_1 may transfer control to L_2, and $\text{assigns}(L, V)$ means that line L assigns to variable V, the above defines $\text{reaches}(L_{\text{dest}}, V, L_{\text{src}})$ to mean that the assignment to V at line L_{src} may reach L_{dest}.

Datalog's fluency at defining static analyses is not limited to toy examples. For instance, it has been successfully commercialized by Semmle, a company which uses a custom in-house Datalog dialect & engine (Avgustinov et al., 2016) to do static analysis on large codebases to find potential security vulnerabilities – in particular variant analysis, which searches for variants of previously discovered issues. Their system has been used on NASA's Curiosity Rover,[5] at Microsoft,[6] and at the Nasdaq stock exchange company,[7] among others.

On the academic side, the DOOP project implements a state-of-the-art points-to analysis for Java code entirely in Datalog (Bravenboer and Smaragdakis, 2009). In a testimonial of sorts for the application of Datalog to static analysis, Smaragdakis and Bravenboer (2010) give several reasons why they found Datalog to be useful when implementing a scalable points-to analysis, compared in particular with a conventional language like Java or C++:

1. The high-level, declarative nature of Datalog allowed them to experiment with implementation techniques and algorithmic tweaks without having to entirely rewrite their analyses; for instance, they needed to carefully choose how to index their relations for maximal performance, a choice that would in a conventional language involve rewriting your data-access code. In their words:

   > *[W]e believe that our ability to efficiently optimize our implementation was largely due to the declarative specifications of analyses. Working at the Datalog level eliminated much of the artificial complexity of a points-to analysis implementation, allowing us to concentrate on indexing optimizations and on the algorithmic essence of each analysis.* (p. 1)

2. Datalog easily handles highly mutually-recursive relation definitions, which are common in static analysis: for example, *"the logic for computing a callgraph depends on having*

[5] https://web.archive.org/web/20210727023523/https://semmle.com/case-studies/semmle-nasa-landing-curiosity-safely-mars

[6] https://web.archive.org/web/20211130053831/https://msrc-blog.microsoft.com/2018/08/16/vulnerability-hunting-with-semmle-ql-part-1/

[7] https://web.archive.org/web/20210624221802/https://semmle.com/case-studies/semmle-nasdaq-improving-roi-and-reducing-time-market

points-to information for pointer expressions, which, in turn, requires a callgraph." (p. 3) They also observe that one of their crucial optimizations, performing exception analysis on-the-fly, *"would be quite hard to consider in a non-declarative context"* because it *"results in highly recursive definitions of core relations"* (p. 6).

3. Rather than hand-optimizing their code as might be necessary in a conventional language, in Datalog they could rely on the language implementation to do a good chunk of this work for them by applying decades of work on query optimisation:

   > *We relied on query optimization (i.e., intra-rule, as opposed to inter-rule, optimization) being performed automatically. This was crucial for performance and, although a straightforward optimization in the context of database relations, results in far more automation than programming in a mainstream high-level language.* (p. 6)

4. As we've seen with our examples so far, Datalog is very concise:

   > *Generally, the declarative nature of DOOP often allows for very concise specifications of analyses. We show in an earlier publication the striking example of the logic for the Java cast checking—i.e., the answer to the question "can type A be cast to type B?" The Datalog rules are almost an exact transcription of the Java Language Specification.* (p. 4)

We must also consider that static analysis blunts one of Datalog's primary drawbacks. As mentioned previously, one weakness of forward chaining is that it is undirected, deducing everything it can. If we only need the results of a narrow query, this can waste a lot of effort. Various optimizations and alternative evaluation strategies exist to address this problem, including *magic sets* (Bancilhon et al., 1986; Beeri and Ramakrishnan, 1987), which statically rewrites recursive relation definitions to produce only a subset of facts relevant to a given query (for instance, automatically transforming all-pairs graph reachability to single-source reachability); and *tabling* (Swift and Warren, 2012; Tekle and Liu, 2011), which hybridizes forward- and backward-chaining by careful use of memoization. In static analysis, however, the eagerness of forward-chaining is less problematic, as we generally wish to analyse the whole program exhaustively, whether for optimization or for bug-finding purposes.

1.4 What Datalog can't do

In §1.2 we discussed Datalog's semantics and the deliberate restrictions that make it tractable; in §1.3 we saw how this made Datalog an attractive language for writing static analyses. However, Datalog's restrictions are not without their drawbacks. In this section will consider four things which Datalog does not support and their use-cases: (1) functional abstraction, (2) semilattices other than set union, (3) arithmetic and aggregations, and (4) compound data.

1.4.1 Functional abstraction

Consider our graph-reachability example again:

$$\text{reachable}(\text{start}).$$
$$\text{reachable}(Y) \leftarrow \text{reachable}(X),\ \text{edge}(X, Y).$$

Suppose we wish to compute reachability over multiple different graphs. One way is to repeat ourselves:

$$\text{reachable1}(\text{start1}).$$
$$\text{reachable1}(Y) \leftarrow \text{reachable}(X),\ \text{edge1}(X, Y).$$
$$\text{reachable2}(\text{start2}).$$
$$\text{reachable2}(Y) \leftarrow \text{reachable}(X),\ \text{edge2}(X, Y).$$
$$\text{reachable3}(\text{start3}).$$
$$\text{reachable3}(Y) \leftarrow \text{reachable}(X),\ \text{edge3}(X, Y).$$
$$\vdots$$

This quickly becomes obnoxious if we wish to repeat something more complex than a two-liner like graph reachability. In an ordinary language, we would factor out this repeated code into a procedure or a function. Unfortunately, Datalog cannot do this: Datalog does not have functions, procedures, or modules, only relations; and relations are first-order, unable to manipulate or abstract over other relations. It is unclear how to lift this restriction without giving up the guarantees that make forward-chaining Datalog evaluation tractable.

1.4.2 Semilattices other than set union

We have shown how to define reachability and reaching-definitions in Datalog. What about our second example, single-source shortest paths? A naive approach at expressing this in Datalog might look like this:

$$\text{shortest}(\text{start}, 0).$$
$$\text{shortest}(Y, D_3) \leftarrow \text{shortest}(X, D_1),\ \text{edge}(X, Y, D_2),\ D_1 + D_2 = D_3.$$

The immediate issue here is the use of the infinite relation $D_1 + D_2 = D_3$; recall that infinite relations can cause a problem for a forward-chaining evaluator. However, in this case D_1 and D_2 are supplied by the first two premises, and together these allow computing D_3. The deeper problem is that, despite its name, $\text{shortest}(X, D)$ finds *all* distances D from start to X rather than only the shortest. Besides failing to capture our intent, if the graph has cycles, there may be infinitely many such distances, causing an infinite loop.

Datalog really only understands one semilattice – finite sets under union – and has no way to specify that multiple sources of information, like the distance D in $\text{shortest}(X, D)$, should be combined using a different strategy. This puts any computation using a custom semilattice out of reach. This particularly impacts Datalog's application to static analysis, where custom semilattices are frequently used to represent carefully-chosen approximations of the full set of values an expressions or variables may take on – for instance, "flat" lattices for constant propagation, or numeric intervals (or for multiple variables, convex polyhedra) under convex hull and intersection.

1.4.3 Arithmetic, user-defined functions, and aggregation

Pure Datalog does not permit using arithmetic or other functions. Although it is not difficult to extend a forward-chaining Datalog evaluator with support for these, and consequently almost everybody does so, this sacrifices Datalog's termination guarantee, as we saw with shortest paths. Even more problematic is support for aggregations, without which arithmetic by itself is of limited use. We also saw the need for aggregation in our shortest paths example: after calculating the lengths of multiple paths to the same node, we need to aggregate them and keep only their minimum.

In this example, our desired aggregation arises from a semilattice – we are taking the least upper bound $\bigvee_{p \in s} \text{length}(p)$ over some set s of paths p; to add support for other semilattices to Datalog, we need the ability to compute these semilattice aggregations. Semilattice aggregations are well-behaved because they are monotone: the value of $\bigvee_{x \in s} f(x)$ grows monotonically with respect to both the set s and the function f. Thus the ascending chain condition provides sufficient conditions for their termination when used as part of a fixed point computation. However, not all useful aggregations form semilattices; for instance summing or averaging do not. Aggregations like this may or may not be monotone depending upon the order involved. Thus adding aggregations without careful restrictions can invalidate Datalog's least-fixed-point semantics.

1.4.4 Compound data

Datalog's *constructor-freedom* prevents code that would directly manipulate compound data structures, like lists, trees, or even tuples. This restriction rules out some infinite relations, such as our "lists of digits" example from page 8, but it also rules out some legitimate, finite use-cases as well. For instance, Smaragdakis and Bravenboer (2010) added a macro system to their Datalog dialect to make writing context-sensitive static analyses less of a chore.

In a context-sensitive analysis, each rule depends upon a *context* representing an approximation of the conditions the code is executing under; for example, what call-site the function being analysed was invoked from. A deep context requires multiple pieces of data to specify (e.g. the preceding two or three functions on the call stack). It would be convenient to bundle this information into a single piece of data, hiding the choice of exactly how deep the context is, a choice orthogonal to the essence of the analysis. Although logically and operationally unproblematic, this use of compound data is not possible in Datalog, barring a macro system or some more principled extension.

1.5 Our goal and strategy

We started by introducing the computational pattern of monotone fixed points. We've seen that Datalog can express and compute some, but not all, instances of this pattern. Datalog's limitations are both practical – it is not obvious how to extend Datalog with higher-order abstraction – and theoretical – real world Datalog engines often support aggregation, arithmetic, and compound data, but in doing so raise the question of these features' semantics.

The goal of this thesis is to design a language which improves on Datalog's ability to express monotone fixed point computation over semilattices by finding ways to lift Datalog's restrictions without sacrificing either its simple semantics or its practical implementation

strategies. Our approach will be to combine Datalog with typed functional programming, on the basis of three hypotheses:

1. We can gain functional abstraction by mixing Datalog with higher-order functional programming. From a traditional logic programming perspective, this is a strange move: functions are a special-case of relations, so the "natural" way to make a logic language higher-order is to allow higher-order relations.

   Datalog's power, however, comes from carefully *limiting*, not *expanding*, how relations may be defined; precisely because of their generality, higher-order relations are more complex to implement than higher-order functions, especially if one wishes to keep a natural semantics (Charalambidis et al., 2013). By separating our facility for deduction (relations, queries, & fixed points) from our facility for abstraction (functions), we hope to gain the best of both worlds.

2. We can capture the restrictions that make Datalog work by deconstructing it type-theoretically. Type theory studies compositional properties of programs: if we recast syntactic restrictions (stratification, constructor-freedom) in terms of the properties they ensure (monotonicity, ACC), we can design a type system to capture these properties by finding compositional ways to provide them. Guided by this type system and its semantics, we can add practical features to our language, such as semilattices, aggregation, arithmetic, and compound datatypes, without sacrificing these properties.

Chapter 2

The Datafun Language

2.1 Syntax sketch

The idea behind Datafun is to capture the essence of Datalog in a typed, higher-order, functional setting. Since the key restriction that makes Datalog's combination of recursion and negation tractable – stratification – requires tracking *monotonicity,* we locate Datafun's semantics in the category **Poset** of partial orders and monotone maps. Since **Poset** is cartesian closed, it can interpret the simply typed λ-calculus, giving us a notation for writing monotone and higher-order functions. This lets us *abstract* over Datalog rules, something impossible in Datalog itself! In this section we sketch the construction of Datafun hewing closely to this semantic intuition.

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

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

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

Semantically, $\Box$ is a monoidal comonad or necessity modality, and so we base our syntax on Pfenning and Davies (2001)'s syntax for the necessity fragment of constructive S4 modal logic. This involves distinguishing two kinds of variable: discrete variables are in lowercase

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

$$
\begin{array}{rrcl}
\text{types} & A, B & ::= & \{\underset{EQ}{A}\} \mid 1 \mid A \times B \mid A + B \mid A \to B \mid \Box A \\
\text{eqtypes} & \underset{EQ}{A}, \underset{EQ}{B} & ::= & \{\underset{EQ}{A}\} \mid 1 \mid \underset{EQ}{A} \times \underset{EQ}{B} \mid \underset{EQ}{A} + \underset{EQ}{B} \\
\text{semilattices} & L, M & ::= & \{\underset{EQ}{A}\} \mid 1 \mid L \times M \\
\text{finite eqtypes}^{2} & \underset{FIN}{A}, \underset{FIN}{B} & ::= & \{\underset{FIN}{A}\} \mid 1 \mid \underset{FIN}{A} \times \underset{FIN}{B} \mid \underset{FIN}{A} + \underset{FIN}{B} \\
\text{fixtypes} & \underset{FIX}{L}, \underset{FIX}{M} & ::= & \{\underset{FIN}{A}\} \mid 1 \mid \underset{FIX}{L} \times \underset{FIX}{M} \\
\text{terms} & e, f & ::= & X \mid x \mid \lambda X.\, e \mid e\ f \mid () \mid (e, f) \mid \pi_i\ e \\
 & & & \mathsf{in}_i\ e \mid \textbf{case}\ e\ \textbf{of}\ (\mathsf{in}_i\ X_i \to f_i)_{i \in \{1,2\}} \\
 & & & \{e_i\}_i \mid \textbf{for}\ (x \in e)\ f \\
 & & & [e] \mid \textbf{let}\ [x] = e\ \textbf{in}\ f \\
 & & & e = f \mid \mathsf{empty?}\ e \mid \mathsf{split}\ e \\
 & & & \bot \mid e \vee f \mid \mathsf{fix}\ e \\
\text{monotone variables} & X, Y, Z & \text{are abstract names} & \\
\text{discrete variables} & x, y, z & \text{are abstract names} &
\end{array}
$$

FIGURE 2.1 *Datafun syntax*

$(x, y, \mathsf{foo}, \mathsf{bar})$, while monotone variables are capitals (X, Y, Z). Discrete variables may be used without restrictions, but monotone variables may only be used in ways that respect the ordering on their types: they must be used *monotonically*. This is enforced by a kind of variable hygiene: we remove monotone variables from scope within non-monotone expressions. For example, we cannot compare two monotone variables for equality, $X = Y$, because equality comparison is non-monotone. To aid the reader, we highlight non-monotone expressions with a light blue background; monotone variables bound outside of a non-monotone expression like $e = f$ may not be used within it. Putting this all together, we construct the type $\Box A$ with the non-monotone introduction form $[e]$ and eliminate it by pattern-matching, **let** $[x] = e$ **in** f, giving access to a discrete variable x.

Finally, to express Datalog's recursive queries, Datafun includes a fixed point combinator fix, which computes the least fixed point of a map f. To ensure termination, this map must be monotone and take a *semilattice type with decidable equality and no infinite ascending chains*, $\underset{FIX}{L}$. For our purposes, a semilattice is a partial order with a least element $\bot$ and a least upper bound operation $\vee$ ("join"). Finite sets (with the empty set as least element, and union as join) are an example, as are tuples of semilattices. As long as the semilattice has no infinite ascending chains, the chain of iterations $\bot \leqslant f(\bot) \leqslant f(f(\bot)) \leqslant \ldots$ is guaranteed to stabilize at the least fixed point $f^i(\bot) = f^{i+1}(\bot)$ for some finite i. Decidable equality ensures we can tell when this fixed point is reached. While not used by fixed point iteration itself, semilattice join is used to define f in all our motivating examples (following the pattern from §1.1), so we don't bother introducing "posets with bottom" as a concept separate from semilattices.[2]

[2] We are sweeping a few technical details under the rug here. First, for reasons which will not be explained until §3.3.5 we treat fix as if it were a non-monotone operator. Second, observe that the finite set type $\{\underset{EQ}{A}\}$ will possess infinite ascending chains if $\underset{EQ}{A}$ has infinitely many inhabitants. Thus we need to distinguish a class of *finite* eqtypes $\underset{FIN}{A}$. Although their grammars in figure 2.1 are identical, their intent is different. For example, if we extended Datafun with integers, they would form an eqtype, but not a finite one.

2.2 Examples

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

1. We mentioned earlier that Datafun's boolean type bool is ordered false $<$ true. This is because we encode booleans as sets of empty tuples, or the type $\{1\}$, so written because 1 is the "unit type" of empty tuples. We desugar true to the singleton $\{()\}$ and false to the empty set $\{\}$. In a loop over a boolean, **for** $(x \in e)$ f, the variable x contains no useful information; if for brevity we omit it, the condensed expression **for** (e) f may be thought of as a "one-sided" conditional:

$$\textbf{for}\ (e)\ f = \begin{cases} f & \text{if } e \text{ is true} \\ \bot & \text{if } e \text{ is false} \end{cases}$$

   Compared with encoding booleans as a sum type $1+1$, our approach has the advantage that it can express the type of "monotone" predicates $P : A \to \text{bool}$ such that $P(x)$ may change from false to true as x grows, but cannot revert from true to false.

2. We make use of pattern matching. Besides the usual sum/tuple patterns, we support box-patterns $[p]$ and equality-check patterns $!e$. Box patterns $[p]$ correspond to box-elimination **let** $[x] = e$ **in** f, and their effect is to make all of the variables bound by p discrete. The equality-check pattern $!e$ matches only a value equal to e – this is particularly useful when combined with set comprehensions.

3. We make use of set comprehensions, which can be desugared into the monadic operator **for** (Wadler, 1992). We also support looping/comprehending over only those elements of a set which match a certain pattern.

4. We express fixed points as a binding form, **fix** X **is** e, instead of explicitly supplying a lambda to the fix combinator, fix $[\lambda X.\ e]$.

5. We permit ourselves a top-level surface syntax similar to Haskell or SML. In particular, we allow curried function definitions, type aliases, and algebraic datatype definitions:

```
disjunction : bool → bool → bool
disjunction X Y = X ∨ Y
type colorname = string
data color = BLACK | NAMED colorname | RGB int int int
```

   We do not allow **data**-types to be defined recursively, so they can be easily desugared into sums of products in the standard way. Similarly, we allow ourselves n-ary tuples, which are easily desugared into nested binary tuples.

We summarize the desugaring rules we use in figure 2.2, excepting our top-level declarations and the desugaring of algebraic data types, which should be fairly familiar.

$$
\begin{array}{rrcl}
\text{types} & A, B & ::= & \ldots \mid \mathsf{bool} \\
\text{terms} & e, f & ::= & \ldots \mid \mathsf{true} \mid \mathsf{false} \\
 & & & \mathbf{for}\ (C)\ e \mid \{e \mid C\} \\
 & & & \mathbf{if}\ p \leftarrow e\ \mathbf{then}\ f_1\ \mathbf{else}\ f_2 \\
 & & & \mathbf{fix}\ X\ \mathbf{is}\ e \\
\text{discrete patterns} & p & ::= & x \mid \_ \mid\ !e \mid (p_1, p_2) \mid \mathsf{in}_i\, p \\
\text{loop clauses} & C, D & ::= & p \in e \mid e \mid C, D \mid
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{bool} & \xrightarrow{\text{desugar}} & \{1\} & \mathbf{for}\ ()\ e & \xrightarrow{\text{desugar}} & e \\
\mathsf{false} & \xrightarrow{\text{desugar}} & \{\} & \mathbf{for}\ (C, D)\ e & \xrightarrow{\text{desugar}} & \mathbf{for}\ (C)\ \mathbf{for}\ (D)\ e \\
\mathsf{true} & \xrightarrow{\text{desugar}} & \{()\} & \mathbf{for}\ (e)\ f & \xrightarrow{\text{desugar}} & \mathbf{for}\ (\_ \in e)\ f \\
\{e \mid C\} & \xrightarrow{\text{desugar}} & \mathbf{for}\ (C)\ \{e\} & \mathbf{for}\ (p \in e)\ f & \xrightarrow{\text{desugar}} & \mathbf{for}\ (x \in e)\ \mathbf{if}\ p \leftarrow x\ \mathbf{then}\ f\ \mathbf{else}\ \bot
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{fix}\ X\ \mathbf{is}\ e & \xrightarrow{\text{desugar}} & \mathsf{fix}\ [\lambda X.\ e] \\
\lambda[p].\ e & \xrightarrow{\text{desugar}} & \lambda Y.\ \mathbf{let}\ [p] = Y\ \mathbf{in}\ e \\
\mathbf{let}\ [(x, y)] = e\ \mathbf{in}\ f & \xrightarrow{\text{desugar}} & \mathbf{let}\ [z] = e\ \mathbf{in}\ \mathbf{let}\ [x] = [\pi_1\ z]\ \mathbf{in}\ \mathbf{let}\ [y] = [\pi_2\ z]\ \mathbf{in}\ f \\
\mathbf{if}\ \_ \leftarrow e\ \mathbf{then}\ f_1\ \mathbf{else}\ f_2 & \xrightarrow{\text{desugar}} & f_1 \\
\mathbf{if}\ x \leftarrow e\ \mathbf{then}\ f_1\ \mathbf{else}\ f_2 & \xrightarrow{\text{desugar}} & \mathbf{let}\ [x] = [e]\ \mathbf{in}\ f_1 \\
\mathbf{if}\ !e_1 \leftarrow e_2\ \mathbf{then}\ f_1\ \mathbf{else}\ f_2 & \xrightarrow{\text{desugar}} & \mathbf{case}\ \mathsf{empty?}\ (e_1 = e_2)\ \mathbf{of}\ \mathsf{in}_1\ \_ \to f_2;\ \mathsf{in}_2\ \_ \to f_1 \\
 & & \textit{(NB. empty? e yields } \mathsf{in}_1 \textit{ if e is empty, i.e. false.)} \\
\mathbf{if}\ (p_1, p_2) \leftarrow e\ \mathbf{then}\ f_1\ \mathbf{else}\ f_2 & \xrightarrow{\text{desugar}} & \mathbf{if}\ p_1 \leftarrow \pi_1\ e \\
 & & \mathbf{then}\ (\mathbf{if}\ p_2 \leftarrow \pi_2\ e\ \mathbf{then}\ f_1\ \mathbf{else}\ f_2) \\
 & & \mathbf{else}\ f_2 \\
\mathbf{if}\ \mathsf{in}_i\, p \leftarrow e\ \mathbf{then}\ f_1\ \mathbf{else}\ f_2 & \xrightarrow{\text{desugar}} & \mathbf{case}\ \mathsf{split}\ [e]\ \mathbf{of} \\
 & & \quad \mathsf{in}_i\, X \to \mathbf{let}\ [y] = X\ \mathbf{in}\ (\mathbf{if}\ p \leftarrow y\ \mathbf{then}\ f_1\ \mathbf{else}\ f_2) \\
 & & \quad \mathsf{in}_{(3-i)}\, X \to f_2
\end{array}
$$

Fresh variables introduced by desugaring are colored pink.

FIGURE 2.2 *Syntax sugar*

2.2.1 Set operations and relational algebra

One of the main features of Datafun is that it permits manipulating relations as first class values. In this subsection we will show how a variety of standard operations on sets can be represented in Datafun. The first operation we consider is testing membership:

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

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

Next we turn to set union and intersection. Union is baked into Datafun as the semilattice join, $x \cup y = x \vee y$, while intersection is definable using member, by taking the union of every singleton $\{x\}$ such that x is in both s and t:

$$\begin{aligned}&\_ \cap \_ : \{\underset{\text{EQ}}{A}\} \to \{\underset{\text{EQ}}{A}\} \to \{\underset{\text{EQ}}{A}\}\\ &S \cap T = \textbf{for}\ (x \in S,\ \mathsf{member}\ [x]\ T)\ \{x\}\end{aligned}$$

Using comprehensions, this could alternately be written as:

$$S \cap T = \{x \mid x \in S,\ \mathsf{member}\ [x]\ T\}$$

From now on, we'll use comprehensions whenever possible. For example, comprehensions make cross product and relational composition look almost exactly like their mathematical definitions:

$$\begin{aligned}&\_ \times \_ : \{\underset{\text{EQ}}{A}\} \to \{\underset{\text{EQ}}{B}\} \to \{\underset{\text{EQ}}{A} \times \underset{\text{EQ}}{B}\}\\ &S \times T = \{(x, y) \mid x \in S,\ y \in T\}\end{aligned}$$

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

The definitions of functional programming stalwarts *filter* and *map* (or in relational algebra terms, *select* and *project*) are slightly complicated by the need to be explicit about (non-) monotonicity:

$$\begin{aligned}&\mathsf{filter} : (\square\underset{\text{EQ}}{A} \to \mathsf{bool}) \to \{\underset{\text{EQ}}{A}\} \to \{\underset{\text{EQ}}{A}\}\\ &\mathsf{filter}\ F\ S = \{x \mid x \in S,\ F\ [x]\}\end{aligned}$$

$$\begin{aligned}&\mathsf{map} : \square(\square\underset{\text{EQ}}{A} \to \underset{\text{EQ}}{B}) \to \{\underset{\text{EQ}}{A}\} \to \{\underset{\text{EQ}}{B}\}\\ &\mathsf{map}\ [f]\ S = \{f\ [x] \mid x \in S\}\end{aligned}$$

Why is filter monotone in its function argument while map is not? Recall that functions are ordered pointwise while sets are ordered by inclusion, and observe that increasing the filtering function (making it *true* on more inputs) enlarges the result of filter, but the same does not hold for map:

$$\begin{array}{rcccccl}\mathsf{filter}\ (\leqslant 0)\ \{0,1\} & = & \{0\} & \subseteq & \{0,1\} & = & \mathsf{filter}\ (\leqslant 1)\ \{0,1\}\\ \mathsf{map}\ (\leqslant 0)\ \{0,1\} & = & \{\mathsf{true},\ \mathsf{false}\} & \not\subseteq & \{\mathsf{true}\} & = & \mathsf{map}\ (\leqslant 1)\ \{0,1\}\end{array}$$

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

$$\neg : \square\mathsf{bool} \to \mathsf{bool}$$
$$\neg[t] = \textbf{case}\ \mathsf{empty?}\ t\ \textbf{of}\ \mathsf{in}_1\ () \to \mathsf{true};\ \mathsf{in}_2\ () \to \mathsf{false}$$

$$\_ \setminus \_ : \{A_{EQ}\} \to \square\{A_{EQ}\} \to \{A_{EQ}\}$$
$$S \setminus [t] = \{x \mid x \in S,\ \neg[\mathsf{member}\ [x]\ t]\}$$

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

Finally, generalizing our Datalog graph reachability example in §1.2, we can define the transitive closure of a relation:

$$\mathsf{trans} : \square\{A_{FIN} \times A_{FIN}\} \to \{A_{FIN} \times A_{FIN}\}$$
$$\mathsf{trans}\ [\mathsf{edge}] = \textbf{fix}\ R\ \textbf{is}\ \mathsf{edge} \vee (\mathsf{edge} \bullet R)$$

This definition uses a least fixed point, just like the mathematical definition – a transitive closure is the least relation R including both the original relation edge and the composition of edge with R. However, a peculiar feature of this definition is that the argument type is $\square\{A_{FIN} \times A_{FIN}\}$; transitive closure takes a *discrete* relation. This is because we must use the relation within the fixed point, and Datafun treats fix as a discrete operator. This restriction is artificial – transitive closure is semantically a monotone operation – but its explanation will have to wait until §3.3.5.

2.2.2 Regular expression combinators

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

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

$$\mathsf{length} : \square\mathsf{string} \to \mathsf{int}$$
$$\mathsf{length}\ [s] = \text{the length of } s$$

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

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

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

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

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

$$\mathsf{sym} : \Box\mathsf{char} \to \mathsf{regex}$$
$$\mathsf{sym}\ [c]\ [s] = \{(i, i+1) \mid (i, !c) \in \mathsf{chars}\ [s]\}$$

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

$$\mathsf{nil} : \mathsf{regex}$$
$$\mathsf{nil}\ [s] = \{(i, i) \mid (i, \_) \in \mathsf{chars}\ [s]\} \vee \{(\mathsf{length}\ [s], \mathsf{length}\ [s])\}$$

Appending regexes R_1, R_2 amounts to relation composition, since we wish to find all substrings consisting of adjacent substrings $s_i \ldots s_{j-1}$ and $s_j \ldots s_{k-1}$ matching R_1 and R_2 respectively:

$$\mathsf{seq} : \mathsf{regex} \to \mathsf{regex} \to \mathsf{regex}$$
$$\mathsf{seq}\ R_1\ R_2\ S = R_1\ S \bullet R_2\ S$$

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

$$\mathsf{alt} : \mathsf{regex} \to \mathsf{regex} \to \mathsf{regex}$$
$$\mathsf{alt}\ R_1\ R_2\ S = R_1\ S \vee R_2\ S$$

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

$$\mathsf{star} : \Box\mathsf{regex} \to \mathsf{regex}$$
$$\mathsf{star}\ [r]\ [s] = \mathsf{nil}\ [s] \vee \mathsf{trans}\ [r\ [s]]$$

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

2.2.3 Regular expression combinators, take two

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

$$\textbf{type}\ \mathsf{regex} = \Box(\mathsf{string} \times \mathsf{int}) \to \{\mathsf{int}\}$$

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

[3] Technically the inclusion order on sets of integer pairs does not satisfy the ascending chain condition, so this use of trans is not well-typed. However, since the positions in a particular string form a finite set, semantically there is no issue.

```
sym : □char → regex
sym [c] [(s, i)] = {i + 1 | !(i, c) ∈ chars [s]}

nil : regex → regex
nil [(s, i)] = {i}
```

Appending regexes seq R_1 R_2 simply applies R_2 starting from every ending position that R_1 can find:

```
seq : regex → regex → regex
seq R1 R2 [(s, i)] = for (j ∈ R1 [(s, i)]) R2 [(s, j)]
```

Regex alternation alt is effectively unchanged:

```
alt : regex → regex → regex
alt R1 R2 X = R1 X ∨ R2 X
```

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

```
star : □regex → regex
star [r] [(s, i)] = fix X is ({i} ∨ for (j ∈ X) r [(s, j)])
```

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

2.2.4 CYK parsing

Parsing can be understood logically: a parse tree is a proof that a string belongs to a language, and parsing is proof search (Shieber et al., 1995). One of the simplest parsing algorithms is the Cocke-Younger-Kasami (CYK) algorithm for parsing grammars in Chomsky normal form; that is, where each production is either of the form $A \to B\ C$ or $A \to \vec{a}$, with A, B, C ranging over nonterminals and $\vec{a}$ over strings of terminals. Fix a Chomsky-normal grammar G and a word $w = w_0 w_1 ... w_{n-1}$ to be parsed, and write $w_{i..j}$ for the substring $w_i ... w_{j-1}$. Now, we introduce a family of predicates $A(i, j)$ (sometimes called *facts*), intended to represent the proposition that $w_{i..j}$ is generated by the nonterminal A. Then, we can specify the CYK algorithm with the following two inference rules:

$$\frac{(A \to B\ C) \in G \qquad B(i,j) \qquad C(j,k)}{A(i,k)} \qquad\qquad \frac{(A \to \vec{a}) \in G \qquad \vec{a} = w_{i..j}}{A(i,j)}$$

Then the whole word w is generated by the start symbol S if $S(0, n)$ is deducible.

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

```
type symbol = string
data rule = STRING string | CONCAT symbol symbol
type grammar = {symbol × rule}
type fact = symbol × int × int
```

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

```
length : □string → int
range : □int → int → {int}
substring : □(string × int × int) → string
(+) : int → int → int
(−) : int → □int → int
```

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

```
iter : □string → □grammar → {fact} → {fact}
iter [text] [grammar] F =
    { (a, i, k) | (a, CONCAT b c) ∈ grammar, (!b, i, j) ∈ F, (!c, !j, k) ∈ F }
  ∪ { (a, i, i + length s) | (a, STRING s) ∈ grammar,
                             i ∈ range [0] (length text − [length s]),
                             substring [text, i, i + length s] }
```

We can then use iter to implement the parse function:

```
parse : □string → □grammar → {symbol}
parse [text] [grammar] =
  {a | (a, !0, !(length text)) ∈ fix X is iter [text] [grammar] X}
```

This finds all nonterminals in grammar that generate the entire string text.

This program is not expressible in Datalog, because Datalog provides no way to *abstract* over grammars. The rules of a grammar are easily represented as Datalog relations — but since Datalog is first-order, it cannot parameterize one relation by another; so there is no way in Datalog to express a generic parser. This demonstrates one of the key benefits of moving to a functional language like Datafun.

2.2.5 Dataflow analysis

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

```
type var = string
type label = int
data op = EQ | LE | ADD | SUB | MUL | DIV
data atom = VAR var | NUM int
data expr = ATOM atom | APPLY op atom atom
data statement = ASSIGN var expr | IF expr label label
type program = {label × statement}
```

We represent a program as a set of nodes. Each node has a label and contains a statement, either an assignment (ASSIGN) or a conditional jump (IF). Valid programs p associate any label l with at most one node $(l, s) \in p$. In what follows, we use a few trivial functions whose definitions we omit.

```
labels : program → {label}
uses : □statement → {var}
defines : □statement → {var}
```

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

Given a program, we can recover its 1-step control flow graph with the flow function:

```
type flow = {label × label}
flow : program → flow
flow P = for ((i, s) ∈ P)
           if IF _ j k ← s
           then {(i, j), (i, k)}
           else {(i, i + 1) | !(i + 1) ∈ labels P}
```

This says that if a program node (i, s) is a conditional jump, IF _ j k, then it may flow to either j or k; otherwise, it flows to the next program position $i + 1$ if it exists.

Using this we can define liveness analysis, one of the classic backwards dataflow analyses. The function live takes a program P and its flow graph F and produces a set of label/variable pairs (i, v) indicating that the variable v is live at program point i.

```
live : □program → □flow → {label × var}
live [program] [flow] =
  fix L is
  for ((i, s) ∈ program)
    ({i} × uses s)
  ∪ {(i, v) | (!i, j) ∈ flow, (!j, v) ∈ L, ¬[member [v] (defines s)]}
```

At each label i, a variable v is live if either of two conditions holds: (1) it is used by the current statement s, or (2) it is live at some label j to which i flows, and which does not define the variable v for itself.

Next we give one of the classic forwards dataflow analyses: reaching definitions. This determines which program points the value assigned by a particular statement (a "definition") can reach.

```
reaching : □program → □flow → {var × label × label}
reaching [program] [flow] =
  fix R is
  for ((k, s) ∈ program)
    {(v, k, k) | v ∈ defines s}
  ∪ {(v, i, k) | (j, !k) ∈ flow, (v, i, !j) ∈ R, ¬[member [v] (defines s)]}
```

The function reaching takes a program and its flow graph, and returns a set of tuples (v, i, j) indicating that the definition of v at i might reach the program point j. These tuples are

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

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

$$\textsc{unit}\ \frac{}{\Gamma \vdash () : 1} \qquad \textsc{pair}\ \frac{(\Gamma \vdash e_i : A_i)_i}{\Gamma \vdash (e_1, e_2) : A_1 \times A_2} \qquad \textsc{prj}\ \frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \pi_i\ e : A_i} \qquad \textsc{inj}\ \frac{\Gamma \vdash e : A_i}{\Gamma \vdash \mathsf{in}_i\ e : A_1 + A_2}$$

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

$$\textsc{letbox}\ \frac{\Gamma \vdash e : \Box A \qquad \Gamma, x :: A \vdash f : B}{\Gamma \vdash \textbf{let}\ [x] = e\ \textbf{in}\ f : B} \qquad \textsc{bot}\ \frac{}{\Gamma \vdash \bot : L} \qquad \textsc{join}\ \frac{(\Gamma \vdash e_i : L)_i}{\Gamma \vdash e_1 \vee e_2 : L} \qquad \textsc{set}\ \frac{(\lceil \Gamma \rceil \vdash e_i : \underset{EQ}{A})_i}{\Gamma \vdash \{e_i\}_i : \{\underset{EQ}{A}\}}$$

$$\textsc{for}\ \frac{\Gamma \vdash e : \{\underset{EQ}{A}\} \qquad \Gamma, x :: \underset{EQ}{A} \vdash f : L}{\Gamma \vdash \textbf{for}\ (x \in e)\ f : L} \qquad \textsc{eq}\ \frac{(\lceil \Gamma \rceil \vdash e_i : \underset{EQ}{A})_i}{\Gamma \vdash e_1 = e_2 : \mathsf{bool}} \qquad \textsc{empty}\ \frac{\lceil \Gamma \rceil \vdash e : \{1\}}{\Gamma \vdash \mathsf{empty?}\ e : 1 + 1}$$

$$\textsc{split}\ \frac{\Gamma \vdash e : \Box(A + B)}{\Gamma \vdash \mathsf{split}\ e : \Box A + \Box B} \qquad \textsc{fix}\ \frac{\Gamma \vdash e : \Box(\underset{FIX}{L} \to \underset{FIX}{L})}{\Gamma \vdash \mathsf{fix}\ e : \underset{FIX}{L}}$$

FIGURE 2.3 *Datafun typing rules*

generated by two rules, corresponding to the two clauses in reaching's inner loop: (1) if v is defined at i, then it reaches i, and (2) if the definition of v at i reaches j and j flows to k then the definition also reaches k unless j has an intervening definition of v.

2.3 Typing and denotational semantics

Our guiding intuition so far has been that Datafun is a language for writing monotone, higher-order functions. Here we substantiate that intuition by giving typing rules for core Datafun and showing how to interpret well-typed Datafun terms into **Poset**, the category of partially ordered sets and monotone maps.

2.3.1 Typing rules

The syntax of core Datafun is given in figure 2.1 and its typing rules in figure 2.3. 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 typing judgement $\Gamma \vdash e : A$ may be read as "assuming the variables in Γ have their given types, the term e has type A; moreover, e is monotone with

respect to the monotone variables in Γ".

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

BOX says that $[e]$ has type $\Box A$ when e has type A in the stripped context $\lceil\Gamma\rceil$. The stripping operation $\lceil\Gamma\rceil$ drops all monotone hypotheses from the context Γ, removing them from scope in e and implementing the "variable hygiene" discussed in §2.1. This ensures we don't smuggle any information we must treat monotonically into a discretely-ordered $\Box$ expression. The elimination rule LETBOX for $(\textbf{let}\ [x] = e\ \textbf{in}\ f)$ allows us to "cash in" a boxed expression $e : \Box A$ by binding its result to a discrete variable $x :: A$ in the body f.

At this point, our typing rules correspond to standard constructive S4 modal logic (Pfenning and Davies, 2001). We get to Datafun by adding a handful of domain-specific types and operations. First, SPLIT provides an operator $\text{split} : \Box(A + B) \to \Box A + \Box B$ to distribute box across sum types. The other direction, $\Box A + \Box B \to \Box(A + B)$, is already derivable, as is the isomorphism $\Box A \times \Box B \cong \Box(A \times B)$. This is used implicitly by box pattern-matching – e.g., in the pattern $[(\text{in}_1\, x, \text{in}_2\, y)]$, the variables x and y are both discrete, which is information we propagate via these conversions. Semantically, all of these operations are the identity, as we shall see shortly.[4]

This leaves only the rules for manipulating sets and other semilattices. BOT and JOIN tell us that $\bot$ and $\vee$ are valid at any semilattice type L, that is, at sets and products of semilattice types. The rule for set-elimination, FOR, is almost a monadic bind. However, we generalize it by allowing $\textbf{for}\ (x \in e)\ f$ to eliminate into any semilattice type, not just sets, denoting a "big semilattice join" rather than a "big union".

The set-introduction rule SET gives $\{e_i\}_{i\in I}$ type $\{\underset{\text{EQ}}{A}\}$ when each of the e_i has type $\underset{\text{EQ}}{A}$. Just as in BOX, each e_i has to typecheck in a stripped context; constructing a set is a discrete operation, since $1 \leqslant 2$ but $\{1\} \not\subseteq \{2\}$.

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

Finally, the rule FIX says that the fixed point fix combinator accepts a boxed function $f : \Box(\underset{\text{FIX}}{L} \to \underset{\text{FIX}}{L})$ and returns a value of type $\underset{\text{FIX}}{L}$. The restriction to "fixtypes" ensures $\underset{\text{FIX}}{L}$ has no infinite ascending chains, guaranteeing the recursion will terminate. The restriction to boxed functions, treating fix as a non-monotone operator, is motivated not by our semantics but by our strategy for evaluating Datafun efficiently. This will be explained in detail in §3.3.5, but as a foreglimpse, to evaluate Datafun efficiently, we incrementalize monotone functions; incrementally maintaining fix is difficult, so we treat it as non-monotone.

[4] An alternative to box pattern-matching and split, pursued in Arntzenius and Krishnaswami (2016), would be to give two rules for **case**, depending on whether or not the scrutinee can be typechecked in a stripped context.

2.3.2 The category Poset and its structures

An object of **Poset** is a pair $(A, \leqslant_A)$ consisting of a set A and a reflexive, transitive, antisymmetric relation $\leqslant_A \subseteq A \times A$. For convenience, we usually denote these by a single letter A, leaving $\leqslant_A$ implicit. Following this convention, a morphism $f : A \to B$ is a function such that $x \leqslant_A y \implies f(x) \leqslant_B f(y)$.

Bicartesian structure

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

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

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

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

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

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

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

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

The discreteness comonad

Given a poset $(A, \leqslant_A)$ we define the discreteness comonad $\Box(A, \leqslant_A)$ as $(A, \leqslant_{\Box A})$, where $a \leqslant_{\Box A} a' \iff a = a'$. That is, the discrete order preserves the underlying elements, but reduces the partial order to mere equality. This forms a rather boring comonad whose functorial action $\Box(f)$, extraction $\varepsilon_A : \Box A \to A$, and duplication $\delta_A : \Box A \to \Box\Box A$ are all identities on the underlying sets:

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

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

Sets and semilattices

Given a poset $(A, \leqslant_A)$ we define the finite powerset poset $\mathsf{P}(A, \leqslant_A)$ as $(\mathcal{P}_{\mathsf{fin}}\, A, \subseteq)$, that is, the finite subsets of A ordered by inclusion.[5] Finite sets admit a pair of useful morphisms:

$$\mathsf{singleton} \;:\; \Box A \to \mathsf{P}A \qquad\qquad \mathsf{isEmpty} \;:\; \Box \mathsf{P}A \to 1 + 1$$

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

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

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

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

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

Equality

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

$$\mathsf{eq} \;:\; \Box A \times \Box A \to \mathsf{P}1$$

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

The domain must be discrete, since $x = y$ and $y \leqslant z$ certainly doesn't imply $x = z$. Although in principle every object $A \in \mathbf{Poset}$ admits eq, in practice our semantics only uses it when equality is decidable.

Fixed points

Given a semilattice $L \in \mathbf{Poset}$ without infinite ascending chains, we can define a family of fixed point morphisms $\mathsf{fix} : \Box(L \Rightarrow L) \to L$ as follows:[6]

---

[5] Note that the subset ordering completely ignores the element ordering $\leqslant_A$. There are orderings on $\mathcal{P}_{\mathsf{fin}}\, A$ which are not so forgetful; for instance, the free semilattice FA consists of finite sets ordered by $s \leqslant_{FA} t \iff (\forall x \in s, \exists y \in t)\; x \leqslant_A y$ and quotiented by antisymmetry (this may also be seen as the semilattice of finitely-generated downward closed sets under union, or of finite antichains). Our $\mathsf{P}A$ is isomorphic to $F\Box A$. However, Datalog's semantics use only the inclusion order, as do all of our motivating examples; so for simplicity we have stuck to it.

[6] In fact, fix is monotone and could be regarded as a map $(L \Rightarrow L) \to L$, but because the typing rule for fix boxes its argument, we do the same here.

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

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

2.3.3 Interpretation of Datafun in Poset

Figure 2.4 shows how to interpret Datafun into **Poset** using the structures developed above. We interpret Datafun types and typing contexts as **Poset**-objects $[\![A]\!]$, $[\![\Gamma]\!]$ and well-typed Datafun terms (or more precisely, their typing derivations) $\Gamma \vdash e : A$ as **Poset**-morphisms $[\![\Gamma]\!] \to [\![A]\!]$. This follows the usual interpretation for constructive S4 (Alechina et al., 2001), with the addition of sets, semilattices, fixed points, and the ability to distribute $\Box$ over sums. We give the interpretation in combinatory style; to increase readability, we freely use n-ary products to represent our typing context, to avoid the book-keeping of reassociating binary products.

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

2.4 Operational semantics

We consider the denotational semantics to be primary in Datafun; as with Datalog, any implementation technique is valid so long as it lines up with this semantics. To show such an implementation is possible, we present a simple call-by-value structural operational semantics in figure 2.5 and show that all well-typed terms terminate. In our operational semantics we:

1. Drop the distinction between discrete and monotone variables, writing both in lowercase x, y, z, and cease using a light blue background for non-monotone expressions.

2. Assume all equality tests and all semilattice operations ($\bot$, $\vee$, **for**, and fix) are subscripted with their type.

3. Add iter expressions, which occur as intermediate forms in the evaluation of fix.

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

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

$$\textsc{eval context}\quad \frac{e \longmapsto e'}{E\{\bigcirc \mapsto e\} \longmapsto E\{\bigcirc \mapsto e'\}}$$

TYPE AND CONTEXT DENOTATIONS

$$\llbracket 1 \rrbracket = 1 \qquad \llbracket A \to B \rrbracket = \llbracket A \rrbracket \Rightarrow \llbracket B \rrbracket$$
$$\llbracket \{\underset{EQ}{A}\} \rrbracket = P\llbracket \underset{EQ}{A} \rrbracket \qquad \llbracket A \times B \rrbracket = \llbracket A \rrbracket \times \llbracket B \rrbracket$$
$$\llbracket \Box A \rrbracket = \Box \llbracket A \rrbracket \qquad \llbracket A + B \rrbracket = \llbracket A \rrbracket + \llbracket B \rrbracket$$

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

TERM DENOTATIONS

$$\begin{aligned}
\llbracket \Gamma \vdash X : A \rrbracket &= \pi_X \qquad (\text{for } X : A \in \Gamma) \\
\llbracket \Gamma \vdash x : A \rrbracket &= \pi_x \mathbin{;} \varepsilon \qquad (\text{for } x :: A \in \Gamma) \\
\llbracket \Gamma \vdash \lambda X.\, e : A \to B \rrbracket &= \lambda_X \llbracket \Gamma, X : A \vdash e : B \rrbracket \\
\llbracket \Gamma \vdash e_1\ e_2 : B \rrbracket &= \langle \llbracket \Gamma \vdash e_1 : A \to B \rrbracket, \llbracket \Gamma \vdash e_2 : A \rrbracket \rangle \mathbin{;} \mathsf{eval} \\
\llbracket \Gamma \vdash (e_1, e_2) : A_1 \times A_2 \rrbracket &= \langle \llbracket \Gamma \vdash e_1 : A_1 \rrbracket, \llbracket \Gamma \vdash e_2 : A_2 \rrbracket \rangle \\
\llbracket \Gamma \vdash \pi_i\ e : A_i \rrbracket &= \llbracket \Gamma \vdash e : A_1 \times A_2 \rrbracket \mathbin{;} \pi_i \\
\llbracket \Gamma \vdash \mathsf{in}_i\ e : A_1 + A_2 \rrbracket &= \llbracket \Gamma \vdash e : A_i \rrbracket \mathbin{;} \mathsf{in}_i \\
\llbracket \Gamma \vdash \textbf{case } e \textbf{ of } (\mathsf{in}_i\ X_i \to f_i)_i : B \rrbracket &= \langle \mathsf{id}_{\llbracket \Gamma \rrbracket}, \llbracket \Gamma \vdash e : A_1 + A_2 \rrbracket \rangle \\
&\quad \mathbin{;} \mathsf{dist}^{\times}_{+} \\
&\quad \mathbin{;} \left[\llbracket \Gamma, X_i : A_i \vdash f_i : B \rrbracket \right]_{i \in \{1,2\}} \\
\llbracket \Gamma \vdash [e] : \Box A \rrbracket &= \mathsf{box}_\Gamma(\llbracket \lceil \Gamma \rceil \vdash e : A \rrbracket) \\
\llbracket \Gamma \vdash \textbf{let } [x] = e \textbf{ in } f : B \rrbracket &= \langle \mathsf{id}_{\llbracket \Gamma \rrbracket}, \llbracket \Gamma \vdash e : \Box A \rrbracket \rangle \mathbin{;} \llbracket \Gamma, x :: A \vdash f : B \rrbracket \\
\llbracket \Gamma \vdash \bot : L \rrbracket &= \langle\rangle \mathbin{;} \mathsf{join}_0^{\llbracket L \rrbracket} \\
\llbracket \Gamma \vdash e \vee f : L \rrbracket &= \langle \llbracket \Gamma \vdash e : L \rrbracket, \llbracket \Gamma \vdash f : L \rrbracket \rangle \mathbin{;} \mathsf{join}_2^{\llbracket L \rrbracket} \\
\llbracket \Gamma \vdash \mathsf{empty?}\ e : 1 + 1 \rrbracket &= \mathsf{box}_\Gamma(\llbracket \lceil \Gamma \rceil \vdash e : \{1\} \rrbracket) \mathbin{;} \mathsf{isEmpty} \\
\llbracket \Gamma \vdash \mathsf{split}\ e : \Box A + \Box B \rrbracket &= \llbracket \Gamma \vdash e : \Box(A + B) \rrbracket \mathbin{;} \mathsf{dist}^{\Box}_{+} \\
\llbracket \Gamma \vdash e_1 = e_2 : \mathsf{bool} \rrbracket &= \langle \mathsf{box}_\Gamma(\llbracket \lceil \Gamma \rceil \vdash e_1 : \underset{EQ}{A} \rrbracket), \mathsf{box}_\Gamma(\llbracket \lceil \Gamma \rceil \vdash e_2 : \underset{EQ}{A} \rrbracket) \rangle \mathbin{;} \mathsf{eq} \\
\llbracket \Gamma \vdash \mathsf{fix}\ e : \underset{FIX}{L} \rrbracket &= \llbracket \Gamma \vdash e : \Box(\underset{FIX}{L} \to \underset{FIX}{L}) \rrbracket \mathbin{;} \mathsf{fix} \\
\llbracket \Gamma \vdash \{e_i\}_i : \{\underset{EQ}{A}\} \rrbracket &= \langle \mathsf{box}_\Gamma(\llbracket \lceil \Gamma \rceil \vdash e_i : \underset{EQ}{A} \rrbracket) \mathbin{;} \mathsf{singleton} \rangle_i \mathbin{;} \mathsf{join}^{P\llbracket \underset{EQ}{A} \rrbracket} \\
\llbracket \Gamma \vdash \textbf{for } (x \in e)\ f : L \rrbracket &= \langle \mathsf{id}_{\llbracket \Gamma \rrbracket}, \llbracket \Gamma \vdash e : \{\underset{EQ}{A}\} \rrbracket \rangle \mathbin{;} \mathsf{collect}(\llbracket \Gamma, x :: \underset{EQ}{A} \vdash f : L \rrbracket)
\end{aligned}$$

AUXILLIARY OPERATIONS

$$\mathsf{dist}^{\times}_{+} : A \times (B_1 + B_2) \to (A \times B_1) + (A \times B_2) \qquad \mathsf{box}_\Gamma : \llbracket \lceil \Gamma \rceil \vdash A \rrbracket \to \llbracket \Gamma \vdash \Box A \rrbracket$$
$$\mathsf{dist}^{\times}_{+} = \langle \pi_2 \mathbin{;} [\lambda(\langle \pi_2, \pi_1 \rangle \mathbin{;} \mathsf{in}_i)]_i, \pi_1 \rangle \mathbin{;} \mathsf{eval} \qquad \mathsf{box}_\Gamma(f) = \langle \pi_x \mathbin{;} \delta \rangle_{x :: A \in \Gamma} \mathbin{;} \mathsf{dist}^{\times}_{\Box} \mathbin{;} \Box(f)$$

FIGURE 2.4 *Semantics of Datafun*

ADDITIONAL SYNTAX

$$
\begin{array}{rrcl}
\text{expressions} & e, f, g & ::= & \ldots \mid e =_{\underset{\text{EQ}}{A}} f \mid \bot_L \mid e \vee_L f \mid \mathbf{for}_L\ (x \in e)\ f \mid \mathsf{fix}_{\underset{\text{FIX}}{L}}\ e \\
& & & \mathsf{iter}_{\underset{\text{EQ}}{A}}(v,\ e,\ f) \\
\text{values} & v, u & ::= & \lambda x.\ e \mid () \mid (v, u) \mid \mathsf{in}_i\ v \mid \{v_i\}_i \mid [v] \\
\text{evaluation contexts} & E & ::= & \bigcirc \mid E\ e \mid v\ E \mid (E, e) \mid (v, E) \mid \pi_i\ E \\
& & & \mathsf{in}_i\ E \mid \mathbf{case}\ E\ \mathbf{of}\ (\mathsf{in}_i\ x_i \to e_i)_i \\
& & & [E] \mid \mathbf{let}\ [x] = E\ \mathbf{in}\ e \\
& & & \{\vec{v}, E, \vec{e}\} \mid E \vee_L e \mid v \vee_L E \mid \mathbf{for}_L\ (x \in E)\ e \\
& & & E =_A e \mid v =_A E \mid \mathsf{split}\ E \mid \mathsf{empty?}\ E \\
& & & \mathsf{fix}_{\underset{\text{FIX}}{L}}\ E \mid \mathsf{iter}_A(v,\ E,\ f) \mid \mathsf{iter}_A(v,\ u,\ E)
\end{array}
$$

VALUE (IN)EQUALITY

$$
() \leqslant () : 1 \qquad \frac{v_1 \leqslant u_1 : \underset{\text{EQ}}{A} \qquad v_2 \leqslant u_2 : \underset{\text{EQ}}{B}}{(v_1, v_2) \leqslant (u_1, u_2) : \underset{\text{EQ}}{A} \times \underset{\text{EQ}}{B}} \qquad \frac{v \leqslant u : A_i}{\mathsf{in}_i\ v \leqslant \mathsf{in}_i\ u : A_1 + A_2}
$$

$$
\frac{(\forall i, \exists j)\ v_i = u_j : \underset{\text{EQ}}{A}}{\{v_i\}_i \leqslant \{u_j\}_j : \{\underset{\text{EQ}}{A}\}} \qquad \frac{v \leqslant u : \underset{\text{EQ}}{A} \qquad u \leqslant v : \underset{\text{EQ}}{A}}{v = u : \underset{\text{EQ}}{A}}
$$

β REDUCTIONS

$$
\begin{array}{rcl@{\qquad}rcl}
(\lambda x.\ e)\ v & \longmapsto & e\{x \mapsto v\} & \mathbf{let}\ [x] = [v]\ \mathbf{in}\ e & \longmapsto & e\{x \mapsto v\} \\
\pi_i\ (v_1, v_2) & \longmapsto & v_i & \mathbf{case}\ \mathsf{in}_i\ v\ \mathbf{of}\ (\mathsf{in}_j\ x_j \to e_j)_j & \longmapsto & e_i\{x_i \mapsto v\} \\
\mathbf{for}_L\ (x \in \{\})\ e & \longmapsto & \bot_L & & & \\
\mathbf{for}_L\ (x \in \{\vec{u}, v\})\ e & \longmapsto & (\mathbf{for}_L\ (x \in \{\vec{u}\})\ e) \vee_L (e\{x \mapsto v\}) & & &
\end{array}
$$

OTHER REDUCTIONS

$$
\begin{array}{rcl@{\qquad}rcl}
\bot_1 & \longmapsto & () & () \vee_1 () & \longmapsto & () \\
\bot_{\{A\}} & \longmapsto & \{\} & \{\vec{v}\} \vee_{\{A\}} \{\vec{u}\} & \longmapsto & \{\vec{v}, \vec{u}\} \\
\bot_{L \times M} & \longmapsto & (\bot_L, \bot_M) & (v_1, v_2) \vee_{L \times M} (u_1, u_2) & \longmapsto & (v_1 \vee_L u_1, v_2 \vee_M u_2) \\
\mathsf{empty?}\ \{\} & \longmapsto & \mathsf{in}_1\ () & \mathsf{empty?}\ \{v, \vec{u}\} & \longmapsto & \mathsf{in}_2\ () \\
\mathsf{split}\ [\mathsf{in}_i\ v] & \longmapsto & \mathsf{in}_i\ [v] & v =_{\underset{\text{EQ}}{A}} u & \longmapsto & \begin{cases} \mathsf{true} & \text{if } v = u : \underset{\text{EQ}}{A} \\ \mathsf{false} & \text{otherwise} \end{cases} \\
\mathsf{fix}_{\underset{\text{FIX}}{L}}\ [v] & \longmapsto & \mathsf{iter}_{\underset{\text{FIX}}{L}}(v,\ \bot_{\underset{\text{FIX}}{L}},\ v\ \bot_{\underset{\text{FIX}}{L}}) & & & \\
\mathsf{iter}_{\underset{\text{EQ}}{A}}(v,\ u_1,\ u_2) & \longmapsto & \begin{cases} u_1 & \text{if } u_1 = u_2 : \underset{\text{EQ}}{A} \\ \mathsf{iter}_{\underset{\text{EQ}}{A}}(v,\ u_2,\ v\ u_2) & \text{otherwise} \end{cases} & & &
\end{array}
$$

FIGURE 2.5 *Operational semantics*

In our rules for $e \longmapsto e'$ where e is an iter expression we make use of a decidable ordering test on values, $v \leqslant u : \underset{\text{EQ}}{A}$, and a corresponding equality test $v = u : \underset{\text{EQ}}{A}$. We define these using inference rules, but they are easily seen to be decidable by induction on $\underset{\text{EQ}}{A}$.

Our implementation strategy for fix f is straightforward: starting from $\bot$, iteratively apply f until quiescence. We introduce the form $\mathsf{iter}(f, e_1, e_2)$ to represent these intermediate iterative steps. The intention is that e_1, e_2 shall be successive iterations of f, with $e_2 = f\ e_1$. The fixed point expression fix f, after evaluating f, steps to $\mathsf{iter}(f, \bot, f\ e)$, which kicks off the first two iterations. Once these have reduced to values, $\mathsf{iter}(f, u_1, u_2)$ tests $u_1 = u_2$ to determine if a fixed point has been reached. If so, its value u_1 is returned; otherwise we step to $\mathsf{iter}(f, u_2, f\ u_2)$ to evaluate the next iteration, and so on.

Observe that values don't step and evaluation is deterministic:

Lemma 1 (Values don't step)**.** If v is a value, there is no e such that $v \longmapsto e$.

Proof. The left hand side of each reduction rule can never be a value. This is easily verified by inspection for the rules in figure 2.5; and for EVAL CONTEXT we can see by the definition of evaluation contexts E that filling a hole with a non-value always produces a non-value. □

Lemma 2 (Determinism)**.** If $e \longmapsto e_1'$ and $e \longmapsto e_2'$ then $e_1' = e_2'$; thus inductively, since values don't step, if $e \longmapsto^* v$ and $e \longmapsto^* u$ then $v = u$.

Proof. The left-hand sides of all reduction rules $e \longmapsto e'$ are disjoint; there is no term to which two distinct reduction rules could apply. This applies inductively to EVAL CONTEXT because decompositions of a term into an evaluation context and a reducible subterm are unique. □

2.4.1 A logical relation for termination

To prove that all well-typed terms terminate according to our operational semantics, we use a logical relations argument. As a matter of notation, we will let v, u, w range over values; a, b, c range over closed terms; and γ, σ range over closing substitutions.

Our guiding intuition is that since we need an order structure in our denotational semantics to prove the definedness of fixed points, we likewise need an order structure on our syntax to prove the termination of fixed points. To this end we interpret each type A as a *partial preorder,* $x \prec y : A$. A partial preorder is a relation which is transitive and *partially reflexive,* that is, $x \prec y \implies x \prec x \wedge y \prec y$. While reflexivity may be glossed as "every element is related to itself", partial reflexivity glosses as "if an element is related to anything, it is related to itself"; in other words, unlike reflexivity, it permits some elements to be "outside the relation" and unrelated to anything, even themselves. Any partial preorder $x \prec y$ gives rise to a symmetric, transitive relation $x \equiv y \iff x \prec y \wedge y \prec x$.[7]

In fact we define a mutually inductive collection of partial preorders: on values $v \prec u : A$, an extension to closed terms $a \prec b : A$, on closing substitutions $\gamma \prec \sigma : \Gamma$, on open terms

[7] Symmetric, transitive relations are also known as partial equivalence relations (PERs). Moreover, letting $[x]_\equiv$ denote the equivalence class of x (defined only when $x \prec x$), the relation $[x]_\equiv \leqslant [y]_\equiv \iff x \prec y$ is a partial order over these equivalence classes; so our approach may also be considered to interpret types as PERs equipped with partial orders on their equivalence classes.

$e \prec f : \Gamma \vdash A$, and on open terms paired with closing substitutions $\gamma_1, e_1 \prec \gamma_2, e_2 : \Gamma, A$. The rules for the value-relation are:

$$\frac{}{() \prec () : 1} \qquad \frac{v_1 \prec u_1 : A \qquad v_2 \prec u_2 : B}{(v_1, v_2) \prec (u_1, u_2) : A \times B} \qquad \frac{v \prec u : A_i}{\mathsf{in}_i\, v \prec \mathsf{in}_i\, u : A_1 + A_2}$$

$$\textsc{lr fn}\ \frac{e \prec f : (X : A \vdash B)}{\lambda X.\, e \prec \lambda X.\, f : A \to B} \qquad \frac{v \equiv u : A}{[v] \prec [u] : \Box A} \qquad \textsc{lr set}\ \frac{(\forall i, \exists j)\ v_i \equiv u_j : \underset{EQ}{A} \qquad (\forall j)\ u_j \prec u_j : \underset{EQ}{A}}{\{v_i\}_i \prec \{u_i\}_i : \{\underset{EQ}{A}\}}$$

Note that LR FN depends on the relation for open terms, making this definition mutually inductive. The second premise of LR SET may seem strange but it is necessary to ensure partial reflexivity. We extend this value-relation to closed terms:

$$a \prec b : A \iff (\exists v, u)\ a \longmapsto^* v \wedge b \longmapsto^* u \wedge v \prec u : A$$

Note that if a, b are values, this definition coincides with the relation on values, since values do not step; this justifies using the same notation for the relation on values and closed terms. We extend this relation to closing substitutions pointwise, noting that discrete hypotheses are required to be equivalent:

$$\gamma \prec \sigma : \Gamma \iff ((\forall X : A \in \Gamma)\ \gamma_x \prec \sigma_x : A) \wedge ((\forall x :: A \in \Gamma)\ \gamma_x \equiv \sigma_x : A)$$

Finally, we extend the relation to open terms, which involves an auxiliary relation on pairs of terms and closing substitutions:

$$\begin{aligned} e_1 \prec e_2 : \Gamma \vdash A &\iff (\forall \gamma_1 \prec \gamma_2 : \Gamma)\ \gamma_1, e_1 \prec \gamma_2, e_2 : \Gamma, A \\ \gamma_1, e_1 \prec \gamma_2, e_2 : \Gamma, A &\iff (\forall i = 1, 2)\ \gamma_i(e_1) \prec \gamma_i(e_2) : A \wedge \gamma_1(e_i) \prec \gamma_2(e_i) : A \end{aligned}$$

Note that $\gamma_1, e_1 \prec \gamma_2, e_2 : \Gamma, A$ may be seen as a transitive square:

$$\begin{array}{ccc} \gamma_1(e_1) & \prec & \gamma_1(e_2) \\ \curlywedge & & \curlywedge \\ \gamma_2(e_1) & \prec & \gamma_2(e_2) \end{array}$$

This ensures partial reflexivity; if we replace e_1 with e_2 or vice-versa this square collapses to one of its sides. If we had instead only required the diagonal, $\gamma_1(e_1) \prec \gamma_2(e_2) : A$, we could not derive $\gamma_1(e_1) \prec \gamma_2(e_1) : A$ (or the same for e_2) as required by partial reflexivity.

Theorem 3 (Fundamental theorem)**.** If $\Gamma \vdash e : A$ then $e \prec e : \Gamma \vdash A$.

Termination of well-typed programs follows as a corollary by unrolling definitions:

Theorem 4 (Termination)**.** Every closed, well-typed program $\varepsilon \vdash a : A$ terminates.

Proof.

$$
\begin{array}{ll}
\quad \varepsilon \vdash a : A & \\
\Longrightarrow a \prec a : \varepsilon \vdash A & \text{Fundamental theorem} \\
\Longrightarrow (\forall \gamma_1 \prec \gamma_2 : \varepsilon)\ \gamma_1, a \prec \gamma_2, a : \Gamma, A & \text{expand the definition} \\
\Longrightarrow (), a \prec (), a : \varepsilon, A & \text{since } () \prec () : \varepsilon \text{ vacuously} \\
\Longrightarrow a \prec a : A & \text{expand the definition and simplify} \\
\Longrightarrow (\exists v, u)\ a \longmapsto^* v \wedge a \longmapsto^* u \wedge v \prec u : A & \text{expand the definition} \\
\Longrightarrow (\exists v)\ a \longmapsto^* v & \text{simplify}
\end{array}
$$

□

The proof of the fundamental theorem itself proceeds by induction on $\Gamma \vdash e : A$. The key case is the fixed point rule, whose proof is a syntactic version of the proof of the existence of least fixed points in the denotational semantics. We give the proof of the fundamental theorem at the end of this section; to build up to it we must first develop several auxiliary definitions and lemmas.

2.4.2 Metatheory of the logical relation

First, any partial preorder over a set S gives rise to a subset $\{x \in S \mid x \prec x\}$ over which $\prec$ is reflexive and thus a true preorder. It will be convenient to apply this point of view to our logical relations:

Definition 5 (Good terms)**.** We define the following preordered sets of "good" terms:

$$
\begin{array}{rll}
\mathsf{Ok}_V(A) = \{v \mid v \prec v : A\} & & \text{the good values} \\
\mathsf{Ok}_C(A) = \{a \mid a \prec a : A\} & & \text{the good closed terms} \\
\mathsf{Ok}(\Gamma \vdash A) = \{e \mid e \prec e : \Gamma \vdash A\} & & \text{the good open terms}
\end{array}
$$

We preorder these by the corresponding logical relation, so $v \leqslant u : \mathsf{Ok}_V(A) \iff v \prec u : A$, etc.

Lemma 6 (Closed term evaluation map)**.** There exists a monotone map $\mathsf{value}_A : \mathsf{Ok}_C(A) \to \mathsf{Ok}_V(A)$ such that $\mathsf{value}\ a = v$ if and only if $a \longmapsto^* v$.

Proof. The definition of $a \prec a : A$ shows that every good closed term evaluates to some good value. Determinism shows this value is unique, so we can name it $\mathsf{value}\ a$. And given $a \prec b : A$, applying its definition and this uniqueness shows that $\mathsf{value}\ a \prec \mathsf{value}\ b : A$, showing monotonicity. □

Lemma 7 (Closed term application map)**.** If $a_1 \prec a_2 : A \to B$ and $b_1 \prec b_2 : A$ then $a_1\ b_1 \prec a_2\ b_2 : B$. Equivalently, there exists a monotone map $\mathsf{apply}_{A,B} : \mathsf{Ok}_C(A \to B) \times \mathsf{Ok}_C(A) \to \mathsf{Ok}_V(B)$ such that $\mathsf{apply}\ (a, b) = \mathsf{value}\ (a\ b) = v$ if and only if $a\ b \longmapsto^* v$.

Proof. Suppose $a_1 \prec a_2 : A \to B$ and $b_1 \prec b_2 : A$. Unrolling these assumptions, we have e_1, e_2, u_1, u_2 satisfying:

$$
\begin{array}{lll}
a_1 \longmapsto^* \lambda X.\ e_1 & a_2 \longmapsto^* \lambda X.\ e_2 & e_1 \prec e_2 : (X : A) \vdash B \\
b_1 \longmapsto^* u_1 & b_2 \longmapsto^* u_2 & u_1 \prec u_2 : A
\end{array}
$$

From $u_1 \prec u_2 : A$ we have $(X \mapsto u_1) \prec (X \mapsto u_2) : (X : A)$, and applying this to $e_1 \prec e_2 : (X : A) \vdash B$ we have a transitive square:

$$\begin{array}{ccc} e_1\{X \mapsto u_1\} & \prec & e_2\{X \mapsto u_1\} \\ \curlywedge & & \curlywedge \\ e_1\{X \mapsto u_2\} & \prec & e_2\{X \mapsto u_2\} \end{array}$$

Taking the diagonal of this square, we have $e_1\{X \mapsto u_1\} \prec e_2\{X \mapsto u_2\} : B$ and therefore v_1, v_2 such that $e_i\{X \mapsto u_i\} \longmapsto^* v_i$ and $v_1 \prec v_2 : B$. Thus for $i \in \{1, 2\}$ we have:

$$a_i\ b_i \longmapsto^* (\lambda X.\ e_i)\ b_i \longmapsto^* (\lambda X.\ e_i)\ u_i \longmapsto e_i\{X \mapsto u_i\} \longmapsto^* v_i$$

and $v_1 \prec v_2 : B$ as desired. □

Lemma 8 (Closed term pairing)**.** If $a_1 \prec a_2 : A$ and $b_1 \prec b_2 : B$ then $(a_1, b_1) \prec (a_2, b_2) : A \times B$.

Proof. Applying our assumptions' definitions we have v_1, v_2, u_1, u_2 such that:

$$\begin{array}{lll} a_1 \longmapsto^* v_1 & a_2 \longmapsto^* v_2 & v_1 \prec v_2 : A \\ b_1 \longmapsto^* u_1 & b_2 \longmapsto^* u_2 & u_1 \prec u_2 : B \end{array}$$

From this we have:

$$(a_1, b_1) \longmapsto^* (v_1, b_1) \longmapsto^* (v_1, u_1) \qquad (a_2, b_2) \longmapsto^* (v_2, b_2) \longmapsto^* (v_2, u_2)$$

And $(v_1, u_1) \prec (v_2, u_2) : A \times B$ because $v_1 \prec v_2 : A$ and $u_1 \prec u_2 : B$, which is what we wished to show. □

Lemma 9 (Closure under stepping)**.** $\prec$ is closed under $\longmapsto^*$; that is, if $a \prec b : A$ and $(a' \longmapsto^* a) \vee (a \longmapsto^* a')$ and $(b' \longmapsto^* b) \vee (b \longmapsto^* b')$, then $a' \prec b' : A$.

Proof. $a \longmapsto^* v$ and $b \longmapsto^* u$ such that $v \prec u : A$ so by determinism $a' \longmapsto^* v$ and $b' \longmapsto^* u$, thus $a' \prec b' : A$. □

Lemma 10 (First-order agreement on values)**.** If $v \in \mathsf{Ok}_V(\underset{EQ}{A})$ then $\varepsilon \vdash v : \underset{EQ}{A}$ and moreover for $v, u \in \mathsf{Ok}_V(\underset{EQ}{A})$:

$$v \prec u : \underset{EQ}{A} \iff [\![v]\!] \leqslant [\![u]\!] : [\![\underset{EQ}{A}]\!] \iff v \leqslant u : \underset{EQ}{A}$$

(To be precise, by $[\![v]\!] : [\![\underset{EQ}{A}]\!]$ we mean the map $[\![\varepsilon \vdash v : \underset{EQ}{A}]\!] : \mathbf{Poset}([\![\varepsilon]\!], [\![\underset{EQ}{A}]\!])$ applied to the empty environment $() : [\![\varepsilon]\!]$.)

Proof. By induction on $\underset{EQ}{A}$. To show $\varepsilon \vdash v : \underset{EQ}{A}$, in each case we apply the definition of $v \prec v : \underset{EQ}{A}$ (e.g. for $\underset{EQ}{A_1} \times \underset{EQ}{A_2}$ we find that $v = (u_1, u_2)$ for some $u_1, u_2 \in \mathsf{Ok}_V(\underset{EQ}{A_1})$) followed by applying our inductive hypotheses and the following typing rules (here specialized to $\Gamma = \varepsilon$):

$$\textsc{unit}\ \frac{}{\varepsilon \vdash () : 1} \qquad \textsc{pair}\ \frac{(\varepsilon \vdash e_i : A_i)_i}{\varepsilon \vdash (e_1, e_2) : A_1 \times A_2} \qquad \textsc{inj}\ \frac{\varepsilon \vdash e : A_i}{\varepsilon \vdash \mathsf{in}_i\, e : A_1 + A_2} \qquad \textsc{set}\ \frac{(\varepsilon \vdash e_i : \underset{EQ}{A})_i}{\varepsilon \vdash \{e_i\}_i : \{\underset{EQ}{A}\}}$$

As for equivalence of the orderings, observe that:

$$[\![()]\!] = () : [\![1]\!] \qquad [\![(v,u)]\!] = ([\![v]\!],[\![u]\!]) : [\![\underset{EQ}{A} \times \underset{EQ}{B}]\!]$$
$$[\![\mathsf{in}_i\, v]\!] = \mathsf{in}_i\, [\![v]\!] : [\![\underset{EQ}{A} + \underset{EQ}{B}]\!] \qquad [\![\{v_i\}_i]\!] = \{[\![v_i]\!]\}_i : [\![\{\underset{EQ}{A}\}]\!]$$

and therefore:

$$\begin{aligned}
[\![()]\!] \leqslant [\![()]\!] : [\![1]\!] &\iff \top \\
[\![(v_1,u_1)]\!] \leqslant [\![(v_2,u_2)]\!] : [\![\underset{EQ}{A} \times \underset{EQ}{B}]\!] &\iff [\![v_1]\!] \leqslant [\![v_2]\!] : [\![\underset{EQ}{A}]\!] \wedge [\![u_1]\!] \leqslant [\![u_2]\!] : [\![\underset{EQ}{B}]\!] \\
[\![\mathsf{in}_i\, v]\!] \leqslant [\![\mathsf{in}_j\, u]\!] : [\![\underset{EQ}{A}_1 + \underset{EQ}{A}_2]\!] &\iff i = j \wedge [\![v]\!] \leqslant [\![u]\!] : [\![\underset{EQ}{A}_i]\!] \\
[\![\{v_i\}_i]\!] \leqslant [\![\{u_j\}_j]\!] : [\![\{\underset{EQ}{A}\}]\!] &\iff \{[\![v_i]\!]\}_i \subseteq \{[\![u_j]\!]\}_j \iff (\forall i, \exists j)\ [\![v_i]\!] = [\![u_j]\!] : [\![\underset{EQ}{A}]\!]
\end{aligned}$$

In each case, these coincide (after applying our inductive hypothesis) with the rules defining $v \prec u : \underset{EQ}{A}$ and $v \leqslant u : \underset{EQ}{A}$, except for $\{v_i\}_i \prec \{u_j\}_j : \{\underset{EQ}{A}\}$, which has the additional premise $(\forall j)\ u_j \prec u_j : \underset{EQ}{A}$; but this is satisfied by the assumption $\{u_j\}_j \in \mathsf{Ok}_V(\{\underset{EQ}{A}\})$. Thus inductively all three preorders coincide on good values of equality types.

□

Lemma 11 (Bottom is bottom)**.** $\bot_L \in \mathsf{Ok}_C(L)$ and $(\forall a \in \mathsf{Ok}_C(L))\ \bot_L \prec a : L$.

Proof. By induction on L:

Case 1*:* Follows trivially from $\bot_1 \longmapsto ()$ and the relation at 1.

Case $L_1 \times L_2$*:* We have $a \longmapsto^* (v_1, v_2)$ for good v_i and by IH we have $\bot_{L_i} \prec v_i : L_i$. Thus by closed term pairing $(\bot_{L_1}, \bot_{L_2}) \prec (v_1, v_2) : L_1 \times L_2$ and since $\bot_{L_1 \times L_2} \longmapsto (\bot_{L_1}, \bot_{L_2})$ by closure under stepping we have what we desire.

Case $\{\underset{EQ}{A}\}$*:* Since $\bot_{\{\underset{EQ}{A}\}} \longmapsto \{\}$ it suffices to show $\{\} \prec v : \{\underset{EQ}{A}\}$ for $v \in \mathsf{Ok}_V(\{\underset{EQ}{A}\})$. This follows from LR SET; the first premise is vacuous and the second follows from goodness of v.

□

Lemma 12 (Join is join)**.** $a \vee_L b$ is the least upper bound of $a, b \in \mathsf{Ok}_C(L)$ with respect to the logical relation. That is, $a \prec a \vee_L b : L$ and $b \prec a \vee_L b : L$ and for any c such that $a \prec c : L$ and $b \prec c : L$ we have $a \vee_L b \prec c : L$.

Proof. By closure under stepping it suffices to show the same for only good values. For this it suffices to show that $[\![\mathsf{value}\ (v \vee_L u)]\!] = [\![v]\!] \vee [\![u]\!]$ because by lemma 10 the semantic ordering and the logical relation ordering agree, so a least upper bound in one is a least upper bound in the other. We show this by induction on L:

Case 1*:* Follows from $() \vee_L () \longmapsto^* ()$ and the trivial order on 1.

Case $L \times M$*:* Then we have

$$
\begin{aligned}
&[\![\mathsf{value}\ ((v_1, v_2) \vee_{L \times M} (u_1, u_2))]\!] \\
&= [\![(\mathsf{value}\ (v_1 \vee_L u_1), \mathsf{value}\ (v_2 \vee_M u_2))]\!] && \text{calculation} \\
&= ([\![\mathsf{value}\ (v_1 \vee_L u_1)]\!], [\![\mathsf{value}\ (v_2 \vee_M u_2)]\!]) && \text{calculation} \\
&= ([\![v_1]\!] \vee [\![u_1]\!], [\![v_2]\!] \vee [\![u_2]\!]) && \text{inductive hypothesis} \\
&= ([\![v_1]\!], [\![v_2]\!]) \vee ([\![u_1]\!], [\![u_2]\!]) && \text{join in product semilattice} \\
&= [\![(v_1, v_2)]\!] \vee [\![(u_1, u_2)]\!] && \text{calculation}
\end{aligned}
$$

Case $\{\underset{\mathsf{EQ}}{A}\}$: We have

$$
\begin{aligned}
[\![\mathsf{value}\ (\{\vec{v}\} \vee_{\{\underset{\mathsf{EQ}}{A}\}} \{\vec{u}\})]\!] &= [\![\{\vec{v}, \vec{u}\}]\!] && \text{calculation} \\
&= \{[\![v_i]\!]\}_i \cup \{[\![u_j]\!]\}_j && \text{calculation} \\
&= [\![\{\vec{v}\}]\!] \vee [\![\{\vec{u}\}]\!] && \text{calculation}
\end{aligned}
$$

□

Lemma 13 (Discrete contexts make terms equivalent)**.** If $e \prec f : \lceil \Gamma \rceil \vdash A$ and $\gamma_1 \prec \gamma_2 : \Gamma$ then $\gamma_1(e) \equiv \gamma_2(f) : A$.

Proof. From $\gamma_1 \prec \gamma_2 : \Gamma$ we have $\gamma_1 \equiv \gamma_2 : \lceil \Gamma \rceil$ because $\lceil \Gamma \rceil$ restricts to only discrete hypotheses $x :: A \in \Gamma$ for which we know $\gamma_1(x) \equiv \gamma_2(x) : A$. Thus applying $e \prec f : \lceil \Gamma \rceil \vdash A$ we have $\gamma_1(e) \equiv \gamma_2(f) : A$ as desired. □

2.4.3 Proof of the fundamental theorem

We now have the groundwork to prove the fundamental theorem, starting with the crucial case of fixed point expressions fix e.

Theorem 3 (Fundamental theorem)**.** If $\Gamma \vdash e : A$ then $e \prec e : \Gamma \vdash A$.

Proof. Unrolling the definition of the logical relation, we may assume $\gamma_1 \prec \gamma_2 : \Gamma$ and wish to show from this that $\gamma_1(e) \prec \gamma_2(e) : A$. We do this by induction on $\Gamma \vdash e : A$.

Case $\dfrac{\Gamma \vdash e : \square(\underset{\mathsf{FIX}}{L} \to \underset{\mathsf{FIX}}{L})}{\Gamma \vdash \mathit{fix}\ e : \underset{\mathsf{FIX}}{L}}$. We wish to show that

$$\mathsf{fix}_{\underset{\mathsf{FIX}}{L}}\ \gamma_1(e) \prec \mathsf{fix}_{\underset{\mathsf{FIX}}{L}}\ \gamma_2(e) : \underset{\mathsf{FIX}}{L}$$

By our inductive hypothesis we have $\gamma_1(e) \prec \gamma_2(e) : \square(\underset{\mathsf{FIX}}{L} \to \underset{\mathsf{FIX}}{L})$; applying this we have v_1, v_2 such that $\gamma_i(e) \longmapsto^* [v_i]$ and $v_1 \equiv v_2 : \underset{\mathsf{FIX}}{L} \to \underset{\mathsf{FIX}}{L}$. Thus for $i \in \{1, 2\}$:

$$\mathsf{fix}_{\underset{\mathsf{FIX}}{L}}\ \gamma_i(e) \longmapsto^* \mathsf{fix}_{\underset{\mathsf{FIX}}{L}}\ [v_i] \longmapsto \mathsf{iter}_{\underset{\mathsf{FIX}}{L}}(v_i,\ \bot_{\underset{\mathsf{FIX}}{L}},\ v_i\ \bot_{\underset{\mathsf{FIX}}{L}})$$

Let $f_i(u) = \mathsf{apply}\ (v_i, u)$ for brevity. Then applying lemmas 6 and 7 we have:

$$\mathsf{iter}_{\underset{\mathsf{FIX}}{L}}(v_i,\ \bot_{\underset{\mathsf{FIX}}{L}},\ v_i\ \bot_{\underset{\mathsf{FIX}}{L}}) \longmapsto^* \mathsf{iter}_{\underset{\mathsf{FIX}}{L}}(v_i,\ \mathsf{value}\ \bot_{\underset{\mathsf{FIX}}{L}},\ f_i(\mathsf{value}\ \bot_{\underset{\mathsf{FIX}}{L}}))$$

By bottom-is-bottom we have value $\bot_{\underset{\text{FIX}}{\text{L}}} \prec f_i(\bot_{\underset{\text{FIX}}{\text{L}}}) : \underset{\text{FIX}}{\text{L}}$. To understand the way evaluation will proceed from here, consider the generalized situation $\text{iter}_{\underset{\text{FIX}}{\text{L}}}(v_i,\, u,\, f_i(u))$ where $u \prec f_i(u) : \underset{\text{FIX}}{\text{L}}$. This steps like so:

$$\text{iter}_{\underset{\text{FIX}}{\text{L}}}(v_i,\, u,\, f_i(u)) \longmapsto^* \begin{cases} u & \text{if } u = f_i(u) : \underset{\text{EQ}}{\text{A}} \\ \text{iter}_{\underset{\text{EQ}}{\text{A}}}(v,\, f_i(u),\, f_i(f_i(u))) & \text{otherwise} \end{cases}$$

Starting with $u = \bot_{\underset{\text{FIX}}{\text{L}}}$, this calculates the sequence $u, f_i(u), f_i^2(u), f_i^3(u), \ldots$ until the first k such that $f_i^k(u) = f_i^{k+1}(u) : \underset{\text{FIX}}{\text{L}}$ and returns $f_i^k(u)$. Note that $f_i : \text{Ok}_C(\underset{\text{FIX}}{\text{L}}) \to \text{Ok}_V(\underset{\text{FIX}}{\text{L}})$ is monotone by lemma 7 and therefore this sequence ascends in the logical relation: $f_i^j(u) \prec f_i^{j+1}(u) : \underset{\text{FIX}}{\text{L}}$. Also by monotonicity of apply, since $v_1 \equiv v_2 : \underset{\text{FIX}}{\text{L}} \to \underset{\text{FIX}}{\text{L}}$ and $u \equiv u : \underset{\text{FIX}}{\text{L}}$ by partial reflexivity, we have inductively that $f_1^j(u) \equiv f_2^j(u) : \underset{\text{FIX}}{\text{L}}$. By lemma 10 these also hold in the semantic order, so the denotations both ascend $[\![f_i^j(u)]\!] \leqslant [\![f_i^{j+1}(u)]\!] : [\![\underset{\text{FIX}}{\text{L}}]\!]$ and coincide $[\![f_1^j(u)]\!] = [\![f_2^j(u)]\!] : [\![\underset{\text{FIX}}{\text{L}}]\!]$. By the ascending chain condition on $[\![\underset{\text{FIX}}{\text{L}}]\!]$ we know there must be some k such that $[\![f_1^k(u)]\!] = [\![f_1^{k+1}(u)]\!]$; and as the sequences for f_1, f_2 coincide, $[\![f_1^k(u)]\!] = [\![f_1^{k+1}(u)]\!] = [\![f_2^k(u)]\!] = [\![f_2^{k+1}(u)]\!]$. Applying lemma 10 again this shows $f_i^k(u) = f_i^{k+1}(u) : \underset{\text{FIX}}{\text{L}}$, and therefore $f_i^k(u)$ for the least such k is the value we terminate with; since $f_1^k(u) \equiv f_2^k(u) : \underset{\text{FIX}}{\text{L}}$, applying closure under stepping we are done.

Cases $\dfrac{X : A \in \Gamma}{\Gamma \vdash X : A} \quad \dfrac{x :: A \in \Gamma}{\Gamma \vdash x : A}$. Follows directly from $\gamma_1 \prec \gamma_2 : \Gamma$.

Case $\dfrac{\Gamma,\, X : A \vdash e : B}{\Gamma \vdash \lambda X.\, e : A \to B}$. What we wish to show is equivalent to:

$$\begin{aligned} & \gamma_1(\lambda X.\, e) \prec \gamma_2(\lambda X.\, e) : A \to B \\ \iff & \lambda X.\, \gamma_1(e) \prec \lambda X.\, \gamma_2(e) : A \to B \\ \iff & \gamma_1(e) \prec \gamma_2(e) : (X : A) \vdash B \end{aligned}$$

For the last it suffices to assume $a_1 \prec a_2 : A$ and show the transitive square at the logical relation for B:

$$\begin{array}{ccc} e\{\gamma_1,\, X \mapsto a_1\} & \prec & e\{\gamma_2,\, X \mapsto a_1\} \\ \curlywedge & & \curlywedge \\ e\{\gamma_1,\, X \mapsto a_2\} & \prec & e\{\gamma_2,\, X \mapsto a_2\} \end{array}$$

By our IH we have $(\forall \sigma \prec \sigma' : \Gamma,\, X : A)\ \sigma(e) \prec \sigma'(e) : B$, so it suffices to show the transitive square at the logical relation for $\Gamma, X : A$:

$$\begin{array}{ccc} (\gamma_1,\, X \mapsto a_1) & \prec & (\gamma_2,\, X \mapsto a_1) \\ \curlywedge & & \curlywedge \\ (\gamma_1,\, X \mapsto a_2) & \prec & (\gamma_2,\, X \mapsto a_2) \end{array}$$

This holds by $\gamma_1 \prec \gamma_2 : \Gamma$ and $a_1 \prec a_2 : A$ and the definition of the logical relation for contexts.

Case $\dfrac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash f : A}{\Gamma \vdash e\ f : B}$. We wish to show $\gamma_1(e)\ \gamma_1(f) \prec \gamma_2(e)\ \gamma_2(f) : B$. By IH we have $\gamma_1(e) \prec \gamma_2(e) : A \to B$ and $\gamma_1(f) \prec \gamma_2(f) : A$. What we wish to show then follows from lemma 7.

Case $\dfrac{}{\Gamma \vdash () : 1}$. Trivial.

Case $\dfrac{(\Gamma \vdash e_i : A_i)_i}{\Gamma \vdash (e_1, e_2) : A_1 \times A_2}$. Apply our inductive hypotheses to get $\gamma_i(e_j) \longmapsto^* v_{i,j}$ with $v_{1,j} \prec v_{2,j} : A_j$; this shows $(v_{1,1}, v_{1,2}) \prec (v_{2,1}, v_{2,2}) : A_1 \times A_2$ and since $\gamma_i((e_1, e_2)) = (\gamma_i(e_1), \gamma_i(e_2)) \longmapsto^* (v_{i,1}, \gamma_i(e_2)) \longmapsto^* (v_{i,1}, v_{i,2})$ by closure under stepping we are done.

Case $\dfrac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \pi_i\ e : A_i}$. By IH we have $\gamma_j(e) \longmapsto^* (v_{1,j}, v_{2,j})$ with $v_{i,1} \prec v_{i,2} : A_i$; thus we have $\gamma_j(\pi_i\ e) = \pi_i\ \gamma_j(e) \longmapsto^* \pi_i\ (v_{1,j}, v_{2,j}) \longmapsto v_{i,j}$ and we are done.

Case $\dfrac{\Gamma \vdash e : A_i}{\Gamma \vdash in_i\ e : A_1 + A_2}$. By IH we have $\gamma_j(e) \longmapsto^* v_j$ for some $v_1 \prec v_2 : A_i$. Applying the definition of the LR we have $\mathsf{in}_i\ v_1 \prec \mathsf{in}_i\ v_2 : A_1 + A_2$ and since $\gamma_j(\mathsf{in}_i\ e) = \mathsf{in}_i\ \gamma_j(e) \longmapsto^* \mathsf{in}_i\ v_j$ by closure under stepping we are done.

Case $\dfrac{\Gamma \vdash e : A_1 + A_2 \qquad (\Gamma, X_i : A_i \vdash f_i : B)_i}{\Gamma \vdash \textbf{\textit{case}}\ e\ \textbf{\textit{of}}\ (in_i\ X_i \to f_i)_i : B}$. By IH for e and the LR for $A_1 + A_2$ we have $\gamma_j(e) \longmapsto^* \mathsf{in}_i\ v_j$ for some i and $v_1 \prec v_2 : A_i$. Using this and our IH for f_i we have $f_i\{\gamma_1, X_i \mapsto v_1\} \prec f_i\{\gamma_2, X_i \mapsto v_2\} : B$. Then by calculation:

$$\begin{aligned}\gamma_j(\textbf{case}\ e\ \textbf{of}\ (\mathsf{in}_i\ X_i \to f_i)_i) &\longmapsto^* \textbf{case}\ \mathsf{in}_i\ v_j\ \textbf{of}\ (\mathsf{in}_i\ X_i \to \gamma_j(f_i))_i \\ &\longmapsto f_i\{\gamma_j, X_i \mapsto v_j\}\end{aligned}$$

and by closure under stepping we are done.

Case $\dfrac{\lceil\Gamma\rceil \vdash e : A}{\Gamma \vdash [e] : \Box A}$. By our IH and lemma 13 we have for some v_i that $\gamma_i(e) \longmapsto^* v_i$ with $v_1 \equiv v_2 : A$. Thus we have $\gamma_i([e]) = [\gamma_i(e)] \longmapsto^* [v_i]$ and by the definition of the LR for $\Box A$ we are done.

Case $\dfrac{\Gamma \vdash e : \Box A \qquad \Gamma, x :: A \vdash f : B}{\Gamma \vdash \textbf{\textit{let}}\ [x] = e\ \textbf{\textit{in}}\ f : B}$. By our IH for e and the logical relation for $\Box A$ we

have $\gamma_i(e) \longmapsto^* [v_i]$ for some $v_1 \equiv v_2 : A$. Then using this and applying our IH for f we have $f\{\gamma_i, x \mapsto v_i\} \longmapsto^* u_i$ for some $u_1 \prec u_2 : B$. Then we have:

$$\begin{aligned}\textbf{let } [x] = \gamma_i(e) \textbf{ in } \gamma_i(f) &\longmapsto^* \textbf{let } [x] = [v_i] \textbf{ in } \gamma_i(f) \\ &\longmapsto f\{\gamma_i, x \mapsto v_i\} \\ &\longmapsto u_i\end{aligned}$$

And by closure under stepping we are done.

Case $\dfrac{}{\Gamma \vdash \bot : L}$. By bottom is bottom.

Case $\dfrac{(\Gamma \vdash e_i : L)_i}{\Gamma \vdash e_1 \vee e_2 : L}$. Applying our IH we have $\gamma_1(e_i) \prec \gamma_2(e_i) : L$; since join is join we know in particular that $\vee$ is a monotone operator on good closed terms and therefore $\gamma_1(e_1) \vee \gamma_1(e_2) \prec \gamma_2(e_1) \vee \gamma_2(e_2) : L$ as desired.

Case $\dfrac{(\lceil\Gamma\rceil \vdash e_i : \underset{EQ}{A})_i}{\Gamma \vdash \{e_i\}_i : \{\underset{EQ}{A}\}}$. Applying lemma 13 we have by our IH that $\gamma_1(e_i) \equiv \gamma_2(e_i) : \underset{EQ}{A}$ and thus for some $v_{i,j}$ we have $\gamma_j(e_i) \longmapsto^* v_{i,j}$ with $v_{i,1} \equiv v_{i,2} : \underset{EQ}{A}$. Then by calculation we have $\gamma_j(\{e_i\}_i) \longmapsto^* \{v_{i,j}\}_i$ and applying LR SET (using partial reflexivity for its second premise) we have $\{v_{i,1}\}_j \prec \{v_{i,2}\}_i : \{\underset{EQ}{A}\}$ and by closure under stepping we are done.

Case $\dfrac{\Gamma \vdash e : \{\underset{EQ}{A}\} \qquad \Gamma, x :: \underset{EQ}{A} \vdash f : L}{\Gamma \vdash \textbf{\textit{for}}\ (x \in e)\ f : L}$. We wish to show that

$$\textbf{for } (x \in \gamma_1(e))\ \gamma_1(f) \prec \textbf{for } (x \in \gamma_2(e))\ \gamma_2(f) : L$$

By our IH we have $\gamma_1(e) \prec \gamma_2(e) : \{\underset{EQ}{A}\}$. Applying the definition of this, let $v_{i,j} \in \mathsf{Ok}_V(\underset{EQ}{A})$ and $a_{i,j}$ be defined by:

$$\{v_{i,j}\}_j = \mathsf{value}\ (\gamma_i(e)) \qquad\qquad a_{i,j} = f\{\gamma_i, x \mapsto v_{i,j}\}$$

From the logical relation we have $(\forall j, \exists k)\ v_{1,j} \prec v_{2,k} : \underset{EQ}{A}$. From this, our IH for f, and the definition of the logical relation for contexts we have $(\forall i, \exists j)\ a_{1,i} \prec a_{2,j} : \underset{EQ}{A}$. Observe that:

$$\begin{aligned}\textbf{for } (x \in \gamma_i(e))\ \gamma_i(f) &\longmapsto^* \textbf{for } (x \in \{v_{i,j}\}_j)\ \gamma_i(f) \\ &\longmapsto^* \bot_L \vee_L f\{\gamma_i, x \mapsto v_{i,1}\} \vee_L f\{\gamma_i, x \mapsto v_{i,2}\} \vee_L \dots \\ &= \bot_L \vee_L a_{i,1} \vee_L a_{i,2} \vee_L \dots\end{aligned}$$

Then by closure under stepping we wish to show:

$$(\bot_L \vee_L a_{1,1} \vee_L a_{1,2} \vee_L \dots) \prec (\bot_L \vee_L a_{2,1} \vee_L a_{2,2} \vee_L \dots) : L$$

Applying bottom is bottom and join is join, this follows from $(\forall i, \exists j)\ a_{1,i} \prec a_{2,j} : \underset{EQ}{A}$, because then transitively

$$a_{1,i} \prec a_{2,i} \prec (\bot_L \vee_L a_{2,1} \vee_L a_{2,2} \vee_L \dots) : L$$

Showing that the latter is an upper bound of each $a_{1,i}$ and since $(\bot_L \vee_L a_{1,1} \vee_L a_{1,2} \vee_L \dots)$ is least among such upper bounds, we have what we desire.

Case $\dfrac{(\lceil\Gamma\rceil \vdash e_i : \underset{EQ}{A})_i}{\Gamma \vdash e_1 = e_2 : \mathit{bool}}$. By lemma 13 for some $v_{i,j}$ we have $\gamma_j(e_i) \longmapsto^* v_{i,j}$ such that $v_{i,1} \equiv v_{i,2} : \underset{EQ}{A}$. It suffices to show that $(v_{1,1} = v_{2,1}) \equiv (v_{1,2} = v_{2,2})$: bool. Since pretty plainly true $\prec$ true : bool and false $\prec$ false : bool (by the same reasoning as in the case for $\{e_i\}_i$), and since $v_{1,j} = v_{2,j}$ steps to either true or false depending on whether $v_{1,j} = v_{2,j} : \underset{EQ}{A}$, it suffices to show that $v_{1,1} = v_{2,1} : \underset{EQ}{A} \iff v_{1,2} = v_{2,2} : \underset{EQ}{A}$. By lemma 10 we know that this decidable ordering test coincides with equivalence in the logical relation, thus this holds because $v_{i,1} \equiv v_{i,2} : \underset{EQ}{A}$.

Case $\dfrac{\lceil\Gamma\rceil \vdash e : \{1\}}{\Gamma \vdash \mathit{empty?}\ e : 1 + 1}$. Applying our IH and lemma 13 we have $\gamma_1(e) \equiv \gamma_2(e) : \{1\}$. This means we have some $\vec{v}, \vec{u}$ such that $\gamma_1(e) \longmapsto^* \{\vec{v}\}$ and $\gamma_2(e) \longmapsto^* \{\vec{v}\}$ and $\{\vec{v}\} \equiv \{\vec{u}\} : \{1\}$. This implies that $\vec{v}$ is empty if and only if $\vec{u}$ is empty. In one case $\mathit{empty?}\ \gamma_i(e) \longmapsto^* \mathsf{in}_1\ ()$; in the other, $\mathsf{in}_2\ ()$. So by closure under stepping it suffices to show $\mathsf{in}_j\ () \in \mathsf{Ok}_V(1 + 1)$ for $j \in \{1, 2\}$; which it is by unrolling the definitions involved.

Case $\dfrac{\Gamma \vdash e : \Box(A_1 + A_2)}{\Gamma \vdash \mathit{split}\ e : \Box A_1 + \Box A_2}$. By our IH and the LR for $\Box(A_1 + A_2)$ we have for some $i \in \{1, 2\}$ and $v_1 \equiv v_2 : A_i$ that $\gamma_j(e) \longmapsto^* [\mathsf{in}_i\ v_j]$. Then $\gamma_j(\mathsf{split}\ e) \longmapsto^* \mathsf{split}\ [\mathsf{in}_i\ v_j] \longmapsto \mathsf{in}_i\ [v_j]$, and using the definition of the LR we have $\mathsf{in}_i\ [v_1] \prec \mathsf{in}_i\ [v_2] : \Box A_1 + \Box A_2$ and by closure under stepping we are done.

□

Chapter 3

Seminaïve Evaluation

In chapter 2 we presented Datafun's syntax and semantics. These semantics are straightforward to implement directly; implementing them *efficiently* is more difficult. Datalog has decades of well-studied implementation and optimization techniques. To explore whether these techniques can be transferred to Datafun, in this chapter we'll examine just one classic Datalog optimization, *seminaïve evaluation*, which makes practical Datalog and Datafun's defining feature: iterative fixed points.

In §3.1 we'll see how the direct approach to finding fixed points wastes time by rediscovering previously-known facts at each iteration, and how seminaïve evaluation fixes this by computing the differences or changes between iterations. We therefore see seminaïve evaluation as a matter of incremental computation, that is, efficiently responding to change. To apply this insight, in §3.2 we adapt prior work on the incremental lambda calculus (Cai et al., 2014) to construct a category of incrementalizable monotone maps capable of interpreting Datafun's semantics. Using this construction as a guide, our central contribution (§3.4) is a pair of static Datafun-to-Datafun translations which enable the seminaïve fixed-point-finding strategy. Finally, we prove these transformations correct using a logical relation (§3.5).

3.1 Seminaïve evaluation as incremental computation

Consider the following Datalog program:

$$\mathsf{path}(X, Z) \leftarrow \mathsf{edge}(X, Z).$$
$$\mathsf{path}(X, Z) \leftarrow \mathsf{edge}(X, Y),\ \mathsf{path}(Y, Z).$$

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

$$\mathsf{path}_{i+1}(X, Z) \leftarrow \mathsf{edge}(X, Z).$$
$$\mathsf{path}_{i+1}(X, Z) \leftarrow \mathsf{edge}(X, Y),\ \mathsf{path}_i(Y, Z).$$

When we index a relation in this way, the indexed relation at time i will contain exactly those facts deducible by applying the original rules at most i times. For instance, since the rules for path append edges one at a time, we can show by induction that path_i contains exactly the

nonempty paths of i or fewer edges. By omission $\text{path}_0 = \emptyset$: there are no empty nonempty paths. Inductively assuming for some $i \geqslant 0$ that path_i contains the nonempty paths of at most i edges, note that "a nonempty path of length at most $i+1$" is the same as "an edge, optionally followed by a nonempty path of length at most i". The singleton edges are included by the first clause, and edges followed by paths by the second clause (applying our inductive hypothesis); so path_{i+1} contains exactly the paths of length at most $i+1$ as desired.

The first clause $\text{path}_{i+1}(X, Z) \leftarrow \text{edge}(X, Z)$ ensures path_{i+1} includes all 1-edge paths; $\text{path}_{i+1}(X, Z) \leftarrow \text{edge}(X, Y)$, $\text{path}_i(Y, Z)$ includes all paths formed by prepending an edge to a path from path_i, i.e. (by our inductive hypothesis) paths of between 2 and $i+1$ edges. So path_{i+1} contains exactly the paths of length 1 to $i+1$, as desired.

Unfortunately, this strategy *re-deduces* each previously known fact on every subsequent iteration. For example, suppose $\text{path}_i(j, k)$ holds. Then at step $i+1$ the second rule deduces $\text{path}_{i+1}(j-1, k)$ from $\text{edge}(j-1, j) \wedge \text{path}_i(j, k)$. But since $\text{path}_{i+1}(j, k)$ holds (a path of length at most i is also a path of length at most $i+1$), we perform the same deduction at time $i+2$, and again at $i+3$, $i+4$, etc.

For our linear graph, it's easy to calculate how much work these re-deductions waste. The longest path in a linear graph of n nodes has $n-1$ edges, so we take n steps to discover a fixed point $\text{path}_{n-1} = \text{path}_n$. Since step i involves $|\text{path}_i| = \sum_{j=1}^{i} n-j \in \Theta(i \cdot n)$ deductions, we make $\Theta(n^3)$ deductions in total. There being only $\Theta(n^2)$ paths in the final result, this is terribly wasteful; hence we term this *naïve evaluation*.

Now let's move from Datalog to Datafun.[1] The transitive closure of edge is the least fixed point of the monotone function step defined by:

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

The naïve way to compute this is to simply iterate step, computing $\text{path}_i = \text{step}^i\ \emptyset$ inductively by letting:

$$\text{path}_0 = \emptyset \qquad\qquad \text{path}_{i+1} = \text{step path}_i$$

But as before, $\text{path}_i \subseteq \text{step path}_i$; each iteration re-computes the paths found by its predecessor. We'd rather not compute the entire set, step path_i, but instead find a smaller subset of *new* paths, let's call them dpath_i, such that $\text{path}_i \cup \text{dpath}_i = \text{step path}_i$. The smallest such set is of course $\text{step path}_i \setminus \text{path}_i$, but we won't need this most-precise difference to prove our strategy correct, and the freedom to approximate can be useful for avoiding unnecessary work. Our iteration strategy then becomes:

$$\begin{aligned} \text{path}_0 &= \emptyset & \text{path}_{i+1} &= \text{path}_i \cup \text{dpath}_i \\ \text{dpath}_0 &= ? & \text{dpath}_{i+1} &= ? \end{aligned}$$

The base case is easily solved by letting $\text{dpath}_0 = \text{step}\ \emptyset$. This wastes no work since there are no previously-known paths to be rediscovered. In the inductive case, we need to compute dpath_{i+1} from path_i and dpath_i. Let's imagine we have a function that does this, called step':

$$\begin{aligned} \text{path}_0 &= \emptyset & \text{path}_{i+1} &= \text{path}_i \cup \text{dpath}_i \\ \text{dpath}_0 &= \text{step}\ \emptyset & \text{dpath}_{i+1} &= \text{step}'\ \text{path}_i\ \text{dpath}_i \end{aligned}$$

[1] In this section we do not bother distinguishing monotone variables X or discrete expressions e, as it muddies our examples to no benefit.

What must step' satisfy to prove this iteration strategy correct? We wish to show inductively that $\text{step}\ \text{path}_i = \text{path}_{i+1}$ for all i. The base case is trivial. So assuming $\text{step}\ \text{path}_i = \text{path}_{i+1}$, let's look at what we wish to prove and simplify it:

$$\begin{aligned}
\text{step}\ \text{path}_{i+1} &= \text{path}_{i+2} && \text{what we wish to show} \\
\text{step}\ (\text{path}_i \cup \text{dpath}_i) &= \text{path}_{i+1} \cup \text{dpath}_{i+1} && \text{apply definitions} \\
\text{step}\ (\text{path}_i \cup \text{dpath}_i) &= \text{step}\ \text{path}_i \cup \text{step}'\ \text{path}_i\ \text{dpath}_i && \text{apply IH and definitions}
\end{aligned}$$

So it suffices for step' to have the following property:

$$\text{step}\ (s \cup ds) = \text{step}\ s \cup \text{step}'\ s\ ds$$

Intuitively speaking, $\text{step}'\ s\ ds$ captures how step's output changes in response to changing input: as s grows to $s \cup ds$, how does $\text{step}\ s$ grow to $\text{step}\ (s \cup ds)$? This makes sense: our iterations are the outputs of step applied to increasing inputs. To compute the changes between them, we want to know how step's output responds to growth in its input.

The next question is: can we find a step' with this property? We can: for instance, $\text{step}'\ s\ ds = \text{step}\ (s \cup ds) \setminus \text{step}\ s$, or the even simpler $\text{step}'\ s\ ds = \text{step}\ (s \cup ds)$. While technically correct, these solutions are not efficient: if we plug them into our revised iteration strategy, we find ourselves repeatedly calling step on ever-growing inputs, returning us to naïve iteration. So let's examine the behavior of $\text{step}\ (s \cup ds)$ to see if we can find a better alternative:

$$\begin{aligned}
&\text{step}\ (s \cup ds) \\
&= \text{edge} \cup \{(x,z) \mid (x,y) \in \text{edge},\ (y,z) \in s \cup ds\} \\
&= \text{edge} \cup \{(x,z) \mid (x,y) \in \text{edge},\ (y,z) \in s\} \cup \{(x,z) \mid (x,y) \in \text{edge},\ (y,z) \in ds\} \\
&= \text{step}\ s \cup \{(x,z) \mid (x,y) \in \text{edge},\ (y,z) \in ds\}
\end{aligned}$$

Thus, a satisfactory definition of step' is:

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

Is this efficient? Plugging this into our iteration strategy:

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

Applying this to our original linear graph example, $\text{edge} = \{(0,1), (1,2), \ldots, (n-1,n)\}$, we find:

$$\begin{aligned}
\text{dpath}_0 &= \{(i, i+1) \mid 0 \leqslant i < i+1 \leqslant n\} \\
\text{dpath}_1 &= \{(i, i+2) \mid 0 \leqslant i < i+2 \leqslant n\} \\
\text{dpath}_2 &= \{(i, i+3) \mid 0 \leqslant i < i+3 \leqslant n\} \\
&\vdots \\
\text{dpath}_k &= \{(i, i+k+1) \mid 0 \leqslant i < i+k+1 \leqslant n\}
\end{aligned}$$

Thus dpath_k captures exactly the paths of length k, so each path is discovered exactly once: we have avoided redundant work by computing only the change between iterations of our step function.

The problem of seminaïve evaluation for Datafun reduces to *automatically* finding functions, like step', that efficiently 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 use an approach rooted in the *incremental* λ*-calculus* (Cai et al., 2014; Giarrusso, 2020; Giarrusso et al., 2019).

3.2 Change structures for Datafun

To solve the problem of computing how a function's output changes in response to its input, we must first make precise the notion of *change* for each type in our language. To do this, incremental λ-calculi associate every type A with a *change structure*. In our case, noting that Datafun types denote posets, we define change structures as follows:

Definition 14. A *change structure* A consists of a poset VA, a poset ΔA, and a relation $R_A \subseteq \Delta A \times VA \times VA$. For $dx : \Delta A$ and $x, y : VA$, we will write $(dx, x, y) \in R_A$ interchangeably as $dx \rhd x \hookrightarrow y : A$. This relation must satisfy three properties:

Functionality	If $dx \rhd x \hookrightarrow y : A$ and $dx \rhd x \hookrightarrow z : A$ then $y = z$.
Soundness	If $dx \rhd x \hookrightarrow y : A$ then $x \leqslant_A y$.
Zero changes	If $x : VA$ there is some $dx : \Delta A$ such that $dx \rhd x \hookrightarrow x : A$.

Some useful terminology and notation: We can think of elements $dx \in \Delta A$ as *changes* or "diffs" to values $x \in VA$. The relation R_A tell us how changes affect values: we gloss $dx \rhd x \hookrightarrow y : A$ as "dx changes x into y". We say that dx is a *valid* change to x if there is some y such that $dx \rhd x \hookrightarrow y : A$. When $dx \rhd x \hookrightarrow x$ we call dx a *zero change* to x; when we need to pick such a change we write $\mathbb{0}_x$. By the axiom of choice, the *zero changes* property is equivalent to the existence of such a $\mathbb{0}$ function.

Although we use multi-letter variable names prefixed with "d" for elements $dx, dy : \Delta A$ of delta posets, this is merely a naming convention; we could instead use single-letter variables like $p, q : \Delta A$ with the same meaning.

To motivate our three properties, it will help to consider an example of a change structure corresponding to an important Datafun type: finite sets $\{A\}_{EQ}$. Recall that our goal is to speed up fixed point computation. Since iterations toward a fixed point grow monotonically, in Datafun we only need *increasing* changes. Therefore, changes to sets are themselves sets, to be unioned in:

$$V\{A\}_{EQ} = \{A\}_{EQ} \qquad \Delta\{A\}_{EQ} = \{A\}_{EQ} \qquad \frac{x \cup dx = y}{dx \rhd x \hookrightarrow y : \{A\}_{EQ}}$$

Functionality says that $dx \rhd x \hookrightarrow y : \{A\}_{EQ}$ must be a partial function from (dx, x) to y. In this case, it's a total function: set union. *Soundness* requires that all changes are increasing, which is true since $x \subseteq x \cup dx$. Finally, *zero changes* holds since $x \cup \emptyset = x$; one can leave a set unchanged by adding nothing.[2]

[2] Indeed, sets have not only zero changes but all increasing changes: for any $x \leqslant y$ there is a dx such that

We'll see more examples of change structures later, including ones where the validity relation is a partial rather than a total function, but first, let's revisit our transitive closure example from §3.1. Using change structures we can generalize the relation between step and step′. We call step′ a *derivative*, because it tells us how step's output changes in respond to its input changing:

Definition 15. A *derivative* of a monotone map $f : A \to B$ between change structures A, B is a monotone map $f' : \Box VA \to \Delta A \to \Delta B$ satisfying the law (for all x, y, dx):

$$dx \triangleright x \hookrightarrow y : A \implies f'\ x\ dx \triangleright f\ x \hookrightarrow f\ y : B$$

We say *a* derivative, not *the* derivative, because derivatives are not necessarily unique. This is because changes are not necessarily unique: for fixed x, y there may be many dx such that $dx \triangleright x \hookrightarrow y$.

Applying this definition to our change structure for finite sets, we recover the relationship we needed between step and step′ in §3.1 for seminaïve evaluation:

$$\begin{aligned} ds \triangleright s \hookrightarrow t : \{\underset{\text{EQ}}{A}\} &\implies \mathsf{step}'\ s\ ds \triangleright \mathsf{step}\ s \hookrightarrow \mathsf{step}\ t : \{\underset{\text{EQ}}{A}\} \\ &\textit{iff} \\ s \cup ds = t &\implies \mathsf{step}\ s \cup \mathsf{step}'\ s\ ds = \mathsf{step}\ t \\ &\textit{iff} \\ \mathsf{step}\ (s \cup ds) &= \mathsf{step}\ s \cup \mathsf{step}'\ s\ ds \end{aligned}$$

This generalization is useful because differentiable maps (that is, maps possessing a derivative in the above sense) *compose;* in fact, they form a category:

Definition 16. The category **ΔPoset** has as objects change structures A, B and as morphisms differentiable monotone maps $f : VA \to VB$, that is, maps having at least one derivative $f' : \Box VA \to \Delta A \to \Delta B$ (also monotone). Morphism composition and the identity morphism are both as in **Poset**.

Proof. Of course, for this to be a category we need to show that: (1) the identity map is differentiable; (2) the composition of two differentiable maps is differentiable. We also need associativity and identity of composition, but these follow from the same in **Poset**. The derivative for identity is trivial, while the derivative for composition follows the pattern of the chain rule from calculus:

$$\mathsf{id}'\ x\ dx = dx \qquad\qquad (g \circ f)'\ x\ dx = g'\ (f\ x)\ (f'\ x\ dx)$$

That id' is a derivative of id is straightforward, while for composition we need to pick maps f', g' which are derivatives of f, g respectively; then, applying the definition of derivatives:

$$\begin{aligned} & dx \triangleright x \hookrightarrow y \\ \implies & f'\ x\ dx \triangleright f\ x \hookrightarrow f\ y \\ \implies & g'\ (f\ x)\ (f'\ x\ dx) \triangleright g\ (f\ x) \hookrightarrow g\ (f\ y) \end{aligned}$$

$dx \triangleright x \hookrightarrow y : \{\underset{\text{EQ}}{A}\}$; for instance one may let $dx = y \setminus x$, or indeed just y. We call this property *completeness,* as it is the converse of soundness. However, while our change structure for sets is complete, we will later observe that completeness is troublesome at function types, so we do not insist on it in general.

We also need $\mathrm{id}'\ x\ dx$ and $(g \circ f)'\ x\ dx$ to be monotone in dx, which they are, for straightforward compositional reasons; in general, for the remainder of §3.2 and 3.3, we omit showing that functions are monotone unless the argument is non-obvious. □

In the next section we will sketch the most important structures in **ΔPoset** needed to support Datafun's semantics, providing a recipe for incrementalizing Datafun. Applied correctly, this will let us automatically find derivatives for functions used by fix expressions, allowing us to employ the seminaïve evaluation strategy for finding fixed points faster.

3.3 The structure of ΔPoset

We should note up front that ours is only one among many reasonable notions of change structure. For instance, Giarrusso (2020) defines both *basic change structures* (definition 12.1.1), consisting only of a delta-set and a relation, and the more elaborate *change structures* (definition 13.1.1) that have an update operator $\oplus : A \times \Delta A \to A$, a difference operator $\ominus : A \times A \to \Delta A$, and composition of changes $\circledcirc : \Delta A \times \Delta A \to \Delta A$; while Alvarez-Picallo (2020) uses a definition based on monoid actions. We will compare these with our approach in more detail in §5.2, but the "big picture" difference is that we are pervasively concerned with *monotone functions, increasing changes,* and *higher-order computation;* most of our choices flow from one more more of these considerations.

Our eventual destination is a static transformation on Datafun source code which implements seminaïve evaluation (§3.4). This transformation, originally presented by Arntzenius and Krishnaswami (2020), predates the construction of **ΔPoset** presented here and is independent of it. The transformation itself is quite intricate; our aim in presenting **ΔPoset** is to break its core concepts down into small pieces, showing how this complexity arises and suggesting potential alternatives for future investigation. In service of this goal, we have chosen what seems the simplest definition of change structure that both supports the features of Datafun and provides useful intuition. Ultimately, however, we deploy a logical relations argument to prove the translation correct (§3.5). A reader who does not care for a categorical view and is prepared to jump "in the deep end," therefore, may skim this section or jump straight to the definition of the seminaïve transformation itself in §3.4.

Recall that the structures we needed to interpret Datafun into the category **Poset** in §2.3.2 and 2.3.3 were: products, sums, exponentials, a discreteness comonad to interpret □, sets and semilattice objects, equality-test morphisms, and fixed points. In **ΔPoset** we will cover products, sums, exponentials, the discreteness comonad, and fixed points, as they are the most significant for understanding the broad structure of our approach; the other structures are straightforward (with the exception of $\mathsf{collect}(f)$, which we discuss as **for** in §3.4.6).

3.3.1 Products

Products and the terminal object in **ΔPoset** mirror those in **Poset**:

$$V1 = 1 \qquad V(A \times B) = VA \times VB$$

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

$$() \rhd () \hookrightarrow () : 1$$

$$\textsc{product change}\quad \frac{da \rhd a \hookrightarrow a' : A \qquad db \rhd b \hookrightarrow b' : B}{(da, db) \rhd (a, b) \hookrightarrow (a', b') : A \times B}$$

These satisfy functionality, soundness, and zero changes by invoking the corresponding properties at A and B. For instance, picking zero changes 0_a, 0_b for a, b respectively, PRODUCT CHANGE tells us $(0_a, 0_b)$ is a zero change to (a, b).

Finally, the terminal map $\langle\rangle$, projection π_i, and the tupling $\langle f, g\rangle$ of $f : A \to B$ and $g : A \to C$ are all the same as in **Poset** (thus inheriting the necessary universal properties), with derivatives given by:

$$\begin{aligned} \langle\rangle &: A \to 1 & \langle\rangle'\ a\ da &= () \\ \pi_i &: A_1 \times A_2 \to A_i & \pi_i'\ (x_1, x_2)\ (dx_1, dx_2) &= dx_i \\ \langle f, g\rangle &: A \to B \times C & \langle f, g\rangle'\ a\ da &= (f'\ a\ da, g'\ a\ da) \end{aligned}$$

The correctness of $\langle\rangle'$ is trivial; correctness of π_i' follows by inversion of PRODUCT CHANGE; and $\langle f, g\rangle'$ is correct by PRODUCT CHANGE and correctness of f', g'.[3]

3.3.2 Sums

Sums and the initial object also mirror those in **Poset**:

$$\begin{aligned} V0 &= 0 & V(A + B) &= VA + VB \\ \Delta 0 &= 0 & \Delta(A + B) &= \Delta A + \Delta B \\ R_0 &= \emptyset & \frac{dx \rhd x \hookrightarrow y : A_i}{\mathsf{in}_i\ dx \rhd \mathsf{in}_i\ x \hookrightarrow \mathsf{in}_i\ y : A_1 + A_2}\ &\text{SUM CHANGE} \end{aligned}$$

These satisfy functionality, soundness, and zero changes pretty straightforwardly using the corresponding properties at A and B. For instance, $\mathsf{in}_i\ 0_x$ is a zero change to $\mathsf{in}_i\ x$.

The initial map $[]$, injection in_i, and case-analysis $[f_1, f_2]$ (given $f_1 : A_1 \to C$, $f_2 : A_2 \to C$) are the same as in **Poset** (inheriting its universal properties), with derivatives as follows:

$$\begin{aligned} [] &: 0 \to A & []' &= [] \quad \text{(the domain is empty)} \\ \mathsf{in}_i &: A_i \to A_1 + A_2 & \mathsf{in}_i'\ x\ dx &= \mathsf{in}_i\ dx \\ [f, g] &: A_1 + A_2 \to C & [f_1, f_2]'\ (\mathsf{in}_i\ x)\ (\mathsf{in}_j\ dx) &= \begin{cases} f_i'\ x\ dx & \text{if } i = j \\ \textit{anything of type } \Delta C & \text{if } i \neq j \end{cases} \end{aligned}$$

Correctness of $[]'$ is vacuous; correctness of in_i' follows directly from SUM CHANGE; but the definition of $[f_1, f_2]'$ requires explanation. If we take the proposition that $[f_1, f_2]'$ is a derivative of $[f_1, f_2]$ and apply the definition of $R_{A_1+A_2}$ (namely SUM CHANGE), we find that it simplifies to:

$$dx \rhd x \hookrightarrow y : A_i \implies [f_1, f_2]'\ (\mathsf{in}_i\ x)\ (\mathsf{in}_i\ dx) \rhd f_i\ x \hookrightarrow f_i\ y : C$$

This only constrains the behavior of $[f_1, f_2]'\ (\mathsf{in}_i\ x)\ (\mathsf{in}_j\ dx)$ when $i = j$; and in this case, we have $f_i'\ x\ dx \rhd f_i\ x \hookrightarrow f_i\ y : C$ as desired. Since the $i \neq j$ case is unconstrained, any

[3] Note that had we chosen to let $\Delta(A \times B) = \Delta A + \Delta B$, representing a change to a tuple by a change to only one of its components, this would not allow us to differentiate tupling $\langle f, g\rangle$, since a change to the input may cause both components of the output to change simultaneously.

value of type ΔC will suffice; all we need for differentiability is to show one exists, i.e. that ΔC is inhabited. Fortunately, in this case we have an $x : VA_i$ and a differentiable function $f_i : A_i \to C$. Applying *zero changes* at A_i we can pick a zero-change 0_x (although it being a zero-change is unnecessary; all we need is an element of ΔA_i) and take $f_i' \; x \; 0_x : \Delta C$. Or, we could use zero-changes at C instead and take $0_{(f_i \; x)} : \Delta C$.

This $i \neq j$ case is related to the *partiality* of the validity relation: $\mathsf{in}_1 \; dx$ is never a valid change to $\mathsf{in}_2 \; x$. This is hard to avoid given our definition of change structures: to differentiate in_i and $[f_1, f_2]$ we need $\Delta(A_1 + A_2)$ to include both ΔA_1 and ΔA_2 somehow; and a change $dx \in \Delta A_1$ has no natural meaning applied to a value $x \in VA_2$. Furthermore, the fact that the $i \neq j$ case is unconstrained – essentially "dead code" – means that if we had defined **ΔPoset** morphisms as maps *equipped with a particular derivative* (rather than merely differentiable) we would be unable to prove the uniqueness of $[f, g]'$ required by the universal property for sums.[4]

This could, for instance, be addressed by changing the definition of **ΔPoset** to only require derivatives to be defined for valid changes. We don't do this because from a type-theoretic perspective, this requires a dependent or refinement type, while we want the types of our derivatives to be simple so our category corresponds closely with a static transformation on Datafun, a simply-typed language.

3.3.3 Exponentials

The values of the exponentials in **ΔPoset** capture differentiable, monotone maps:

$$\begin{aligned} V(A \Rightarrow B) &= \text{differentiable monotone maps } VA \to VB, \text{ ordered pointwise} \\ &= (\mathbf{\Delta Poset}(A, B), \{(f, g) : (\forall x) \; f \; x \leqslant g \; x\}) \end{aligned}$$

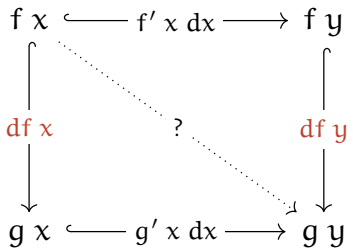
We might expect changes $\Delta(A \Rightarrow B)$ to be given pointwise, as (not necessarily monotone) functions $VA \to \Delta B$ mapping each input to the change in the corresponding output:

$$\Delta(A \Rightarrow B) = \Box VA \Rightarrow \Delta B \qquad \frac{(\forall x) \; df \; x \rhd f \; x \hookrightarrow g \; x : B}{df \rhd f \hookrightarrow g : A \Rightarrow B} \qquad \text{✗ NOT AN EXPONENTIAL}$$

However, this choice makes it difficult to differentiate function application. The function application map $\mathsf{eval} : (A \Rightarrow B) \times A \to B$ is, of course, given by $\mathsf{eval} \; (f, x) = f \; x$. To differentiate this is to ask for some $\mathsf{eval}' \; (f, x) \; (df, dx)$ that captures how $f \; x$ changes as both f and x change simultaneously: supposing $df \rhd f \hookrightarrow g$ and $dx \rhd x \hookrightarrow y$, how do we find a change $f \; x \hookrightarrow g \; y$?

Using a pointwise change $df : VA \to \Delta B$, we can find $df \; x \rhd f \; x \hookrightarrow g \; x$; and applying differentiability of f we can find some $f' \; x \; dx \rhd f \; x \hookrightarrow f \; y$. The former handles a change to the function, the latter a change to the argument. These form two sides of a "square of changes":

[4] We could avoid partiality by defining $\mathsf{in}_i \; dx \rhd \mathsf{in}_j \; x \hookrightarrow \mathsf{in}_j \; x : A_1 + A_2$ for $i \neq j$; that is, treating currently "invalid" changes as zero-changes. This unfortunately doesn't extend to the function case, which as we'll see shortly also needs a partial validity relation. Moreover, it doesn't ensure uniqueness of $[f_1, f_2]'$: although it requires $[f_1, f_2]' \; (\mathsf{in}_i \; x) \; (\mathsf{in}_j \; dx)$ to be a zero change to $f_i \; x$ when $i \neq j$, there may be multiple such zero changes.

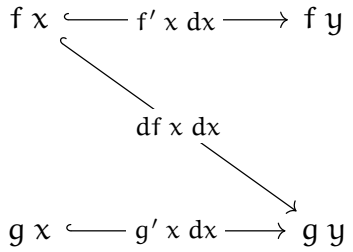


We need the *diagonal* of this square. One approach would be to use pointwise function changes but augment our definition of change structures to allow *composing* changes, and find the diagonal by composing sides. Unfortunately, this is more difficult than it appears: eval' is applied to f, x, df, dx, but to compute $df\ y$ or $g'\ x\ dx$ we need either y or g. This seems to require equipping change structures with an operator $\oplus_A : \Box VA \times \Delta A \to VA$ that extends the validity relation R_A from a partial to a *total* function (since eval' is defined for all inputs, not merely valid ones); then we can recover $y = x \oplus dx$ or $g = f \oplus df$. But $\oplus_{A\to B}$ is difficult to construct, because we must guarantee that $f \oplus df$ is a *monotone* function, no matter the value of $df : \Box VA \to \Delta B$; it is easy to come up with a df such that $\lambda x.\ f\ x \oplus df\ x$ (the natural definition of $f \oplus df$) is non-monotone.[5]

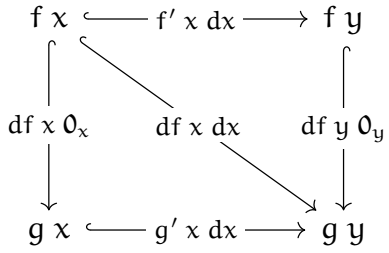
Perhaps there is some way through these difficulties; fortunately, there is a simple approach that side-steps them entirely: following the original incremental λ-calculus (Cai et al., 2014) we require function changes to produce the diagonal *directly*. Since this diagonal depends on the change dx to the argument, function changes df become two-argument functions:

$$\Delta(A \Rightarrow B) = \Box VA \Rightarrow (\Delta A \Rightarrow \Delta B) \qquad \frac{(\forall dx \rhd x \hookrightarrow y : A)\ df\ x\ dx \rhd f\ x \hookrightarrow g\ y : B}{df \rhd f \hookrightarrow g : A \to B}\ \textsc{fn change}$$

With this definition, function changes are exactly what is needed to incrementalize function application $f\ x$. A change to a function $df \rhd f \hookrightarrow g$ accepts a change in its argument $dx \rhd x \hookrightarrow y$ and produces the the change in its output, $df\ x\ dx \rhd f\ x \hookrightarrow g\ y$. If we return to our square of changes, we find it now has a zig-zag shape, with the diagonal filled in but missing the vertical sides:



To recover the missing sides, we can apply df to zero-changes 0_x and 0_y instead of dx:



[5] We further discuss the issue of monotonicity and pointwise changes in §5.2.2.

Zero changes thus let us recover a pointwise change $\lambda x.\ df\ x\ 0_x$ from any $df \triangleright f \hookrightarrow g : A \Rightarrow B$.

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

Although this nicely solves the problem of differentiating eval, it is not immediately obvious that $R_{A\Rightarrow B}$ is functional, sound, and possesses zero-changes.[6] The first two are quite similar, so we'll tackle them together: Suppose $df \triangleright f \hookrightarrow g : A \Rightarrow B$ and likewise $df \triangleright f \hookrightarrow h$ and fix some $x \in VA$. For functionality, we wish to show $g\ x = h\ x$; for soundness, we wish to show $f\ x \leqslant g\ x$. By zero changes at A we can pick some $0_x \triangleright x \hookrightarrow x$. Inverting FN CHANGE we have $df\ x\ 0_x \triangleright f\ x \hookrightarrow g\ x : B$ and likewise $df\ x\ 0_x \triangleright f\ x \hookrightarrow h\ x$. Then by functionality at B we have $g\ x = h\ x$; and by soundness at B we have $f\ x \leqslant g\ x$.

Showing *zero changes* is simple but illuminating. By definition, every $f : V(A \Rightarrow B)$ is differentiable, and a derivative f' of f is exactly a zero change $f' \triangleright f \hookrightarrow f : A \Rightarrow B$:

$$
\begin{aligned}
& df \triangleright f \hookrightarrow f : A \to B && \\
\iff & (\forall dx \triangleright x \hookrightarrow y : A)\ df\ x\ dx \triangleright f\ x \hookrightarrow f\ y : B && \text{FN CHANGE} \\
\iff & df \text{ is a derivative of } f && \text{definition 15}
\end{aligned}
$$

This happens because we've defined function changes $df \triangleright f \hookrightarrow g : A \Rightarrow B$ to tell us how function application responds to changes in both the function and its argument. If the function *doesn't* change, this reduces to how the function's output changes as its argument changes: exactly what a derivative does.

Finally, to make $A \Rightarrow B$ an exponential we need function application $eval_{A,B} : (A \Rightarrow B) \times A \to B$ (which we have already discussed), and for any $f : C \times A \to B$, its currying $\lambda f : C \to A \Rightarrow B$. These are defined as in **Poset**, which ensures their universal property holds; but since $V(A \Rightarrow B)$ contains only *differentiable* maps, besides eval and λf we also require $(\lambda f\ c)$ to be differentiable:

---

[6] We also promised in a footnote on page 45 to show that completeness was problematic at function types. In other words, supposing $f \leqslant g : A \Rightarrow B$, why can't we find some $df \triangleright f \hookrightarrow g$? Well, we would need $df\ x\ dx \triangleright f\ x \hookrightarrow g\ y : B$ whenever $dx \triangleright x \hookrightarrow y$. If we inductively suppose completeness at B, we could pick such a change, since by monotonicity we can show $f\ x \leqslant g\ y$. Of course, we need df to be defined over *all* x, dx, not merely valid ones, but this is nothing the axiom of choice can't handle. More problematic is that we need $df\ x\ dx$ to be *monotone* with respect to dx. So merely picking changes is not enough; we have to pick them in a way that preserves monotonicity.

We conjecture this can be done by strengthening change structures to include (1) monotone completeness and (2) monotone change composition. *Monotone completeness* strengthens completeness by requiring an operator $y \ominus_A x$ defined for $y \geqslant x : VA$ such that $y \ominus_A x \triangleright x \hookrightarrow y : A$ and which is monotone in y and anti-monotone in x, that is, $x' \leqslant x \wedge y \leqslant y' \implies y \ominus x \leqslant y' \ominus x'$. *Monotone change composition* requires a monotone operator $\circledcirc_A : \Delta A \times \Delta A \to \Delta A$ such that if $dx \triangleright x \hookrightarrow y$ and $dy \triangleright y \hookrightarrow z$ then $dy \circledcirc dx \triangleright x \hookrightarrow z$. Then we can define $g \ominus_{A\Rightarrow B} f = \lambda x.\ \lambda dx.\ g'\ x\ dx \circledcirc_B (g\ x \ominus_B f\ x)$ and $dg \circledcirc_{A\Rightarrow B} df = \lambda x.\ \lambda dx.\ dg\ x\ dx \circledcirc df\ x\ 0_x$.

We have not done this because it considerably complicates the definition of change structures and does not help explain any features of the translation given in §3.4, but it might be an interesting direction for future work.

$$\begin{aligned} \mathsf{eval}\ (f, x) &= f\ x & \lambda f\ c\ x &= f\ (c, x) \\ \mathsf{eval}'\ (f, a)\ (df, da) &= df\ a\ da & (\lambda f)'\ c\ dc\ a\ da &= f'\ (c, a)\ (dc, da) \\ & & (\lambda f\ c)'\ a\ da &= f'\ (c, a)\ (\mathcal{0}_c, da) \end{aligned}$$

We've already seen how eval′s correctness follows from FN CHANGE. Applying $R_{A \Rightarrow B}$ and $R_{C \times A}$, we find that $(\lambda f)'$ is a derivative for λf when f' is a derivative for f:

$$\begin{aligned} & (\lambda f)' \text{ is a derivative of } \lambda f \\ \iff & (\forall dc \rhd c \hookrightarrow c' : C)\ (\lambda f)'\ c\ dc \rhd \lambda f\ c \hookrightarrow \lambda f\ c' : A \Rightarrow B \\ \iff & (\forall dc \rhd c \hookrightarrow c' : C)\ (\forall da \rhd a \hookrightarrow a' : A)\ (\lambda f)'\ c\ dc\ a\ da \rhd \lambda f\ c\ a \hookrightarrow \lambda f\ c'\ a' : B \\ \iff & (\forall\ (dc, da) \rhd (c, a) \hookrightarrow (c', a') : C \times A)\ f'\ (c, a)\ (dc, da) \rhd f\ (c, a) \hookrightarrow f\ (c', a') : B \\ \iff & f' \text{ is a derivative of } f \end{aligned}$$

Finally, the correctness of $(\lambda f\ c)'$ follows from that of f' by applying $\mathcal{0}_c \rhd c \hookrightarrow c : C$:

$$\begin{aligned} & f' \text{ is a derivative of } f \text{ and } \mathcal{0}_c \text{ is a zero change to } c \\ \implies & (\forall da \rhd a \hookrightarrow a' : A)\ f'\ (c, a)\ (\mathcal{0}_c, da) \rhd f\ (c, a) \hookrightarrow f\ (c, a') : B \\ \iff & (\forall da \rhd a \hookrightarrow a' : A)\ (\lambda f\ c)'\ a\ da \rhd (\lambda f\ c)\ a \hookrightarrow (\lambda f\ c)\ a' : B \\ \iff & (\lambda f\ c)' \text{ is a derivative of } (\lambda f\ c) \end{aligned}$$

3.3.4 Semilattice change structures and seminaïve fixed points

We've already been introduced to the finite powerset change structure, as our introductory example in §3.2. But to define it properly as a functor $P : \mathbf{\Delta Poset} \to \mathbf{\Delta Poset}$, inheriting from the corresponding P on **Poset**:

$$VPA = PVA \qquad \Delta PA = PVA \qquad dx \rhd x \hookrightarrow y : PA \iff x \cup dx = y$$

This finite powerset change structure forms the prototype for our change structures for semilattices in general, which we need to support various language features, most importantly fixed points. We saw in §3.1–3.2 that given a function $f : PA \to PA$ and a derivative for it $f' : \square PA \to PA \to PA$ we can compute its fixed point seminaïvely as follows:

$$\begin{aligned} x_0 &= \emptyset & x_{i+1} &= x_i \cup dx_i \\ dx_0 &= f\ \emptyset & dx_{i+1} &= f'\ x_i\ dx_i \end{aligned}$$

This takes advantage of the fact that the change to a set is another set, and we apply a change using set union/semilattice join. Following this pattern, we can endow any semilattice $L :$ **Poset** with a similar change structure:

$$VL = L \qquad \Delta L = L \qquad dx \rhd x \hookrightarrow y : L \iff x \vee dx = y$$

This satisfies functionality ($\vee$ is a function), soundness ($x \leqslant x \vee y$), and zero-changes ($x \vee \bot = x$). Let's call this the *semilattice change structure* on L. By construction, the finite powerset change structure PA is the semilattice change structure on PVA; and our seminaïve fixed point strategy generalizes to any semilattice change structure:

Definition 17 (semifix)**.** Given a semilattice L with no infinite ascending chains and monotone maps $f : L \to L$ and $f' : \Box L \to L \to L$, let $\text{semifix}_L\ (f, f') = \bigvee_i x_i$ be the limit of the ascending chain defined by:

$$\begin{aligned} x_0 &= \bot & x_{i+1} &= x_i \vee dx_i \\ dx_0 &= f\ \bot & dx_{i+1} &= f'\ x_i\ dx_i \end{aligned}$$

Theorem 18. $\text{semifix}_L\ (f, f')$ is the least fixed point of f if f' is a derivative of f.

Proof. It suffices to show inductively that $x_{i+1} = f\ x_i$; from this it follows that $x_i = f^i\ \bot$, as in the naïve approach to computing a fixed point. We prove this with essentially the same argument used in §3.1 (page 44). The base case is $x_1 = x_0 \vee dx_0 = \bot \vee f\ \bot = f\ \bot = f\ x_0$, and the inductive case is:

$$\begin{aligned} x_{i+2} &= x_{i+1} \vee dx_{i+1} && \text{definition of } x_{i+2} \\ &= f\ x_i \vee f'\ x_i\ dx_i && \text{inductive hypothesis, definition of } dx_{i+1} \\ &= f\ (x_i \vee dx_i) && f' \text{ is a derivative of } f \\ &= f\ x_{i+1} && \text{definition of } x_{i+1} \end{aligned}$$

□

3.3.5 Fixed points and discreteness comonads

Theorem 18 shows we can speed up fixed points by exploiting the power of derivatives. It may seem as though this justifies a morphism $\text{fix} : (L \Rightarrow L) \to L : \mathbf{\Delta Poset}$ for any semilattice change structure L satisfying ACC. However, morphisms in $\mathbf{\Delta Poset}$ must be differentiable: does fix have a derivative? Prior work (Alvarez-Picallo et al., 2019; Arntzenius, 2017) has answered this affirmatively. One solution is to find the *fixed point of the function change:*

$$\text{fix}'\ f\ df = \text{fix}\ (df\ (\text{fix}\ f))$$

How and why this works is non-obvious; we refer the reader to Arntzenius (2017) for a full explanation. So there is indeed a morphism $\text{fix} : (L \Rightarrow L) \to L : \mathbf{\Delta Poset}$. However, from the perspective of our original goal of speeding up fixed point computations, the derivative of this morphism presents two issues. First, it isn't actually incremental: computing $\text{fix}'\ f\ df$ using this derivative requires re-computing $\text{fix}\ f$ as an argument to df![7] Second, we would naturally like to compute $\text{fix}\ (df\ (\text{fix}\ f))$ *seminaïvely,* but we have no guarantee that $(df\ (\text{fix}\ f))$ is differentiable! This requires a higher-order derivative; a coherent theory of higher-order derivatives and higher-order change structures would be enormously interesting, but we leave it to future work.

Instead, we deliberately limit the scope of our approach to avoid the need to incrementally maintain fixed points. As we mentioned in §2.3.1, in Datafun fix is not treated as a monotone operator; correspondingly the morphisms we require to interpret it are not $\text{fix} : (L \Rightarrow L) \to L$

[7] The need to recompute $\text{fix}\ f$ could likely be solved by caching intermediate values, which we discuss further in §5.2.1. Somewhat unusually, in this case we want to cache the previous output of an operation rather than its previous input.

but rather $\mathsf{fix} : \Box(L \Rightarrow L) \to L$. The idea here is that, just as $\Box$ in **Poset** captures *non-monotonicity* in an otherwise monotone world, in **ΔPoset** we can use it to capture *non-differentiability* or *non-incrementalizability* in an otherwise differentiable world.

More concretely, since we only consider increasing changes and $\Box A$ is ordered discretely, $x \leqslant y : \Box A \iff x = y$, the only possible "change" is to stay the same. We can thus extend the discreteness comonad $\Box$ on **Poset** to a comonad $\Box$ on **ΔPoset** by letting the space of changes be trivial:

$$V\Box A = \Box VA \qquad \Delta\Box A = 1 \qquad () \triangleright a \hookrightarrow a : \Box A$$

This straightforwardly satisfies functionality, soundness, and zero changes. Moreover, it inherits the monoidal comonad structure of $\Box$ from **Poset**. Fixing some map $f : A \to B$, the derivatives of functorial action, extraction, duplication, and distribution are mostly trivial:

$$\begin{aligned}
\Box(f) &: \Box A \to \Box B & \Box(f)'\, x\, () &= () \\
\varepsilon_A &: \Box A \to A & \varepsilon'_A\, x\, () &= 0_x \quad \text{(see footnote[8])} \\
\delta_A &: \Box A \to \Box\Box A & \delta'_A\, x\, () &= () \\
\mathsf{dist}^{\times}_{\Box} &: \textstyle\prod_i \Box A_i \to \Box \prod_i A_i & \mathsf{dist}^{\times}_{\Box}{}'\, x\, dx &= () \\
\mathsf{dist}^{\Box}_{+} &: \Box \textstyle\sum_i A_i \to \sum_i \Box A_i & \mathsf{dist}^{\Box}_{+}{}'\, (\mathsf{in}_i\, x)\, () &= \mathsf{in}_i\, ()
\end{aligned}$$

Finally, observe that any monotone map $f : V\Box A \to VB$ is trivially differentiable by letting $f'\, x\, () = 0_{(f\, x)}$, confirming our intuition that differentiable maps $\Box A \to B$ should coincide with not-necessarily-differentiable maps $A \to B$.

3.4 The ϕ and δ transforms

Our goal in this chapter is to automatically speed up computation of fixed points in Datafun using seminaïve evaluation, replacing $\mathsf{fix}\ f$ by $\mathsf{semifix}\ (f, f')$ where f' is a derivative for f. The previous section suggests this is possible: **ΔPoset** appears capable of interpreting Datafun's semantics; its exponential objects consist of differentiable monotone maps; and its construction provides a recipe for calculating concrete derivatives witnessing this differentiability. In this section, we carry out this recipe (with one significant change), producing two static transformations, ϕ and δ, defined in figure 3.2.

The "speed-up" transform ϕ replaces $\mathsf{fix}\ f$ by $\mathsf{semifix}\ (f, f')$ and decorates other expressions with the information we need to compute f'. In particular, to find f' we need the "incrementalization" transform, δ, which propagates changes through a program. For instance, the derivative of $(\lambda X.\ e)$ depends on how e changes in response to changes in X. In general, δe computes the change in ϕe given changes to its free variables. The rules defining δ closely resemble both the incremental λ-calculus' *Derive* operator (Cai et al., 2014) and the derivatives given in the previous section for morphisms in **ΔPoset**.

Unfortunately, when we consider terms with free variables, there is a gap between *derivatives* and *changes:* we cannot simply let $f' = \delta f$, because we want f' to be the derivative of the

[8] It is again important here that in **ΔPoset** our morphisms are merely differentiable, not equipped with derivatives, that is, that we do not distinguish morphisms by their derivative. Otherwise naturality of ε would require that $(\varepsilon \circ \Box f)' = (f \circ \varepsilon)'$, which requires $\varepsilon'\, (f\, x)\, () = f'\, x\, (\varepsilon'\, x\, ())$ and thus $0_{(f\, x)} = f'\, x\, 0_x$, which is difficult to guarantee without unique zero-changes.

$$
\begin{aligned}
\Phi 1 &= 1 & \Delta 1 &= 1 \\
\Phi\{\underset{\text{EQ}}{A}\} &= \{\Phi \underset{\text{EQ}}{A}\} \quad \text{(see lemma 19)} & \Delta\{\underset{\text{EQ}}{A}\} &= \{\underset{\text{EQ}}{A}\} \\
\Phi(\Box A) &= \Box(\Phi A \times \Delta \Phi A) & \Delta(\Box A) &= 1 \\
\Phi(A \times B) &= \Phi A \times \Phi B & \Delta(A \times B) &= \Delta A \times \Delta B \\
\Phi(A + B) &= \Phi A + \Phi B & \Delta(A + B) &= \Delta A + \Delta B \\
\Phi(A \to B) &= \Phi A \to \Phi B & \Delta(A \to B) &= \Box A \to \Delta A \to \Delta B
\end{aligned}
$$

FIGURE 3.1 *Δ and Φ type transformations*

function f, not the change to it. One way to solve this problem, suggested by the exponential in **ΔPoset**, would be to have ϕ decorate every function with its derivative. However, this is overkill: we only need derivatives for functions used in fixed point computations.

This leads to our *significant change*. Recall from §3.3.3 that a zero change to a function *is* a derivative for it: the problematic gap between derivatives and changes disappears if the function does not change. We ensure this by giving the fix the argument type $\Box(\underset{\text{FIX}}{L} \to \underset{\text{FIX}}{L})$; as we saw in §3.3.5, the type $\Box A$ represents values which do not change. Thus, rather than decorate every function with its derivative, the key strategy of the ϕ transformation is to **decorate expressions of type $\Box A$ with their zero changes.** In this way, we hijack the $\Box$ comonad to track functions that are used inside fixed points and make their derivatives available where we need them: at fix expressions.

3.4.1 Typing ϕ and δ

In order to decorate expressions with extra information, ϕ also needs to decorate their types. In figure 3.1 we give a type translation ΦA capturing this. In particular, if $e : \Box A$ then ϕe will have type $\Phi(\Box A) = \Box(\Phi A \times \Delta \Phi A)$. The idea is that evaluating ϕe will produce a pair $[(x, dx)]$ where $x : \Phi A$ is the sped-up result and $dx : \Delta \Phi A$ is a zero change to x. For example, if $e : \Box(A \to B)$, then ϕe will compute $[(f, f')]$, where f' is the derivative of f.

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

Lemma 19. $\Phi \underset{\text{EQ}}{A} = \underset{\text{EQ}}{A}$ for all equality types $\underset{\text{EQ}}{A}$.

Proof. Induct on $\underset{\text{EQ}}{A}$ applying the equations in figure 3.1, recalling from figure 2.1 that the grammar of equality types is $\underset{\text{EQ}}{A} ::= 1 \mid \underset{\text{EQ}}{A} \times \underset{\text{EQ}}{B} \mid \underset{\text{EQ}}{A} + \underset{\text{EQ}}{B} \mid \{\underset{\text{EQ}}{A}\}$. □

It will also be important that, as in our semilattice change structures in **ΔPoset**, changes at a semilattice type L are drawn from the very same type:

Lemma 20. At each semilattice type L, we have $\Delta L = L$.

Proof. Induct on L applying the equations in figure 3.1, recalling from figure 2.1 that the grammar of semilattice types is $L ::= 1 \mid L_1 \times L_2 \mid \{\underset{\text{EQ}}{A}\}$. □

As we'll see in §3.4.3 and 3.4.4, ϕ and δ are mutually recursive. To make this work, δe must find the change to ϕe rather than e. So if $e : A$ then $\phi e : \Phi A$ and $\delta e : \Delta \Phi A$. However, so far we have neglected to say what ϕ and δ do to typing contexts. To understand this, it's helpful to look at what Φ and $\Delta\Phi$ do to functions and to $\Box$. This is because expressions

SPEED-UP TRANSLATION ϕ

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

DERIVATIVE TRANSLATION δ

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

$$
\begin{aligned}
\delta X &= dX & \delta x &= dx \\
\delta(\lambda X.\ e) &= \lambda[x].\ \lambda dX.\ \delta e & \delta(e\ f) &= \delta e\ [\phi f]\ \delta f \\
\delta(e_i)_i &= (\delta e_i)_i & \delta(\pi_i\ e) &= \pi_i\ \delta e \\
\delta(\mathsf{in}_i\ e) &= \mathsf{in}_i\ \delta e & \delta(e \vee f) &= \delta e \vee \delta f \\
\delta[e] &= () & \delta(\textbf{let}\ [x] = e\ \textbf{in}\ f) &= \textbf{let}\ [(x, dx)] = \phi e\ \textbf{in}\ \delta f \\
\delta(\mathsf{empty?}\ e) &= \mathsf{empty?}\ \phi e & \delta(\mathsf{split}\ e) &\overset{\star}{=} \textbf{case}\ \phi e\ \textbf{of}\ ([(\mathsf{in}_i\ \_, \_)] \to \mathsf{in}_i\ ())_i
\end{aligned}
$$

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

Equations marked with a red star, $\overset{\star}{=}$, use pattern-matching syntax sugar we have not previously defined; see figure 3.3 for expansions.

FIGURE 3.2 *ϕ and δ term translations*

ADDITIONAL DESUGARINGS

$$
\begin{aligned}
\phi(\text{split } e) &\overset{\star}{=} \textbf{case } \phi e \textbf{ of} \\
&\qquad ([(\text{in}_i\ x, \text{in}_i\ dx)] \to \text{in}_i\ [(x, dx)])_i \\
&\qquad ([(\text{in}_i\ x, \text{in}_j\ \_)] \to \text{in}_i\ [(x, \text{dummy } x)])_{i\neq j} \\
&= \textbf{let } [z] = \phi e \textbf{ in} \\
&\quad \textbf{case split } [\pi_1\ z] \textbf{ of} \\
&\qquad (\text{in}_i\ Y \to \textbf{let } [x] = Y \textbf{ in} \\
&\qquad\qquad \textbf{case split } [\pi_2\ z] \textbf{ of} \\
&\qquad\qquad\quad \text{in}_i\ DY \to \textbf{let } [dx] = DY \textbf{ in } \text{in}_i\ [(x, dx)] \\
&\qquad\qquad\quad \text{in}_{i+1 \bmod 2}\ \_ \to \text{in}_i\ [(x, \text{dummy } x)])_i
\end{aligned}
$$

$$
\begin{aligned}
\delta(\text{split } e) &\overset{\star}{=} \textbf{case } \phi e \textbf{ of } ([(\text{in}_i\ \_, \_)] \to \text{in}_i\ ())_i \\
&= \textbf{let } [y] = \phi e \textbf{ in case } \pi_1\ y \textbf{ of } (\text{in}_i\ \_ \to \text{in}_i\ ())_{i\in\{1,2\}}
\end{aligned}
$$

$$
\begin{aligned}
\delta(\textbf{case } e \textbf{ of } (\text{in}_i\ X \to f_i)_i) &\overset{\star}{=} \textbf{case split } [\phi e], \delta e \textbf{ of} \\
&\qquad (\text{in}_i\ [x], \text{in}_i\ DX \to \delta f_i)_i \\
&\qquad (\text{in}_i\ [x], \text{in}_j\ \_ \to \textbf{let } DX = \text{dummy } x \textbf{ in } \delta f_i)_{i\neq j} \\
&= \textbf{case split } [\phi e] \textbf{ of} \\
&\qquad (\text{in}_i\ Y \to \textbf{let } [x] = Y \textbf{ in} \\
&\qquad\qquad (\lambda DX.\ \delta f_i)\ (\textbf{case } \delta e \textbf{ of } \text{in}_i\ DX \to DX \\
&\qquad\qquad\qquad\qquad \text{in}_{i+1 \bmod 2}\ \_ \to \text{dummy } x))_i
\end{aligned}
$$

Fresh variables introduced by desugaring are colored pink.

FIGURE 3.3 *Additional syntax sugar for ϕ and δ transformations*

denote functions of their free variables. Moreover, in Datafun free variables come in two flavors, monotone and discrete, and discrete variables are semantically $\Box$-ed.

Viewed as functions of their free variables, δe denotes the *derivative* of ϕe. And just as the derivative of a unary function $f\ x$ has *two* arguments, $df\ x\ dx$, the derivative of an expression e with n variables $x_1, \ldots, x_n$ will have $2n$ variables: the original $x_1, \ldots, x_n$ and their changes $dx_1, \ldots, dx_n$.[9] 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 $\Box$, Φ, and Δ:

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

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

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

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

Theorem 21 (Well-typedness of ϕ, δ)**.** If $\Gamma \vdash e : A$, then ϕe and δe have the following types:

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

Proof. By induction on the derivation of $\Gamma \vdash e : A$, although as we'll see shortly we will need weakening (theorem 22) in some places. □

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

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

To get the hang of these context and type transformations, suppose $x :: A, Y : B \vdash e : C$. Then theorem 21 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, ϕe requires zero change variables dx for every discrete source variable x. Meanwhile, δe requires changes for *every* source program variable (for discrete variables these will be zero changes), and moreover is *discrete* with respect to the source program variables (the "base points").

We now have enough information to tackle the definitions of ϕ and δ given in figure 3.2. In the remainder of this section, we'll examine the most interesting and important parts of these definitions in detail.

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

3.4.2 Fixed points

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

$$\begin{aligned}
\Phi(\Box(\underset{\text{FIX}}{\mathbb{L}} \to \underset{\text{FIX}}{\mathbb{L}})) &= \Box(\Phi(\underset{\text{FIX}}{\mathbb{L}} \to \underset{\text{FIX}}{\mathbb{L}}) \times \Delta\Phi(\underset{\text{FIX}}{\mathbb{L}} \to \underset{\text{FIX}}{\mathbb{L}})) \\
&= \Box((\Phi\underset{\text{FIX}}{\mathbb{L}} \to \Phi\underset{\text{FIX}}{\mathbb{L}}) \times (\Box\Phi\underset{\text{FIX}}{\mathbb{L}} \to \Delta\Phi\underset{\text{FIX}}{\mathbb{L}} \to \Delta\Phi\underset{\text{FIX}}{\mathbb{L}})) \\
&= \Box((\underset{\text{FIX}}{\mathbb{L}} \to \underset{\text{FIX}}{\mathbb{L}}) \times (\Box\underset{\text{FIX}}{\mathbb{L}} \to \Delta\underset{\text{FIX}}{\mathbb{L}} \to \Delta\underset{\text{FIX}}{\mathbb{L}}) && \text{by lemma 19, } \Phi\underset{\text{FIX}}{\mathbb{L}} = \underset{\text{FIX}}{\mathbb{L}} \\
&= \Box((\underset{\text{FIX}}{\mathbb{L}} \to \underset{\text{FIX}}{\mathbb{L}}) \times (\Box\underset{\text{FIX}}{\mathbb{L}} \to \underset{\text{FIX}}{\mathbb{L}} \to \underset{\text{FIX}}{\mathbb{L}}) && \text{by lemma 20, } \Delta\underset{\text{FIX}}{\mathbb{L}} = \underset{\text{FIX}}{\mathbb{L}}
\end{aligned}$$

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

$$\frac{\Gamma \vdash e : \Box((\underset{\text{FIX}}{\mathbb{L}} \to \underset{\text{FIX}}{\mathbb{L}}) \times (\Box\underset{\text{FIX}}{\mathbb{L}} \to \underset{\text{FIX}}{\mathbb{L}} \to \underset{\text{FIX}}{\mathbb{L}})}{\Gamma \vdash \text{semifix } e : \underset{\text{FIX}}{\mathbb{L}}} \qquad [\![\text{semifix } e]\!]\ \gamma = \text{semifix } (f, f') \quad \text{where } (f, f') = [\![e]\!]\ \gamma$$

As for $\delta(\text{fix } e)$, since e can't change (having $\Box$ type), neither can fix e (or semifix ϕe). All we need is a zero change at type $\underset{\text{FIX}}{\mathbb{L}}$; by lemma 20, $\bot$ suffices.

3.4.3 Variables, λ-abstraction, and application

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

$$\begin{aligned}
\Delta\Phi(A \to B) &= \Box\Phi A \to \Delta\Phi A \to \Delta\Phi B \\
\delta(\lambda X.\ e) &= \lambda[x].\ \lambda DX.\ \delta e \\
\delta(e\ f) &= \delta e\ [\phi f]\ \delta f
\end{aligned}$$

The intuition behind $\delta(\lambda X.\ e) = \lambda[x].\ \lambda DX.\ \delta e$ is that a function change takes two arguments, a base point x and a change DX, and yields the change in the result of the function, δe. However, we are given an argument of type $\Box\Phi A$, but consulting theorem 21 for the type of δe, we need a discrete variable $x :: \Phi A$, so we use pattern-matching to unbox our argument. The intuition behind $\delta(e\ f) = \delta e\ [\phi f]\ \delta f$ is much the same: δe needs two arguments, the original input ϕf and its change δf, to return the change in the function's output. Moreover, it's discrete in its first argument, so we need to box it, $[\phi f]$.

One might ask why this type-checks, since ϕe and δe don't use the same typing context. We're even boxing ϕf, hiding all monotone variables; consequently, it gets the context $\lceil \Box\Phi\Gamma, \Delta\Phi\Gamma \rceil$. However, $\Box$ makes every variable discrete, and $\lceil - \rceil$ leaves discrete variables alone, so this provides *at least* $\Box\Phi\Gamma$, while the context ϕf needs is $\Phi\Gamma$. Thus really this is a question about the interaction of weakening and discreteness: can a discrete variable always substitute for a monotone one?

Indeed it may: making a variable discrete only increases the number of places it can be used, because while some typing rules discard monotone variables, they never discard discrete ones. We formalize this using a weakening relation $\Gamma \sqsubseteq \Delta$ (figure 3.4; note that H

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

FIGURE 3.4 *Weakening relation*

for "hypothesis" ranges over all variable typings, monotone or discrete), which is standard except for the rule DISC, which says that a discrete hypothesis is weaker than a monotone one. We can then show that typing respects weakening:

Theorem 22 (Weakening)**.** If $\Delta \sqsupseteq \Gamma$ and $\Gamma \vdash e : A$ then $\Delta \vdash e : A$.

Proof. By induction on the derivation of $\Gamma \vdash e : A$; see appendix A.2. □

We use this without further note throughout the ϕ and δ transformations.

3.4.4 The discreteness comonad, □

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

Next, consider $\phi[e] = [(\phi e, \delta e)]$. The intuition here is straightforward: ϕ needs to decorate e with its zero change; since e is discrete and cannot change, we use δe. However! In general, one cannot use δ inside the ϕ translation and expect the result to be well-typed; ϕ and δ require different typing contexts. To see this, let's apply theorem 21 to singleton contexts:

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

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

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

$$\phi(\textbf{let}\ [x] = e\ \textbf{in}\ f) = \textbf{let}\ [(x, dx)] = \phi e\ \textbf{in}\ \phi f$$
$$\delta(\textbf{let}\ [x] = e\ \textbf{in}\ f) = \textbf{let}\ [(x, dx)] = \phi e\ \textbf{in}\ \delta f$$

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

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

FIGURE 3.5 *The function* $\mathit{dummy}_A : A \to \Delta A$

3.4.5 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 ϕe evaluates to $\mathrm{in}_i\ x$ and its change δe evaluates to $\mathrm{in}_j\ dx$. Recall that sums are ordered disjointly (§2.3.2); the value x can increase, but the tag in_i must remain the same. Since δe is a change to ϕe, the change structure on sums (§3.3.2) tells us that $i = j$! So the desired change $\delta(\textbf{case}\ e\ \textbf{of}\ \ldots)$ is given by δf_i in a context supplying a discrete base point x (the value x) and the change DX. To bind x discretely, we need to use $[\phi e] : \Box(\Phi A + \Phi B)$; to pattern-match on this, we need split to distribute the $\Box$.

This handles the first two cases, $(\mathrm{in}_i\ [x],\ \mathrm{in}_i\ DX \to \delta f_i)_i$. Since we know the tags on ϕe and δe agree, these are the only possible cases. However, since the output of our translation is Datafun code, to appease the 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 3.5).

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

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

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

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

3.4.6 Semilattices and comprehensions

The translation $\phi(e \vee f) = \phi e \vee \phi f$ is straightforward, but $\delta(e \vee f) = \delta e \vee \delta f$ is not as simple as it seems. Restricting to sets, suppose that dx changes x into x' and dy changes y to y'. In particular, suppose these changes are *precise*: that $dx = x' \setminus x$ and $dy = y' \setminus y$. Then the precise change from $x \cup y$ into $x' \cup y'$ is:

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

This suggests letting $\delta(e \cup f) = (\delta e \setminus \phi f) \cup (\delta f \setminus \phi e)$. This is a valid derivative, but it involves recomputing ϕe and ϕf, and our goal is to avoid recomputation. So instead, we *overapproximate* the derivative: $\delta e \cup \delta f$ might contain some unnecessary elements, but we expect it to be cheaper to include these than to recompute ϕe and ϕf. This overapproximation

agrees with seminaïve evaluation in Datalog: Datalog implicitly unions the results of different rules for the same predicate (e.g. those for path in §3.1), and the seminaïve translations of these rules do not include negated premises to compute a more precise difference.

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

Lemma 23. If $x : \underset{\text{EQ}}{A}$ then dummy $x \rhd x \hookrightarrow x : \underset{\text{EQ}}{A}$.

Proof. By induction on $\underset{\text{EQ}}{A}$, unfolding the definition of $(\text{dummy } dx \rhd x \hookrightarrow x : \underset{\text{EQ}}{A})$ from §3.2 at each step. For example, when $\underset{\text{EQ}}{A} = \{\underset{\text{EQ}}{B}\}$, we need to show that $x \cup \text{dummy } dx = x$, which is true because $\text{dummy}_{\{\underset{\text{EQ}}{B}\}}\ x = \{\}$. □

For clarity, we write $\mathbf{0}$ rather than dummy when we use it to produce zero changes; we only call it dummy in dead code.

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

3.4.7 Leftovers

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

3.5 Proving the seminaïve transformation correct

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

So, inductively on types A, letting $a, b \in [\![A]\!]$, $x, y \in [\![\Phi A]\!]$, and $dx \in [\![\Delta\Phi A]\!]$, we define a five place relation $dx \rhd_A x \not\downarrow a \to y \not\downarrow b$, meaning roughly "$x, y$ speed up a, b respectively, and dx changes x into y". The full definition is in figure 3.6.

$$
\begin{aligned}
() \rhd_1 () \lightning () \to () \lightning () &\iff \top \\
\vec{dx} \rhd_{A_1 \times A_2} \vec{x} \lightning \vec{a} \to \vec{y} \lightning \vec{b} &\iff (\forall i)\ dx_i \rhd_{A_i} x_i \lightning a_i \to y_i \lightning b_i \\
in_i\ dx \rhd_{A_1+A_2} in_j\ x \lightning in_k\ a \to in_l\ y \lightning in_m\ b &\iff i = j = k = l = m \wedge dx \rhd_{A_i} x \lightning a \to y \lightning b \\
df \rhd_{A\to B} f \lightning f' \to g \lightning g' &\iff (\forall dx \rhd_A x \lightning a \to y \lightning b) \\
&\qquad df\ x\ dx \rhd_B f\ x \lightning f'\ a \to g\ y \lightning g'\ b \\
dx \rhd_{\underset{EQ}{\{A\}}} x \lightning a \to y \lightning b &\iff (x, y, x \cup dx) = (a, b, y) \\
() \rhd_{\Box A} (x, dx) \lightning a \to (y, dy) \lightning b &\iff (a, x, dx) = (b, y, dy) \wedge dx \rhd_A x \lightning a \to y \lightning b
\end{aligned}
$$

FIGURE 3.6 *Definition of the logical relation*

At product, sum, and function types this is essentially a more elaborate version of the change structures given in §3.2. At set types, changes are still a set of values added to the initial value, but we additionally insist that the "slow" a, b and "speedy" x, y are equal. This is because we have engineered the definitions of Φ and ϕ to preserve behavior on equality types. Finally, since $\Box A$ represents values which cannot change, dx is an uninformative empty tuple and the original and updated values are identical. However, the "speedy" values are now *pairs* of a value and its zero change. This ensures that at a boxed function type, we will always have a derivative (a zero change) available.

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

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

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

Theorem 24 (Fundamental property)**.** If $\Gamma \vdash e : A$ and $d\gamma \rhd_\Gamma \gamma \lightning \rho \to \gamma' \lightning \rho'$ then

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

Proof. See appendix A.2. □

This theorem follows by a structural induction on typing derivations as usual, but requires a number of lemmas. By and large, these lemmas generalize or build on results stated earlier regarding the simpler change structures from §3.2. For example, we build on lemmas 19 and 23 to characterize the logical relation at equality types $\underset{EQ}{A}$ and the behavior of dummy:

Lemma 25 (Equality changes)**.** If $dx \rhd_{\underset{EQ}{A}} x \lightning a \to y \lightning b$ then $x = a$ and $y = b$.

Lemma 26 (Dummy is zero at eqtypes)**.** If $x \in [\![\underset{EQ}{A}]\!]$ then $\text{dummy}\ x \rhd_{\underset{EQ}{A}} x \lightning x \to x \lightning x$.

Proof. In each case, induct on $\underset{EQ}{A}$. See appendix A.2. □

Lemma 25 tells us that at equality types, the sped-up version of a value is the value itself. This is used later to prove our adequacy theorem. Lemma 26 is an analogue of lemma 23, showing that dummy function computes zero changes at equality types. This is used in the proof of the fundamental theorem for **for**-loops, in whose ϕ and δ translations 0 is implemented by dummy.

Next, we generalize lemma 20 to characterize changes at semilattice type:

Lemma 27 (Semilattice changes)**.** At any semilattice type L, we have $\Delta L = L$, and moreover $dx \triangleright_L x \not\downarrow a \to y \not\downarrow b$ iff $x = a$ and $y = b = x \vee_L dx$

Proof. By induction on semilattice types L, applying lemma 25 (noting that every semilattice type is an equality type). □

We require this lemma in the proofs of the fundamental theorem in all the cases involving semilattice types – namely $\bot$, $\vee$, **for**, and fix.

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

Lemma 28 (Discrete contexts don't change)**.** If $() \triangleright_{\lceil\Gamma\rceil} \gamma \not\downarrow \rho \to \gamma' \not\downarrow \rho'$ then $\gamma = \gamma'$ and $\rho = \rho'$.

Proof. All variables in the stripped contexts are discrete, and therefore the logical relation for discrete variables in contexts, which invokes the logical relation at $\Box$ type, requires their corresponding components be equal. □

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

Lemma 29 (Context stripping)**.** If $d\gamma \triangleright_\Gamma \gamma \not\downarrow \rho \to \gamma' \not\downarrow \rho'$ then

$$() \triangleright_{\lceil\Gamma\rceil} \mathsf{strip}_{\Phi\Gamma}(\gamma) \not\downarrow \mathsf{strip}_\Gamma(\rho) \to \mathsf{strip}_{\Phi\Gamma}(\gamma') \not\downarrow \mathsf{strip}_\Gamma(\rho')$$

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

Proof. Immediate from the definitions. □

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

Combining all these lemmas to establish the fundamental theorem, adequacy follows immediately:

Theorem 30 (Adequacy)**.** If $\varepsilon \vdash e : \underset{\text{EQ}}{A}$ then $[\![e]\!] = [\![\phi e]\!]$.

Proof. By theorem 24, noting the premise is trivial since the context is empty, we have $[\![\delta e]\!] \triangleright_{\underset{\text{EQ}}{A}} [\![\phi e]\!] \not\downarrow [\![e]\!] \to [\![\phi e]\!] \not\downarrow [\![e]\!]$, which by lemma 25 implies $[\![\phi e]\!] = [\![e]\!]$. □

Chapter 4

Implementation and Efficiency

The previous chapter was entirely theoretical, formalizing the intuition that seminaïve evaluation works by computing the changes between iterations toward a fixed point by, first, constructing a theory of changes for Datafun; and second, applying that theory to construct and prove correct a program transformation which implements this strategy. However, the purpose of seminaïve evaluation is not to push changes around, but to compute results faster. We have proven that our transformed program computes the same result, but not shown that it does so more efficiently. In this chapter we remedy this experimentally, observing that at least two further optimizations are necessary to achieve asymptotic performance improvements.

First, in §4.1 we apply the seminaïve program transformation by hand to our running example, transitive closure. In the process we uncover some obvious inefficiencies in the transformed code and demonstrate how to optimize them away. In §4.2 we discuss our implementation of a Datafun-to-Haskell compiler, which we use to demonstrate experimentally that seminaïve evaluation can produce asymptotic performance improvements when combined with these optimizations.

Second, in §4.3 we observe that even with these optimizations, there remain cases where we do asymptotically more work than necessary, not because of inefficiencies in the transformed program, but because of the imprecision of our derivatives. This results in overly large changes which accumulate across fixed point iterations. We implement a simple solution based on change minimization and test it experimentally.

4.1 Applying the seminaïve transformation to transitive closure

Let's try applying the seminaïve transform to a simple Datafun program: the transitive closure function trans from §2.2.1:

$$\begin{aligned}&\mathsf{trans}\ [\mathsf{edge}] = \textbf{fix}\ \mathsf{R}\ \textbf{is}\ \mathsf{edge} \cup (\mathsf{edge} \bullet \mathsf{R})\\ &\mathsf{S} \bullet \mathsf{T} = \textbf{for}\ ((x, y_1) \in \mathsf{S})\ \textbf{for}\ ((y_2, z) \in \mathsf{T})\ \textbf{for}\ (y_1 = y_2)\ \{(x, z)\}\end{aligned}$$

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

In our experience, performing ϕ and δ by hand is easiest done from the inside outwards. At the core of transitive closure is a relation composition, $(e \bullet p)$, and at the core of relation composition is a "one-sided conditional", $\textbf{for}\ (y_1 = y_2)\ \{(x, z)\}$. Let's take a look at its ϕ and δ translations:

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

Frequently, as in this case, ϕ does nothing interesting. For brevity we'll skip such no-op translations. Now for the δ translation:

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

Thus:

$$\delta(\textbf{for}\ (y_1 = y_2)\ \{(x,z)\}) = \bot \tag{4.1}$$

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

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

$$
\begin{aligned}
&\delta(\textbf{for}\ ((y_2,z) \in t)\ \textbf{for}\ (y_1 = y_2)\ \{(x,z)\}) \\
&= \quad \textbf{for}\ ((y_2,z) \in dt)\ \phi(\textbf{for}\ (y_1 = y_2)\ \{(x,z)\}) && \text{apply } \delta \text{ and omit unused } \textbf{let}\text{s} \\
&\quad \cup \textbf{for}\ ((y_2,z) \in t \cup dt)\ \delta(\textbf{for}\ (y_1 = y_2)\ \{(x,z)\}) \\
&= \textbf{for}\ ((y_2,z) \in dt)\ \textbf{for}\ (y_1 = y_2)\ \{(x,z)\} && \text{use equation 4.1, propagate } \bot
\end{aligned}
$$

Thus:

$$\delta(\textbf{for}\ ((y_2,z) \in t)\ \textbf{for}\ (y_1 = y_2)\ \{(x,z)\}) = \textbf{for}\ ((y_2,z) \in dt)\ \textbf{for}\ (y_1 = y_2)\ \{(x,z)\} \tag{4.2}$$

Tackling the outermost **for** loop:

$$
\begin{aligned}
&\delta(\textbf{for}\ ((x,y_1) \in s)\ \textbf{for}\ ((y_2,z) \in t)\ \textbf{for}\ (y_1 = y_2)\ \{(x,z)\}) \\
&= \textbf{for}\ ((x,y_1) \in ds)\ \phi(\textbf{for}\ ((y_2,z) \in t)\ \textbf{for}\ (y_1 = y_2)\ \{(x,z)\}) && \text{apply } \delta(\textbf{for}\ldots) \\
&\cup \textbf{for}\ ((x,y_1) \in s \cup ds)\ \delta(\textbf{for}\ ((y_2,z) \in t)\ \textbf{for}\ (y_1 = y_2)\ \{(x,z)\}) \\
&= \textbf{for}\ ((x,y_1) \in ds)\ \textbf{for}\ ((y_2,z) \in t)\ \textbf{for}\ (y_1 = y_2)\ \{(x,z)\} && \text{use equation 4.2} \\
&\cup \textbf{for}\ ((x,y_1) \in s \cup ds)\ \textbf{for}\ ((y_2,z) \in dt)\ \textbf{for}\ (y_1 = y_2)\ \{(x,z)\} \\
&= (ds \bullet t) \cup ((s \cup ds) \bullet dt) && \text{rewrite using } \bullet
\end{aligned}
$$

This, then, is the derivative of relation composition:

$$\delta(s \bullet t) = (ds \bullet t) \cup ((s \cup ds) \bullet dt) \tag{4.3}$$

Distributing composition over union, this is equivalent to $(ds \bullet t) \cup (s \bullet dt) \cup (ds \bullet dt)$, which is perhaps the derivative a human would give.

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

$$\begin{aligned}
&\delta(e \cup (e \bullet p)) \\
&= \delta e \cup \delta(e \bullet p) \\
&= \delta e \cup (\delta e \bullet p) \cup ((e \cup \delta e) \bullet dp) && \text{use equation 4.3} \\
&= \bot \cup (\bot \bullet p) \cup ((e \cup \bot) \bullet dp) && \delta e \text{ is a zero change; replace with } \bot \\
&= e \bullet dp && \text{propagate } \bot
\end{aligned}$$

Note that the penultimate step here requires a new optimization: by definition $\delta e = de$, but since e is discrete we know de is a zero change, so we may safely replace it by $\bot$, as it will have the same effect. Thus:

$$\delta(e \cup (e \bullet p)) = e \bullet dp \tag{4.4}$$

Putting everything together, we have:

$$\begin{aligned}
&\phi(\textbf{fix}\ P\ \textbf{is}\ e \cup (e \bullet p)) \\
&= \phi(\text{fix}\ [\lambda p.\ e \cup (e \bullet p)]) && \text{desugaring} \\
&= \text{semifix}\ \phi[\lambda p.\ e \cup (e \bullet p)] \\
&= \text{semifix}\ \left[(\phi(\lambda p.\ e \cup (e \bullet p)),\ \delta(\lambda p.\ e \cup (e \bullet p)))\right] \\
&= \text{semifix}\ \left[((\lambda p.\ e \cup (e \bullet p)),\ (\lambda[p].\ \lambda dp.\ e \bullet dp))\right] && \text{use equation 4.4}
\end{aligned}$$

Examining the recurrence produced by this use of semifix, we recover the seminaïve transitive closure algorithm from §3.1:

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

4.2 Implementation

To put the seminaïve transformation presented in chapter 3 to the test, we have implemented it as part of a compiler and runtime for a fragment of Datafun (omitting sum types and pattern-matching), available at `https://github.com/rntz/datafun/tree/popl20/v4-fastfix`. In §4.2.1 we sketch the compiler's front-to-back structure, from Datafun source code through several intermediate stages to Haskell output. In §4.2.2 we explore how an example Datafun program gets compiled, exhibit the small auxiliary library necessary to run the compiled outputs, and explain why we chose Haskell as a target language. Finally, in §4.2.3 we perform some simple benchmarks to test for the expected efficiency gains.

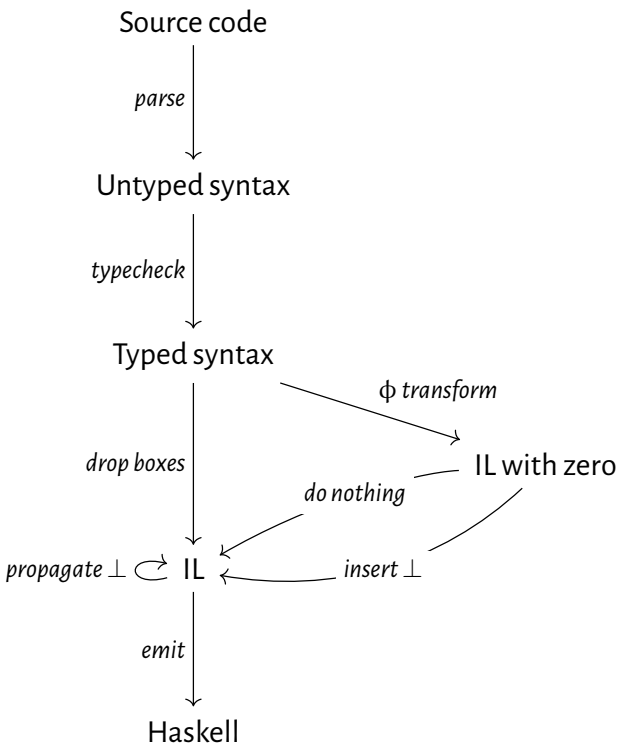


FIGURE 4.1 *Stages of the Datafun compiler*

4.2.1 The compiler structure

The compiler proceeds in several passes, shown as arrows in figure 4.1. It begins with fairly standard parsing and typechecking phases. For parsing, we use OCaml's Menhir library.[1] The typechecker is bidirectional (Pierce and Turner, 2000) rather than performing full inference, for simplicity of implementation. After these, we have a choice of paths.

The simplest approach is to translate Datafun fairly directly into Haskell. Many of Datafun's features have direct parallels in Haskell, including sum, product, and function types; Datafun's sets are implemented using Haskell's `Data.Set` module; Datafun's semilattice types are implemented as a Haskell typeclass; and Datafun's fixed points can be implemented by naïve bottom-up iteration. (We discuss these implementation details in §4.2.2.)

One feature of Datafun that does not translate straightforwardly into Haskell is the discreteness comonad $\Box$. However, its only purpose is to track (non-)monotonicity; at runtime, $\Box A$ may as well be A. Thus before emitting Haskell we *"drop boxes"*, eliminating $\Box$ and its related term syntax by rewriting $[e] \rightsquigarrow e$ and $\textbf{let}\ [x] = e\ \textbf{in}\ f \rightsquigarrow \textbf{let}\ x = e\ \textbf{in}\ f$ and dropping the distinction between discrete and monotone variables. This takes us to an intermediate language, unimaginatively dubbed "IL", supporting Datafun's computational features but lacking its modal typing, which we translate directly into Haskell.

Thus naïve compilation of Datafun code takes a straight path through figure 4.1:

$$\text{Source} \xrightarrow{\textit{parse}} \text{Syntax} \xrightarrow{\textit{typecheck}} \text{Typed syntax} \xrightarrow{\textit{drop boxes}} \text{IL} \xrightarrow{\textit{emit}} \text{Haskell}$$

For seminaïve evaluation, however, we must apply the ϕ *transform* from chapter 3 to the

[1] http://gallium.inria.fr/~fpottier/menhir/

typed syntax before dropping boxes, since the transform works precisely by annotating boxed terms with their zero-changes. As we have no further use for $\Box$, the compiler pass implementing ϕ omits boxes from its output. Thus after applying the ϕ transform we can directly emit Haskell.

As we saw in §4.1, however, the ϕ transform produces code that can often be simplified by replacing certain expressions by $\bot$. Therefore we implement two optimization passes:

1. We *propagate* $\bot$ by applying the following rewrites to the IL:

$$\begin{aligned} e \vee \bot &\rightsquigarrow e & \bot \vee e &\rightsquigarrow e \\ \textbf{for}\ (x \in e)\ \bot &\rightsquigarrow \bot & \textbf{for}\ (x \in \bot)\ e &\rightsquigarrow \bot \\ \textbf{let}\ x = \bot\ \textbf{in}\ e &\rightsquigarrow e\{x \mapsto \bot\} & \textbf{let}\ x = e\ \textbf{in}\ \bot &\rightsquigarrow \bot \\ (\bot, \bot) &\rightsquigarrow \bot & \pi_i\ \bot &\rightsquigarrow \bot \end{aligned}$$

   The most important rewrite here is $\textbf{for}\ (x \in e)\ \bot \rightsquigarrow \bot$: to evaluate the left hand side directly, we evaluate e and iterate over the resulting set, which takes work proportional to its size; but evaluating the right hand side takes constant work. This is where the asymptotic speedups originate. The other rewrites are useful primarily because they enable this one.

2. To make $\bot$-propagation more effective, we first *insert* $\bot$ in place of semilattice-valued zero changes. To this end, the ϕ *transform* pass explicitly marks certain terms produced by δ which are guaranteed to be zero-changes, namely:

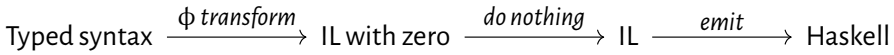
$$\begin{aligned} \delta x &= dx \quad \text{(for discrete } x\text{)} \\ \delta[e] &= () \\ \delta\{e_i\}_i = \delta(e_1 = e_2) = \delta(\text{fix } e) = \delta\bot &= \bot \end{aligned}$$

   The *insert* $\bot$ pass replaces these marked expressions with $\bot$ when they have semilattice type, along with some compound terms guaranteed to produce zero-changes:

$\textbf{let}\ x = e\ \textbf{in}\ f$	when f is a zero change
x	when x is let-bound to a zero change
$e_1\ e_2\ e_3$	when e_1, e_3 are both zero changes

   In case it is not clear, the last case corresponds to the derivative of function application, $\delta(e\ f) = \delta e\ [\phi f]\ \delta f$, when neither the function nor the argument are changing.

To test whether *insert* $\bot$ is actually useful, we also implement a *do nothing* pass for comparison, which simply ignores the zero-change annotations produced by the ϕ transform. Altogether, we have three new paths through the compiler. First, after parsing and typechecking we we can apply the seminaïve transform without further optimizations:



Second, we can optimize the output of the ϕ transform by propagating $\bot$:

$$\text{Typed syntax} \xrightarrow{\phi\ \textit{transform}} \text{IL with zero} \xrightarrow{\textit{do nothing}} \text{IL} \xrightarrow{\textit{propagate}\ \bot} \text{IL} \xrightarrow{\textit{emit}} \text{Haskell}$$

Or finally, we can replace semilattice zero changes with $\bot$ to make $\bot$ propagation more effective:

$$\text{Typed syntax} \xrightarrow{\phi\ \textit{transform}} \text{IL with zero} \xrightarrow{\textit{insert}\ \bot} \text{IL} \xrightarrow{\textit{propagate}\ \bot} \text{IL} \xrightarrow{\textit{emit}} \text{Haskell}$$

4.2.2 Compiling transitive closure

To understand the compiler's behavior more concretely, let's consider what it does to a Datafun program implementing transitive closure:

$$\begin{aligned}&\text{trans} : \square\{\underset{\text{EQ}}{A} \times \underset{\text{EQ}}{A}\} \to \{\underset{\text{EQ}}{A} \times \underset{\text{EQ}}{A}\}\\&\text{trans}\ [\text{edge}] = \textbf{fix}\ P\ \textbf{is}\ \text{edge} \vee \{(x, z) \mid (x, y) \in \text{edge},\ (!y, z) \in P\}\end{aligned}$$

This code needs a few changes for the compiler to accept it. First, we must replace $\underset{\text{EQ}}{A}$ with a specific, concrete type, as the compiler does not support polymorphism. Second, the compiler does not support pattern-matching, so we must replace box-patterns with let-unboxing, tuple-patterns with projections, and equality patterns $!y$ with equality tests:

$$\begin{aligned}&\text{trans} : \square\{\text{string} \times \text{string}\} \to \{\text{string} \times \text{string}\}\\&\text{trans} = \lambda E.\\&\quad \textbf{let}\ [\text{edge}] = E\ \textbf{in}\\&\quad \textbf{fix}\ P\ \textbf{is}\\&\qquad \text{edge} \vee \{(\pi_1\ a, \pi_2\ b) \mid a \in \text{edge},\ b \in P,\ \pi_2\ a = \pi_1\ b\}\end{aligned}$$

In the ASCII syntax the compiler accepts, this becomes:

```
@([{str,str}] -> {str,str}) -- type annotation
\e.
  let [edge] = e in
  fix p is
    edge or {(pi1 a, pi2 b) for a in edge for b in p when pi2 a = pi1 b}
```

Passing this through the naïve compilation path, *parse* → *typecheck* → *drop boxes* → *emit*, produces closely corresponding Haskell code (also shown in figure 4.4):

```
\e_0 ->
  let edge_1 = e_0 in
  fix (\p_2 ->
    union edge_1
      (forIn edge_1 (\a_3 ->
       forIn p_2 (\b_4 ->
       guard (snd a_3 == fst b_4)
       (set [(fst a_3, snd b_4)])))))
```

This code has been prettified by removing unnecessary parentheses and adding line breaks and indentation. Besides minor syntactic details, the primary changes are:

1. Variable names have had unique numeric suffixes added; this is an artifact of the compiler internals.

2. Boxes have been dropped, so `let [edge] = e` becomes simply **`let`** `edge_1 = e_0`.

3. The set comprehension is translated into nested calls to `forIn`, `guard`, and `set`. This uglifies the code, replacing binding forms with higher-order functions applied to lambdas, but otherwise corresponds to desugaring the comprehension (figure 2.2):

$$\{(\pi_1\ a, \pi_2\ b) \mid a \in \text{edge},\ b \in P,\ \pi_2\ a = \pi_1\ b\}$$
$$\xrightarrow{\text{desugar}} \textbf{for}\ (a \in \text{edge})\ \textbf{for}\ (b \in P)\ \textbf{for}\ (\_ \in \pi_2\ a = \pi_1\ b)\ \{(\pi_1\ a, \pi_2\ b)\}$$

4. In a similar way, other Datafun features such as fixed points **fix** X **is** e and semilattice join $e_1 \vee e_2$ have been translated into calls to functions like `fix` and `union`.

The resulting Haskell code depends on functions (highlighted in pink) supplied by a small auxiliary library (figure 4.2). The primary reason we chose Haskell as a target language is because it simplifies this step of translating language features into calls to library functions: Datafun's concept of a "semilattice type" L, over which $\vee$, $\bot$, fix, and **for**-loops are parameterized, translates directly to a Haskell typeclass `Semilat` defined in this auxiliary library. This spares the Datafun compiler the work of determining at which types these primitives are actually invoked and generating the library code necessary for that subset of types.

This `Semilat` typeclass has two core methods, `(<:)` and `unions`:[2]

```
class Semilat a where
  (<:) :: a -> a -> Bool
  unions :: [a] -> a
  ... etc ...
```

The `(<:)` operator computes the type's order relation $\leqslant$,[3] and is used to determine when fixed point iteration has stabilized: seminaïve iteration stabilizes once $dx_i \leqslant x_i$. (This is an optimization over checking equality $x_i = x_{i+1}$.) The `unions` method computes the semilattice join/least upper bound of its list of arguments. Using this we define binary `union`/$\vee$ and nullary `empty`/$\bot$ wrappers. We provide instances for empty and binary products, booleans, and sets, the Haskell types which our Datafun semilattice types get translated into.

The other functions provided by the runtime library are `set`, used to construct literal sets; `forIn`, which implements **for**-loops over sets; `guard`, which implements **for**-loops over booleans (i.e. one-sided conditionals **for** $(e_1)\ e_2$, also seen as boolean "guard" conditions in set comprehensions); and `fix`/`semifix`, which implement naïve and seminaïve iteration respectively.[2]

4.2.3 Benchmarking seminaïve evaluation

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

[2] The `diff` method on `Semilat` and the `semifixMinimized` function that uses it will be explained in §4.3.

[3] We did not use Haskell's built-in `Ord` typeclass for this because it is intended for total orders, not partial ones.

```
module Runtime (Set, Semilat (..), set, guard, forIn,
                fix, semifix, semifixMinimized) where
import qualified Data.Set as Set
import Data.Set (Set)

class Semilat a where
  (<:) :: a -> a -> Bool
  unions :: [a] -> a
  empty :: a
  empty = unions []
  union :: a -> a -> a
  union x y = unions [x,y]
  diff :: a -> a -> a -- Law: union a (diff da a) = union a da
  diff dx x = dx -- always lawful but not always efficient

instance Semilat () where
  () <: () = True
  unions _ = ()
instance (Semilat a, Semilat b) => Semilat (a,b) where
  (a,x) <: (b,y) = a <: b && x <: y
  unions ts = (unions lefts, unions rights) where (lefts, rights) = unzip ts
  diff (da,db) (a,b) = (diff da a, diff db b)
instance Semilat Bool where
  x <: y = not x || y
  unions = or
instance Ord a => Semilat (Set a) where
  (<:) = Set.isSubsetOf
  unions = Set.unions
  diff = Set.difference

set :: Ord a => [a] -> Set a
set = Set.fromList

guard :: Semilat a => Bool -> a -> a
guard c x = if c then x else empty

forIn :: Semilat b => Set a -> (a -> b) -> b
forIn set f = unions [f x | x <- Set.toList set]

fix :: Semilat a => (a -> a) -> a
fix f = loop empty
  where loop x = if x' <: x then x else loop x'
        where x' = f x

semifix, semifixMinimized :: Semilat a => ((a -> a), (a -> a -> a)) -> a
semifix (f, df) = loop empty (f empty)
  where loop x dx = if dx <: x then x else loop (union x dx) (df x dx)
semifixMinimized (f, df) = loop empty (f empty)
  where loop x dx = if dx <: x then x else
                    let x' = union x dx in loop x' (df x dx `diff` x')
```

FIGURE 4.2 *Datafun's runtime library*

1. Finding the transitive closure of a linear graph using the trans function from §4.2.2 (first introduced in §2.2.1). As discussed in §3.1, transitive closure has a well understood asymptotic speed-up under seminaïve evaluation. This means that if we've failed to capture the essence of seminaïve evaluation, it should be highly visible.

2. Finding all matches of the regular expression `/a*/` in the string a^n, using the regex combinators from §2.2.2. Finding all matches for `/a*/` amounts to finding the reflexive, transitive closure of the matches of `/a/`, and on a^n these form a linear graph. Thus this is a close analogue of our first example, but written in a higher-order style, as we represent regular expressions as functions and regex constructors as function combinators. We chose this example to test whether our extension of seminaïve evaluation properly handles Datafun's distinctive feature: higher-order programming.

We send each program through the four compiler paths described in §4.2.1: the direct/naïve translation (*naïve*); the ϕ transform alone (*seminaïve raw*); ϕ with $\bot$ propagation (*seminaïve simplified*); and ϕ with $\bot$ insertion and propagation (*seminaïve optimized*). We exhibit the four translations of trans in figures 4.4–4.7. We graph the benchmark results in figure 4.3, separately showing the running times as well as the speedup factor over naïve evaluation (on a logarithmic plot). The results support two conclusions:

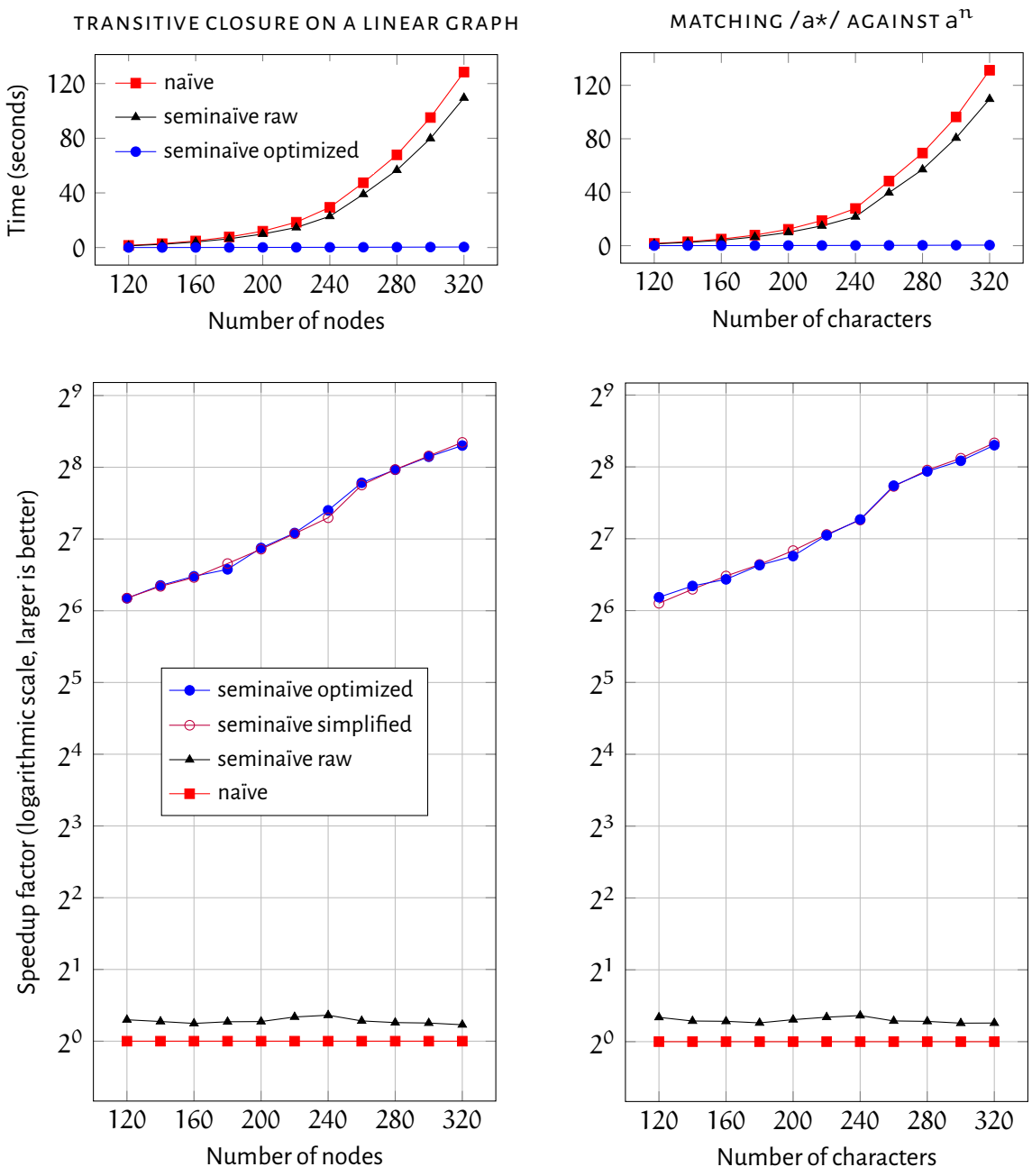
1. The ϕ transformation combined with optimizations enables asymptotic speedups: *seminaïve optimized* is dramatically faster than *naïve,* and the speedup factor increases with the input size.[4] Moreover, the measured times are similar for transitive closure and regex search across all optimization levels, suggesting that higher-order code does not pose a particular problem for our optimizations.

2. These asymptotic improvements depend on $\bot$ propagation: *seminaïve raw* yields only a small constant-factor speedup over *naïve*, roughly 20%. However, the measured times for *seminaïve simplified* and *seminaïve optimized* are effectively identical, so $\bot$ insertion appears to be irrelevant, even (somewhat surprisingly) in higher-order code.

As alluded to in the previous section, these asymptotic speedups come from avoiding wasteful loops $(\textbf{for}\ (x \in e_1)\ e_2)$ where e_1 grows as our input grows but e_2 always produces $\bot$. The ϕ/δ translations alone do not accomplish this: both $\phi(\textbf{for}\ (x \in e)\ \ldots)$ and $\delta(\textbf{for}\ (x \in e)\ \ldots)$ produce loops that iterate over at least every $x \in \phi e$. Consulting our logical relation (figure 3.6) at set type, we see that e and ϕe compute identical sets, therefore the number of iterations never shrinks. For instance, in the *seminaïve raw* translation of trans (figure 4.5) the derivative passed to `semifix` contains the following wasteful loop:

```
forIn (union p_2 dp_2) (\b_4 ->
  let db_4 = ((), ()) in
  if (snd a_3 == fst b_4) then set [] else
  guard False
  (union (set [(fst a_3, snd b_4)]) (set [])))
```

[4] Although faster than naïve evaluation, *seminaïve optimized* is still asymptotically quite slow in these benchmarks. On transitive closure, for example, doubling the graph size from 160 to 320 nodes yields a slowdown factor of $\frac{.407}{.054} \approx 7.54$! However, since there are quadratically many paths and we find all of them, the best possible runtime is $O(n^2)$. Moreover, our nested-loop relational joins are roughly a factor of n slower than optimal, so we expect $O(n^3)$ behavior, which predicts a slowdown of $2^3 = 8$, reasonably close to 7.54.



	GRAPH SIZE / STRING LENGTH										
	120	140	160	180	200	220	240	260	280	300	320
REGEX SEARCH, NAÏVE	1.665	2.940	4.924	7.872	12.251	18.789	27.828	48.393	69.337	96.371	131.300
TRANSITIVE CLOSURE, NAÏVE	1.568	2.843	4.797	7.756	11.944	18.521	29.415	47.446	67.845	95.142	128.403
REGEX SEARCH, SEMINAÏVE RAW	1.317	2.410	4.047	6.568	9.909	14.840	21.636	39.629	57.017	80.622	109.707
TRANSITIVE CLOSURE, SEMINAÏVE RAW	1.275	2.351	4.040	6.429	9.880	14.656	22.886	39.007	56.686	79.837	109.552
REGEX SEARCH, SEMINAÏVE SIMPLIFIED	0.024	0.037	0.055	0.079	0.107	0.141	0.182	0.228	0.279	0.347	0.407
TRANSITIVE CLOSURE, SEMINAÏVE SIMPLIFIED	0.022	0.035	0.054	0.077	0.103	0.138	0.187	0.220	0.271	0.333	0.395
REGEX SEARCH, SEMINAÏVE OPTIMIZED	0.023	0.036	0.057	0.079	0.113	0.142	0.181	0.227	0.283	0.355	0.416
TRANSITIVE CLOSURE, SEMINAÏVE OPTIMIZED	0.022	0.035	0.054	0.081	0.102	0.137	0.174	0.216	0.272	0.336	0.407

FIGURE 4.3 *Naïve vs seminaïve evaluation of transitive closure and regex matching in Datafun*

```
\e_0 ->
  let edge_1 = e_0 in
  fix (\p_2 ->
    union edge_1
      (forIn edge_1 (\a_3 ->
       forIn p_2 (\b_4 ->
       guard (snd a_3 == fst b_4)
       (set [(fst a_3, snd b_4)])))))
```

FIGURE 4.4 *Naïve translation of transitive closure*

```
\e_0 ->
  let (edge_1, dedge_1) = e_0 in
  semifix
    ((\p_2 -> union edge_1
                (forIn edge_1 (\a_3 ->
                 forIn p_2 (\b_4 ->
                 guard (snd a_3 == fst b_4)
                 (set [(fst a_3, snd b_4)])))),
     (\p_2 -> \dp_2 ->
      union dedge_1
        (union
          (forIn dedge_1 (\a_3 ->
           let da_3 = ((), ()) in
           forIn p_2 (\b_4 ->
           guard (snd a_3 == fst b_4)
           (set [(fst a_3, snd b_4)]))))
          (forIn (union edge_1 dedge_1) (\a_3 ->
           let da_3 = ((), ()) in
           union
             (forIn dp_2 (\b_4 ->
              let db_4 = ((), ()) in
              guard (snd a_3 == fst b_4)
              (set [(fst a_3, snd b_4)])))
             (forIn (union p_2 dp_2) (\b_4 ->
              let db_4 = ((), ()) in
              if (snd a_3 == fst b_4) then set [] else
              guard False
              (union (set [(fst a_3, snd b_4)]) (set [])))))))))
```

FIGURE 4.5 *Raw seminaïve translation of transitive closure*

```
\e_0 ->
  let (edge_1, dedge_1) = e_0 in
  semifix
    ((\p_2 -> union edge_1
                (forIn edge_1 (\a_3 ->
                 forIn p_2 (\b_4 ->
                 guard (snd a_3 == fst b_4)
                 (set [(fst a_3, snd b_4)])))),
     (\p_2 -> \dp_2 ->
      union dedge_1
        (union
          (forIn dedge_1 (\a_3 ->
           let da_3 = ((), ()) in
           forIn p_2 (\b_4 ->
           guard (snd a_3 == fst b_4)
           (set [((fst a_3), (snd b_4))])))
          (forIn (union edge_1 dedge_1) (\a_3 ->
           let da_3 = ((), ()) in
           forIn dp_2 (\b_4 ->
           let db_4 = ((), ()) in
           guard (snd a_3 == fst b_4)
           (set [(fst a_3, snd b_4)])))))))
```

FIGURE 4.6 *Seminaïve translation of transitive closure with ⊥ propagation*

```
\e_0 ->
  let (edge_1, dedge_1) = e_0 in
  semifix
    ((\p_2 -> union edge_1
                (forIn edge_1 (\a_3 ->
                 forIn p_2 (\b_4 ->
                 guard (snd a_3 == fst b_4)
                 (set [(fst a_3, snd b_4)])))),
     (\p_2 -> \dp_2 ->
      forIn edge_1 (\a_3 ->
      let da_3 = ((), ()) in
      forIn dp_2 (\b_4 ->
      let db_4 = ((), ()) in
      guard (snd a_3 == fst b_4)
      (set [(fst a_3, snd b_4)])))))
```

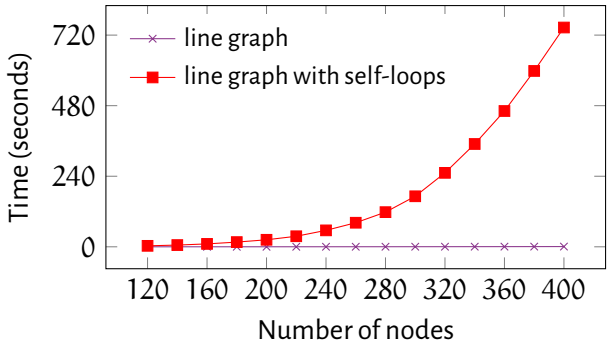
FIGURE 4.7 *Seminaïve translation of transitive closure with ⊥ insertion and propagation*

Each branch of the **`if`**-statement computes an empty set, the first branch explicitly and the else-branch via `(guard False ...)`. The $\bot$ propagation pass recognizes this and rewrites the entire loop to $\bot$, removing it from the *simplified* and *optimized* translations (figures 4.6 and 4.7).

4.3 Change minimization

Before concluding that we have captured the essence of seminaive evaluation, let's try a small twist on our running example: let's add self-loops to every node in our linear graph, producing the graph (V, E) with $V = \{1, ..., n\}$ and $E = \{(i, j) \mid j \in \{i, i+1\}\}$. This makes our reachability relation reflexive, changing the transitive closure from the less-than relation $\{(i, j) \mid 1 \leqslant i < j \leqslant n\}$ to the less-than-or-equal-to relation $\{(i, j) \mid 1 \leqslant i \leqslant j \leqslant n\}$. This produces exactly n new paths, namely $\{(i, i) \mid 1 \leqslant i \leqslant n\}$; since we already had quadratically many paths, ideally this won't affect our performance much.

Unfortunately, adding these self-loops produces an asymptotic slowdown, even with our seminaïve transformation and all optimizations applied (à la *seminaïve optimized*):



What's going on here? Recall from §4.1, page 67 (and confirmed by figure 4.7) that the seminaïve iteration strategy Datafun uses for transitive closure is:

$$x_0 = \emptyset \qquad x_{i+1} = x_i \cup dx_i$$
$$dx_0 = \text{edges} \qquad dx_{i+1} = \text{edges} \bullet dx_i$$

The key computation step here is $dx_{i+1} = \text{edges} \bullet dx_i$. In other words, we prepend edges out of each "frontier" dx_i to get the next frontier dx_{i+1}. Ideally, each frontier consists of pairs (x, y) newly discovered to be reachable; by accumulating them into $x_i = \bigcup_{j<i} dx_j$ we find all such pairs. In a linear graph without self-loops, as we saw in §3.1, this strategy discovers each reachable pair exactly once, because dx_i captures paths of length exactly i, and each reachable pair corresponds to a unique path. But if our edge relation is reflexive, any path from x to y can be adjoined to a self-loop to find a longer path from x to y; thus $dx_i \subseteq dx_{i+1}$. In turn this means that $dx_i = x_{i+1}$; by adding self-loops we've regressed to naïve evaluation!

Taking a logical perspective, at step i, naïve evaluation finds all derivations of depth $d \leqslant i$, while the "seminaïve" strategy we've presented so far finds only derivations of depth $d = i$. This is a clear improvement, but sometimes the same fact may be derived at multiple depths – as in our loopy linear graph, where derivation depth is path length. We care only about *whether* a fact has a derivation, so anything after the first (shallowest) derivation is redundant.

From an incremental computation perspective, this is a problem of unnecessarily large changes. Our seminaïve strategy looks for "new" ways to derive a tuple (x, y), based on whatever was "newly" derived in the previous step. But our notion of "new" is a bit lax, because our derivatives are allowed to be imprecise. Our strategy for finding a fixed point of a function $f : \{\underset{\text{FIN}}{A}\} \to \{\underset{\text{FIN}}{A}\}$ is:

$$\begin{aligned} x_0 &= \emptyset & x_{i+1} &= x_i \cup dx_i \\ dx_0 &= f\ \emptyset & dx_{i+1} &= f'\ x_i\ dx_i \end{aligned}$$

For sets, the derivative property guarantees that $f\ x_i \cup f'\ x_i\ dx_i = f\ x_{i+1}$, but not that $f'\ x_i\ dx_i = f\ x_{i+1} \setminus f\ x_i$. This is exploited in, among others, the derivative rule $\delta(e_1 \cup e_2) = \delta e_1 \cup \delta e_2$. If δe_1 and e_2 intersect (or δe_2 and e_1 intersect), this generates an overly large change.

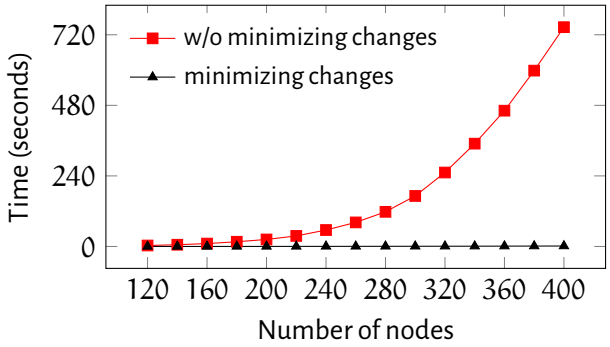
A more precise derivative would be $\delta(e_1 \cup e_2) = (\delta e_1 \setminus e_2) \cup (\delta e_2 \setminus e_1)$. However, this does more work, not less: it does not avoid computing "old" elements $x \in e_1 \cup e_2$, but rather discards them after-the-fact. Indeed, discovering something twice because it has two different derivations seems in general unavoidable; in graph reachability, for instance, how can we know two different paths lead to the same destination except by following them?

So if computing these overly large changes actually takes *less* work, where does the asymptotic slowdown originate? It happens because rediscovering a reachable pair (x, y) at iteration i causes redundant work in all subsequent iterations, because it is included in dx_i (treated as "new") and used to compute $dx_{i+1} = f'\ x_i\ dx_i$. Consequently, dx_{i+1} will include re-derivations of anything the presence of (x, y) makes "newly" deducible; and so on in dx_{i+2}, dx_{i+3}, *etc.*

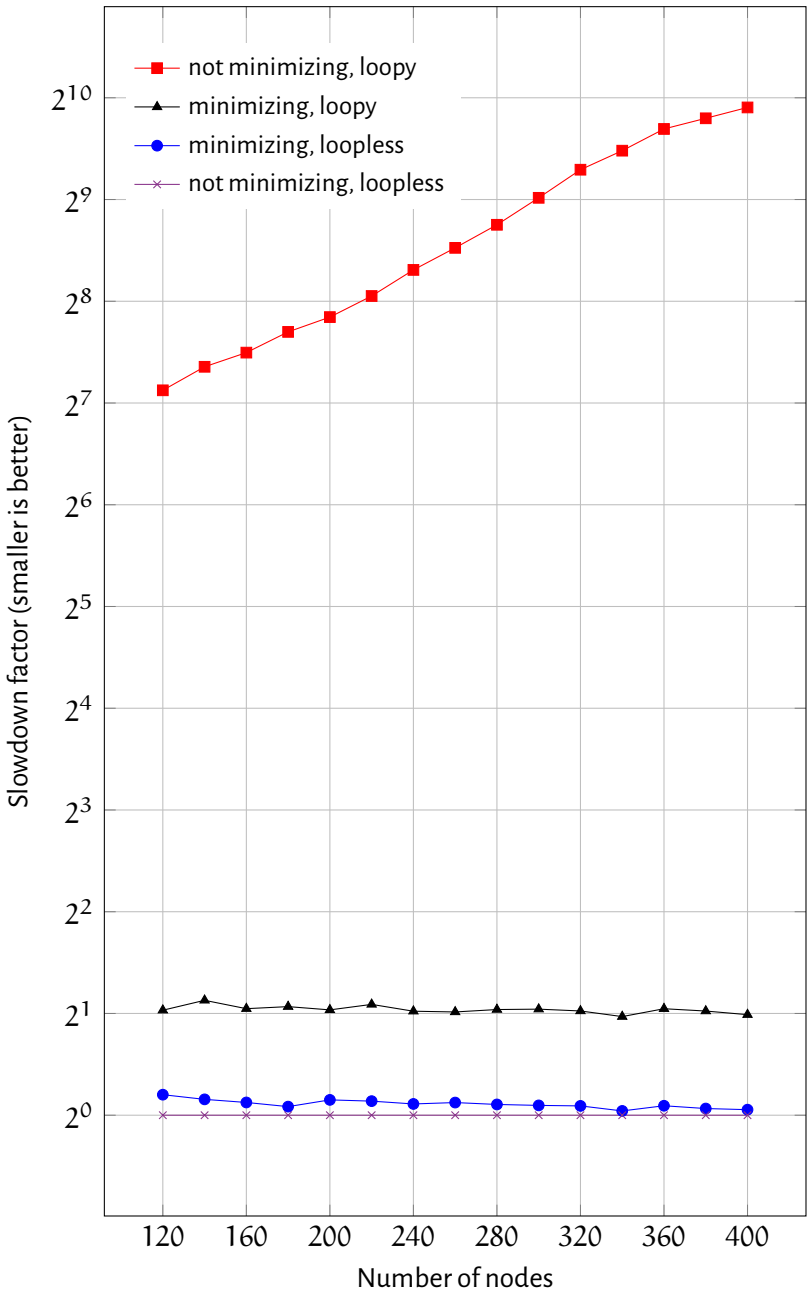
While we may not be able to avoid all rediscovery, we can avoid these unnecessary changes accumulating across iterations – and the resulting asymptotic wastefulness – by *minimizing our changes*. Let's change our strategy for finding the "new" frontier dx_{i+1} to remove anything that's already in x_{i+1}:

$$\begin{aligned} &\textit{for transitive closure} & dx_{i+1} &= (\text{edges} \bullet dx_i) \setminus x_{i+1} \\ &\textit{or more generally} & dx_{i+1} &= (f'\ x_i\ dx_i) \setminus x_{i+1} \end{aligned}$$

This ensures each dx_i is minimal, disjoint from x_i and thus all prior dx_j for $j < i$. (We don't need to do anything to minimize dx_0 since $x_0 = \bot$.) This fixes our asymptotic slowdown:



If we examine all four options – with and without self-loops, with and without minimizing dx_i – and compare their slowdown factors, taking a loopless graph without change minimization



GRAPH SIZE VS SECONDS TO EVALUATE

	120	140	160	180	200	220	240	260	280
NOT MINIMIZING, LOOPY	3.310	5.909	9.881	16.032	23.860	35.866	55.938	81.851	117.876
MINIMIZING, LOOPY	0.048	0.079	0.113	0.162	0.213	0.288	0.359	0.449	0.562
MINIMIZING, LOOPLESS	0.027	0.040	0.060	0.082	0.115	0.149	0.191	0.242	0.294
NOT MINIMIZING, LOOPLESS	0.024	0.036	0.055	0.077	0.104	0.135	0.177	0.222	0.274

	300	320	340	360	380	400
NOT MINIMIZING, LOOPY	171.830	251.552	349.668	461.830	598.083	745.815
MINIMIZING, LOOPY	0.684	0.816	0.959	1.153	1.367	1.545
MINIMIZING, LOOPLESS	0.355	0.428	0.504	0.595	0.704	0.809
NOT MINIMIZING, LOOPLESS	0.332	0.401	0.490	0.558	0.672	0.779

FIGURE 4.8 *The effect of self-loops and change minimization on seminaïve transitive closure on a line graph*

as our baseline (figure 4.8), we find that (a) loopy graphs without change minimization are asymptotically slow; (b) on loopless graphs, minimizing changes has low constant overhead (versus not minimizing); and (c) moving from a loopless to a loopy graph causes a roughly 2x slowdown when minimizing changes, because we must remove rediscovered paths on every iteration.

Two questions remain: (1) why is change minimization correct? and (2) how can we generalize it from finite sets to all semilattice types? As it happens, the answer to the first question also illuminates the second. Change minimization is correct because it preserves validity of changes: if $dx \rhd x \hookrightarrow y : \{\underset{\text{EQ}}{A}\}$, in other words $x \cup dx = y$, then we also have $dx \setminus x \rhd x \hookrightarrow y : \{\underset{\text{EQ}}{A}\}$, because $x \cup (dx \setminus x) = x \cup dx = y$. Thus minimizing changes preserves the inductive invariant that $dx_i \rhd x_i \hookrightarrow f\ x_i$ which guarantees semifix finds a least fixed point.

This condition tells us how to generalize our approach: for each lattice type L, we require a change minimization operator $(\setminus)_L : L \to L \to L$ such that if $dx \rhd x \hookrightarrow y : L$ then $dx \setminus_L x \rhd x \hookrightarrow y : L$.[5] In our runtime library (figure 4.2), we accomplish this by adding a `diff` method to our `Semilat` typeclass:

```
class Semilat a where ...
  diff :: a -> a -> a
  diff dx x = x
```

This method admits a default implementation, $dx \setminus_L x = dx$, which trivially satisfies our correctness condition. Although generic, this is inefficient, for it degenerates to the original non-minimizing implementation of semifix:

$$dx_{i+1} = (f'\ x_i\ dx_i) \setminus_L x_{i+1} = f'\ x_i\ dx_i$$

Instead, we want $dx \setminus_L x$ to be as small as possible (whatever that means for our runtime representation of L) to reduce the work done by any operators applied after change minimization. So we provide more practical implementations for finite sets and product types:

$$dx \setminus_{\{\underset{\text{EQ}}{A}\}} x = dx \setminus x \qquad (dx, dy) \setminus_{L \times M} (x, y) = ((dx \setminus_L x), (dy \setminus_M y))$$

In our Haskell runtime this becomes:

```
instance Ord a => Semilat (Set a) where ...
  diff = Set.difference
instance (Semilat a, Semilat b) => Semilat (a,b) where ...
  diff (da,db) (a,b) = (diff da a, diff db b)
```

Unfortunately, for some semilattices the degenerate default is the *only* valid change minimization operator. This includes any totally-ordered semilattice, such as $\mathbb{N}^{\infty}_{\text{min}}$, the naturals extended with positive infinity under minimum, which is useful for shortest path computations. This inability to meaningfully minimize changes is concerning for the efficiency of computations which use these semilattices; an important direction for future work would be to characterize the worst-case efficiency of seminaïve evaluation over different classes of semilattice.

[5] Given Datafun's monotonicity typing, the reader may wonder whether and with respect to what $(\setminus)_L$ must be monotonic. In practice, on finite sets $dx \setminus x$ is monotone in dx but not x, and this pattern will hold for our other semilattice types as well. However, monotonicity in either argument is not required for correctness.

Chapter 5

Related Work

5.1 Logic, higher-order abstraction, and semilattices

Datafun's value proposition is to extend bottom up logic programming à la Datalog with two additional features: higher-order functional abstraction and support for semilattices other than finite sets. There is a good deal of work on combining logic programming with one or the other of these features, and at least one other system – Flix – which, like Datafun, proposes to combine all three. We'll first briefly consider the systems which feature one or the other and then move on to a more detailed comparison with Flix.

Higher-order extensional logic programming There are many possible approaches to combining the power of logic programming and of higher-order abstractions. The most direct approach would be to directly extend logic programming with support for higher-order relations. Unfortunately, this quickly entangles one with thorny problems of decidability and efficiency. Nonetheless, a line of work starting with Wadge (1991) has explored this approach. Kountouriotis et al. (2005) investigate extending Datalog's bottom-up approach with higher-order relations, an approach very close to Datafun's; however, while they present a prototype implementation, they remark that it can be impractically slow for significantly higher-order programs because it needs to synthesize many large relations. Later approaches move from a bottom-up to a top-down approach, becoming less Datalog and more Prolog (Charalambidis et al., 2013). By contrast, Datafun attempts to reduce implementation difficulties by narrowing the scope of higher-order computation to functions, leaving relations first-order and decidable – higher-order functional programming and decidable first-order relational programming being both well-trodden areas.

Higher-order functional + top-down logic programming Of course, Datafun is not the first language to attempt to add higher-order features to logic programming by combining it with functional programming: Mercury (Somogyi et al., 1994) integrates higher-order functional programming into a logical paradigm, while Curry (Antoy and Hanus, 2010) cleanly combines the lazy functional language Haskell with top-down logic programming. The main difference is that Mercury and Curry's logical features are inspired by Prolog, employing top-down search and unification, while Datafun is inspired by Datalog, uses bottom-up enumeration, and imposes deliberate restrictions to ensure termination and avoid Turing-completeness.

Embedding database queries in functional languages Datalog has sometimes been described as "relational algebra plus fixed points", and there is a long line of work on embedding database query languages into general-purpose languages, including pioneering efforts such as Machiavelli (Ohori et al., 1989) and Kleisli (Wong, 2000), as well as more recent systems such as Ferry (Grust et al., 2009) and LINQ in C# (Cheney et al., 2013). The focus of this work has been on embedding query languages based on relational algebra into general purpose languages, with an emphasis on statically compiling higher-order queries into the first-order queries supported by existing database systems (Cheney et al. (2014) is a representative example). Datafun's approach is different: rather than embed Datalog into a general purpose language, Datafun is *also* a "little language", albeit one that happens to be a higher-order functional language. We have not attempted to embed Datafun into an existing language, as this would greatly complicate the context-management operations needed to ensure monotonicity.

Logic with semilattices We know of two systems which extend Datalog with support for semilattices without incorporating higher-order abstraction. IncA_L (Szabó et al., 2018) is an incremental Datalog engine with support for custom semilattices, aimed at static analysis. Bloom^L (Conway et al., 2012) is a Datalog-inspired language aimed at distributed computing, where monotonicity is used to ensure eventual consistency (Alvaro et al., 2011) and custom semilattices are used to extend the range of types to which monotonicity analysis can be fruitfully applied. Although both of these are clearly related to Datafun's goals, neither includes the crucial ingredient of higher-order abstraction that leads to most of Datafun's unique capacities, as well as its unique design and implementation challenges.

5.1.1 Flix

Flix (Madsen et al., 2016) and Datafun both augment Datalog with higher-order functional programming and semilattice types, but they go about this merger of logic and functional programming in different ways. Datafun embeds Datalog's semantics into a functional language by adding first-class support for finite sets and monotone fixed points, ensuring monotonicity via a custom type system. Flix, however, is really two languages in a trenchcoat – one logical and Datalog-inspired, the other functional. These language halves interoperate closely: the logic fragment can use semilattices whose merge functions are defined in the functional fragment, and with a recent extension (Madsen and Lhoták, 2020) the functional fragment can manipulate first-class values representing groups of logic-fragment rules ("Datalog constraints") which can be composed and evaluated at run-time.

Our main contribution compared to Flix is to demonstrate that this separation between functional and logic layers is unnecessary; they can be smoothly integrated by paying close attention to semantics, and standard Datalog optimizations such as seminaïve evaluation can be generalized to operate on functional languages, with the unexpected side-benefit of revealing a theory of higher-order incremental monotone computation. The practical flip side of this theoretical contribution is that Flix, because it separates functional from logic, can reuse existing techniques for implementing and optimizing Datalog *without* needing to reinvent them in a higher-order setting.

Flix and Datafun also have different approaches to monotonicity: Flix does not check monotonicity via types, but monotonicity is nonetheless important for the interoperation of the

logical and functional fragments; functions invoked in certain positions in the logic fragment must be monotone to ensure that a fixed point exists. To this end, a verification toolchain has been developed for Flix which employs SMT solving and symbolic execution to check properties, including monotonicity but also safety and soundness of static analyses (Madsen and Lhoták, 2018).

The advantage of checking these properties with such high-powered machinery is that it better supports user-defined posets and semilattices: in Datafun, adding a new datatype with a custom ordering is a matter for the language designer, but a Flix programmer may do it for themself, if they can convince the verification machinery. The disadvantage is that these verification techniques have more "black-box" behavior; they are less predictable and reliable (in terms of resource usage, error message quality, and code accepted) than a compositional type system. Nonetheless, for certain applications such as static analysis, the use of user-defined semilattice types is highly desirable. We believe a fruitful direction for future research would be to hybridize Datafun's monotonicity type system with lightweight verification techniques. In our ideal language, types with custom orderings would be defined in a modular, encapsulated fashion. Verification would take place inside the module, to ensure the interface it exposes has the properties (such as monotonicity) it claims, but a compositional type-system would handle code external to the module, which acts as a client of this verified interface.

5.2 Incremental computation

In chapter 3 we presented seminaive evaluation in Datalog and Datafun as a matter of incrementalizing the inner loop of our fixed points to avoid recomputation. Our approach was to only compute the changes between iterations, to which end we used a static transformation to push changes through expressions in our language. However, this is only one of many approaches to incremental computation. In this section we'll examine other approaches and how they relate to ours.

The core problem of incremental computation is to handle change while minimising work. Most approaches to incremental computation are built around one of two insights into how to do this: *dependency tracking* or *finite differencing*.

Dependency tracking The simplest way to avoid doing unnecessary work is to avoid re-executing code if the data it depends on hasn't changed, and re-use its previous result instead. Generally these dependencies are represented by some sort of *dependency graph*. One of the oldest and most familiar applications of this idea is the build system Make and its many relatives. Perhaps even more ubiquitously, this strategy is used by spreadsheet software such as Microsoft Excel, where cells' contents may be (re)computed using the content of other cells, and so on recursively (Mokhov et al., 2020). A long line of work starting with self-adjusting computation (Acar, 2005; Acar et al., 2002) and descendant systems such as Adapton (Hammer et al., 2014) apply dependency tracking to general-purpose programming languages by tracing execution to construct a dynamic dependency graph.

It would be an intriguing line of future work to examine whether a dependency tracking approach, probably building on prior work such as SAC or Adapton, would be useful for computing Datafun's monotone fixed points. However, Datafun was inspired by Datalog, and

dependency tracking is not the approach taken by Datalog's seminaïve evaluation. Instead, it uses *finite differencing*.

Finite differencing Dependency tracking approaches incrementalization with a yes-or-no mindset: did our dependencies change? Suppose instead we ask the question: *how* did our dependencies change? By analyzing this difference we may be able to compute the resulting difference to our output. For constant-time operations over atomic data, like adding two 64-bit numbers, this may not be any faster than simple re-execution. But for bulk operations over collection types, where changes can be much smaller than the original data, differencing is a more natural approach.

For example, you *could* incrementally sum a list of changing numbers using a dependency graph – ideally a balanced tree, so updates take $O(\log n)$ steps. But with direct access to the difference, when an element changes you can simply add the difference to the previous sum. This is $O(1)$, easily extended to handle new or removed elements, and doesn't require balancing or rebalancing a tree.

However, this example works only because addition on the integers commutes and associates.[1] This is what allows us to combine a sum $\sum_i x_i$ with a difference dx to the j^{th} element and find the updated sum:

$$dx + \sum_i x_i = \sum_i \begin{cases} x_i + dx & \text{if } i = j \\ x_i & \text{otherwise} \end{cases}$$

Unlike the fairly generic dependency-graph strategy, efficient derivatives depend crucially on the structure of the modification to the input. Foundational work by Paige and Koenig (1982) takes this literally, differentiating set-valued expressions with respect to lines of program code modifying a depended-on variable. More recent work represents these modifications (the "differences" of "finite differencing") as values of a datatype equipped with some kind of algebraic structure. For instance, DBToaster incrementally maintains SQL queries using rings (Koch, 2010, 2013); Differential Dataflow uses groups (McSherry et al., 2013); recent work on incremental Datalog uses monoid actions (Alvarez-Picallo et al., 2019); and Datafun uses semilattices.

Since finite differencing is advantageous for operations over large collections, but requires algebraic insight that makes it non-obvious how to apply it to arbitrary programs, it should be no surprise that many of the examples just cited come from the database research community, which often deals with structured operations on bulk data. However, this approach has recently crossed over into the PL research community by way of the *incremental λ-calculus*, and it is from this line of work that Datafun takes inspiration.

5.2.1 The incremental λ-calculus

The incremental λ-calculus was introduced by Cai, Giarrusso, Rendel, and Ostermann (2014) and further developed by Giarrusso, Régis-Gianas, and Schuster (2019) and in Giarrusso's PhD thesis (2020). We briefly discussed our adaptation of it in §3.2, but here we give a slightly

[1] More precisely, associativity and commutativity suffice because our aggregation (summation) is the same as the operator that applies our differences (addition). If we were taking the maximum of the list instead of its sum, this strategy would not work, even though maximum itself commutes and associates.

fuller comparison (though still only a summary) between our formulation, the original, and its further developments.

Change structures

The insight of the incremental λ-calculus that is it is possible to extend the differencing approach to higher-order computation by finding an appropriate notion of change for functions. To this end, the incremental λ-calculus associates each type with a *change structure* capturing how values of that type may change, and how to represent these changes. The precise formal definition of a change structure differs between the various versions of the incremental λ-calculus. A good starting point is the notion of a *basic change structure* introduced in Giarrusso's PhD thesis (2020, chapter 12, definition 12.1.1): a basic change structure on a set S consists of a change set ΔS and a validity relation $dx \triangleright x_1 \hookrightarrow x_2 : S$ indicating that $dx : \Delta S$ is a valid change from the base points $x_1 : S$ to $x_2 : S$.

For example, we can endow the naturals $\mathbb{N}$ with a basic change structure by letting changes be integer differences $\Delta\mathbb{N} = \mathbb{Z}$ and letting $dx \triangleright x_1 \hookrightarrow x_2 : \mathbb{N} \iff x_1 + dx = x_2$. To handle higher-order computation, Giarrusso endows function types $A \to B$ with a basic change structure as follows (definition 12.1.8): let $\Delta(A \to B) = A \to \Delta A \to \Delta B$ and define the validity relation by saying that a valid function change $df \triangleright f_1 \hookrightarrow f_2 : A \to B$ is one which takes an argument $x : A$ and valid change to it $dx \triangleright x_1 \hookrightarrow x_2 : A$ to a valid output change $df\ x_1\ dx \triangleright f_1\ x_1 \hookrightarrow f_2\ x_2 : B$, that is, a change between the original function applied to its original argument $f_1\ x_1$ and the updated function applied to an updated argument $f_2\ x_2$.

Giarrusso goes on to define more elaborate "full" change structures (definition 13.1.1) which additionally possesses operators $x \oplus dx$ for updating a base point by a change, $y \ominus x$ for finding a change between two points, $\mathbb{0}$ for finding a zero change from a point to itself, and $dx \circledcirc dy$ for composing two changes.[2]

Datafun's change structures (definition 14) lie somewhere between Giarrusso's basic and "full" change structures, modified to handle monotonicity: in Datafun S (or VS in our notation) and ΔS are not sets but posets. For example, our change structures for functions almost coincide, except that because of Datafun's monotonicity typing we must let $\Delta(A \to B) = \Box A \to \Delta A \to \Delta B$, using $\Box$ to allow function changes to be non-monotone with respect to the base point. The root divergence here is one of goals: in Datafun we are not trying to respond to arbitrary changes to our whole program's input, but only to incrementalize the inner loops of fixed points to calculate them more efficiently. Because these fixed points are monotone, in Datafun we need only handle increasing change (enforced by our *soundness* condition). The price of this simplification is that we must pay careful attention to the interaction of incrementalization with Datafun's monotonicity-checking modal type system in our program transformation and its proof of correctness.

Because of the limited way Datafun uses incremental computation, we only need some of the operators from Giarrusso's full change structures, and only at certain types. We use $\mathbb{0}$

[2] In the original presentation by Cai et al. (2014) these operators (except $\circledcirc$, which is not present) are taken as primary rather than as additions to a basic change structure, and the validity relation $dx \triangleright x_1 \hookrightarrow x_2 : S$ is reduced to a set of valid changes $\Delta_S\ x_1 \subseteq \Delta S$. Datafun's approach to seminaïve computation was originally inspired by this paper, but we moved to a validity-relation approach because of the difficulties of defining these operators in a monotonicity-aware setting; thus the validity relation approach to change structures appears to have been indendently invented in both Datafun and in Giarrusso's work.

explicitly in the ϕ and δ translations of loops, **for** $(x \in e)\ f$, to supply the zero change for the elements x; but since these elements are always of first-order type $\underset{\text{EQ}}{A}$, we only need to compute zero changes for first order values. We implicitly use $\oplus$ in restricted form in the implementation of seminaïve fixed points, to combine the value x_i of the i^{th} iteration with its corresponding change dx_i. However, fixed points are always at first-order lattice types $\underset{\text{FIX}}{L}$, and because of the way our change types are constructed $\oplus$ at these types is simply $\vee$. Even more subtly, we use $\ominus$ in the same place, to find the kick-off change between the first iteration $x_1 = f\ \bot$ and the zeroth $x_0 = \bot$; but, again because of the way these change types are constructed, $f\ \bot \ominus \bot$ is simply $f\ \bot$. These simplifications are fortuitous, because the interaction of the fully general versions of these operators with monotonicity typing presents several difficulties.[3]

The derivative translation

Although change structures allow us to specify the notion of a derivative that takes input changes to output changes, they do not by themselves tell us how to find such a derivative. The incremental λ-calculus and Datafun both accomplish this by static differentiation: we give source-to-source translations from a program to its derivative, essentially by decomposing a program into primitive operations which we know how to differentiate and recombining these using an analogue of the chain rule.

Cai et al. (2014) call this translation *Derive*, while Datafun calls it δ. Datafun's approach was directly inspired by Cai et al., and where their features overlap the two translations nearly coincide. For example, the derivative of function application is:

$$\begin{aligned} \mathit{Derive}(e_1\ e_2) &= \mathit{Derive}(e_1)\ e_2\ \ \mathit{Derive}(e_2) \\ \delta(e_1\ e_2) &= \delta e_1\ [\phi e_2]\ \delta e_2 \end{aligned}$$

Besides notation there are two differences here, which are indicative of the differences from the incremental λ-calculus more generally: (1) Datafun is concerned with monotonicity, and so since the function changes may be non-monotone in their first argument we need to box it; and (2) besides δ, Datafun has another term translation ϕ which speeds up execution by executing fixed points seminaïvely, and for technical reasons these translations must be mutually-recursive; wherever *Derive* uses a part of the original term it is given, δ instead uses its ϕ-translation.

---

[3] For starters, in general $x \ominus y$ may not exist unless $x \geqslant y$ since in Datafun we only support increasing changes. For another example, it is difficult to define the $\oplus$ operator internally in a way that respects monotonicity typing at higher type. Recall that in Datafun all functions are monotone and $\Delta(A \to B) = \Box A \to \Delta A \to \Delta B$. The natural definition of $\oplus_{A \to B}$ would seem to be:

$$\begin{aligned} \oplus_{A \to B} &: (A \to B) \times (\Box A \to \Delta A \to \Delta B) \to (A \to B) \\ (f \oplus df) &= \lambda X.\ f\ X \oplus df\ [X]\ (0\ X) \end{aligned}$$

However, this passes the *monotone* variable X to the function change df, which takes it as a *discrete* argument – it does not type-check! Indeed, it is not hard to come up with functions f, df such that the result of $f \oplus df$ as defined above is not a monotone function.

These difficulties might be overcome with further careful work; for example, it should be possible to prove that $f \oplus df$ is monotone so long as df is a valid change to f. This would make $\oplus$, like semifix, a "trusted primitive" whose implementation cannot be expressed in Datafun itself.

While Datafun adds complications in the form of monotonicity typing and the seminaïve transformation, later work on the incremental λ-calculus adds complications of its own, extending it to handle the untyped λ-calculus and therefore nontermination; to prove correctness, they use a *step-indexed* logical relation (Giarrusso et al., 2019). They also address the problem of caching intermediate results, but in order to explain this problem and its relevance to Datafun, it will help to briefly revisit the idea of dependency tracking.

Dependency tracking as a change structure

We started this section by comparing dependency tracking to finite differencing, observing informally that finite differencing generalizes dependency tracking by asking not merely "did our dependencies change?" but "*how* did our dependencies change?" This insight can be formalized using the incremental λ-calculus's change structures, as there is a simple generic change structure which captures the question "did it change?". Allowing ourselves to dip into pseudo-Haskell for a moment, consider the parameterized type Update A defined by:

$$\textbf{data}\ \mathsf{Update}\ A = \textsc{old} \mid \textsc{new}\ A$$

Any type A may be endowed with a basic change structure by letting $\Delta A = \mathsf{Update}\ A$ and letting $dx \triangleright x \hookrightarrow y : A$ be defined by:

$$\textsc{old} \triangleright x \hookrightarrow x : A \qquad\qquad \textsc{new}\ y \triangleright x \hookrightarrow y : A$$

This change structure has the wonderful property of trivializing differentiation; one valid derivative of $f : A \to B$ is simply:

$$\begin{aligned} f'\ x\ \textsc{old} &= \textsc{old} \\ f'\ x\ (\textsc{new}\ y) &= \textsc{new}\ (f\ y) \end{aligned}$$

The only complication is handling multi-argument functions: since $\mathsf{Update}\ (A \times B) \ncong \mathsf{Update}\ A \times \mathsf{Update}\ B$, a function taking multiple arguments $g : A \to B \to C$ needs a slightly more interesting derivative:

$$\begin{array}{llllll} g' : A \to \mathsf{Update}\ A \to B \to \mathsf{Update}\ B \to \mathsf{Update}\ C & & & & & \\ g' & a & \textsc{old} & b & \textsc{old} & = & \textsc{old} \\ g' & \_ & (\textsc{new}\ a) & b & \textsc{old} & = & \textsc{new}\ (g\ a\ b) \\ g' & a & \textsc{old} & \_ & (\textsc{new}\ b) & = & \textsc{new}\ (g\ a\ b) \\ g' & \_ & (\textsc{new}\ a) & \_ & (\textsc{new}\ b) & = & \textsc{new}\ (g\ a\ b) \end{array}$$

The general strategy is to rerun the original function if any of its arguments change, reusing the previous value for arguments that did not change. This is precisely the strategy behind dependency tracking.

Caching intermediate results

We have observed that dependency tracking's re-execution strategy can be seen as a special case of finite differencing. However, one thing all dependency tracking systems do is store intermediate results between runs so they can reuse them if they don't need to be recomputed

because their dependencies haven't changed. Thus far our presentation of the incremental λ-calculus (or indeed of Datafun) has not mentioned caching intermediate results. This presents an issue; although our goal is to translate input changes into output changes, in general computing the output difference may require both the input difference *and the original input*. For example, in the incremental λ-calculus the derivative of a function $f : A \to B$ has type $A \to \Delta A \to \Delta B$, taking the original argument A as well as its change ΔA. Where does this original argument come from?

By default, both Datafun and the incremental λ-calculus as originally introduced in Cai et al. (2014) recompute these arguments. For example, recall the derivatives of function application in each system:

$$\mathcal{D}erive(e_1\ e_2) = \mathcal{D}erive(e_1)\ e_2\ \ \mathcal{D}erive(e_2)$$
$$\delta(e_1\ e_2) = \delta e_1\ [\phi e_2]\ \delta e_2$$

These expressions recalculate the original argument e_2 (or in the case of Datafun, its sped-up version ϕe_2). On its own, recomputation is a losing strategy: the whole point of finite differencing is to compute the change *more efficiently* than by recomputing the output! Luckily, sometimes we do not need this original input. A simple example is summing a list of changing numbers: the change to the sum is simply the sum of the changes to each element; the original list is not necessary to compute the change. Or, for a natural example in the context of Datafun, fix a binary relation edge and consider two different functions, consing and appending, defined as follows (recall that $R \bullet S$ stands for relation composition):

$$\text{consing path} = \text{edge} \cup (\text{edge} \bullet \text{path})$$
$$\text{appending path} = \text{edge} \cup (\text{path} \bullet \text{path})$$

The fixed point of either function computes the transitive closure of edge: consing by extending paths one edge at a time, and appending by appending paths together. Now let's take a look at these functions' derivatives:[4]

$$\text{consing}'\ \text{path}\ \text{dpath} = \text{edge} \bullet \text{dpath}$$
$$\text{appending}'\ \text{path}\ \text{dpath} = (\text{path} \bullet \text{dpath}) \cup (\text{dpath} \bullet \text{path}) \cup (\text{dpath} \bullet \text{dpath})$$

Observe that consing′, unlike appending′, does not need its first argument path, representing the original argument value.

Following the incremental λ-calculus we call functions whose derivatives do not depend on their original input, like consing or the sum of a list, *self-maintainable*. Because the transformation in Cai et al. (2014) does not cache intermediate results, it is really only suitable for programs composed primarily of self-maintainable functions, where the recomputation of these unused original arguments can be optimized out. To handle non-self-maintainable

[4] To obtain these derivatives, let $\delta\text{edge} = \varnothing$ since we assume the edge-set is fixed and apply the rules:

$$\delta(R \cup S) = \delta R \cup \delta S$$
$$\delta(R \bullet S) = (R \bullet \delta S) \cup (\delta R \bullet S) \cup (\delta R \bullet \delta S)$$

behavior, follow-up work by Giarrusso et al. (2019) extends the derivative translation to cache intermediate results.

Datafun takes a simpler approach. We cache intermediate results in exactly one place: the implementation of semifix. Recall that, to compute the fixed point of a function f, semifix computes the sequences x_i, dx_i defined by:

$$\begin{aligned} x_0 &= \bot & x_{i+1} &= x_i \vee dx_i \\ dx_0 &= f\ \bot & dx_{i+1} &= f'\ x_i\ dx_i \end{aligned}$$

Since x_{i+1} and dx_{i+1} depend only on their immediate predecessors, we need exactly two pieces of state to produce this sequence: the previous iteration x_i and its change dx_i. This is our only cache; unless f is self-maintainable, any intermediate values it requires will be recomputed by $dx_{i+1} = f'\ x_i\ dx_i$. Serendipitously, in practice most step functions are either self-maintainable or do not compute expensive intermediate results.

For instance, consider implementing transitive closure as the fixed point of either consing or appending. We have already observed that consing is self-maintainable.[5] Even though appending is not, however, it does not require extensive recomputation:

$$dx_{i+1} = \mathsf{appending}'\ x_i\ dx_i = (x_i \bullet dx_i) \cup (dx_i \bullet x_i) \cup (dx_i \bullet dx_i)$$

All intermediate results in this expression (for example, $x_i \bullet dx_i$) depend upon dx_i; there is no work that could be saved by caching them, because the cache would be invalidated immediately. We conjecture that this holds so often for the programs we have examined because these fixed point step functions are essentially unions of (possibly many-way) relational joins. (Indeed, in Datalog this is baked into the language syntax!) It's not too hard to see that the derivative of a union of joins is itself a union of joins, and each component join will depend on at least one changing relation. So the question reduces to whether relational joins can be incrementalized efficiently without caching intermediate results. In the case of binary joins, at least, the answer is yes. Pursuing this conjecture further we leave to future work.

5.2.2 The monoidal approach to change

The incremental λ-calculus has its notion of a change structure; Datafun has another; but these do not exhaust the space of possibilities. A line of work starting with Alvarez-Picallo et al. (2019), and most thoroughly expounded in Mario Alvarez-Picallo's thesis (2020) has explored representing change structures using monoid actions, which they call *change actions*. A change action on a set A consists of a monoid $(\Delta A, +_A, 0_A)$ and a monoid action $\oplus_A : A \times \Delta A \to A$. As before we interpret $\oplus$ as applying a change to a base value. The monoid action law $x \oplus (dx + dy) = (x \oplus dx) \oplus dy$ says that $+$ composes changes (corresponding to $\circledcirc$ in the incremental λ-calculus); and the other law $x \oplus 0 = x$ makes 0 a zero change to any value. The primary differences from the incremental λ-calculus are:

1. The lack of a $\ominus$ operator, thus allowing for incomplete change structures, where it may not be possible to find a change from one value to any other.

[5] In fact, it is an exemplar of a large class of self-maintainable functions: join-distributive maps.

2. The requirement that there exists a single value $0 : \Delta A$ which acts as a *universal* zero change, rather than an operator $0 : A \to \Delta A$ that finds a zero change to a particular value.

3. Change actions lack a validity relation: every change must be valid for every base point.

Tantalizingly, Alvarez-Picallo et al. (2019) explicitly use this notion of change action to give a differentiation/incrementalization transformation for Datalog programs. Might this monoidal approach work in Datafun as well? At first it seems as though it might: many of Datafun's change structures, in particular those on semilattice types, fit into this structure: for instance, on finite set types $\{\underset{\text{EQ}}{A}\}$ the change action is simply sets $\Delta\{\underset{\text{EQ}}{A}\} = \{\underset{\text{EQ}}{A}\}$ with $+ = \oplus = \cup$ and $0 = \varnothing$.

However, points (2) and (3) above produce problems when considering the change structure for functions. The incremental λ-calculus and Datafun both take function changes to be generalizations of function derivatives, such that the zero-change to a function is its derivative. But this is incompatible with requiring a universal zero-change $0 : \Delta(A \to B)$; there is no "universal derivative". For this reason both Alvarez-Picallo et al. (2019) and Alvarez-Picallo (2020) use pointwise function changes $\Delta(A \to B) = A \to \Delta B$, letting $(f \oplus df)\ x = f\ x \oplus df\ x$. This fundamental divergence from the incremental λ-calculus would require redesigning the ϕ/δ transformations – neither the 2019 paper nor Alvarez-Picallo's thesis give an explicit derivative transformation for a higher-order language.

However, there is a deeper issue: in a monotonicity-aware context like Datafun, we must choose whether the pointwise change functions are required to be monotone, $\Delta(A \to B) = A \to \Delta B$, or allowed to be non-monotone, $\Delta(A \to B) = \Box A \to \Delta B$. Both choices have serious problems:

Non-monotone changes If we allow function changes to be non-monotone we have an immediate problem: the updated function $f \oplus df = x \mapsto f\ x \oplus df\ x$ is not guaranteed to be monotone. The only way to repair this without requiring that the change itself be monotone would seem to be to re-introduce the incremental λ-calculus's notion of validity, and say that df is only a valid change to f if $f \oplus df$ remains monotone. This would require a considerable elaboration of the theory of change actions.

Monotone changes Unfortunately, insisting that pointwise function changes be monotone makes it impossible to differentiate some perfectly reasonable functions. For example, take the integers $\mathbb{Z}$ equipped with the natural change action $\Delta\mathbb{Z} = \mathbb{Z}$, $\oplus_{\mathbb{Z}} = +_{\mathbb{Z}} = +, 0_{\mathbb{Z}} = 0$, and consider the function expression $\lambda y.\ \max(x, y) : \mathbb{Z} \to \mathbb{Z}$. Now suppose x increases from 0 to 1; how does this function change in response? In other words, what is the change between $\max(0, \_)$ and $\max(1, \_)$? Tabulating its values for $y = 0, 1, 2, \ldots$ we can clearly see it is not monotone:

y	0	1	2	$\cdots$
$\max(0, y)$	0	1	2	$\cdots$
$\max(1, y)$	1	1	2	$\cdots$
$\max(1, y) - \max(0, y)$	1	0	0	$\cdots$

Thus, forcing function changes to be monotone with respect to the base point is a very limiting approach.

It is possible that by combining change actions with a validity relation, thus allowing the monoid action to be partial, one could achieve a synthesis of the change action approach and Datafun's higher-order monotonicity. We leave this to future work.

Chapter 6

Looking Back and Forward

To the extent that we have had in this dissertation a thesis, a singular statement we have aimed to demonstrate, it is that we can seamlessly integrate Datalog's features into a typed higher-order functional language by deconstructing them semantically. Or, as we said in the introduction (page 13):

> The goal of this thesis is to design a language which improves on Datalog's ability to express monotone fixed point computation over semilattices by finding ways to lift Datalog's restrictions without sacrificing either its simple semantics or its practical implementation strategies.

Unfortunately, we cannot claim to have definitively proven our thesis: we have made a start towards this goal, but much remains to be done.

The examples in chapter 2 show Datafun can at least express Datalog-style queries. Moreover in §1.4 we listed four things Datalog's restrictions do not permit: (1) functional abstraction, (2) semilattices other than set union, (3) arithmetic, user-defined functions, and aggregation, and (4) compound data. Of these, Datafun makes *functional abstraction*, *arithmetic*, *user-defined functions*, and *compound data* straightforward. It does not include *semilattices other than sets* (and products of sets), nor *aggregations* other than semilattice aggregation (**for**-loops). However, its design lays a clear foundation for such extensions: new semilattices can be added as semilattice types, and aggregations can be added as primitive higher-order functions.

In addition to lifting Datalog's restrictions, we also wished to preserve two desirable qualities: *simple semantics* and *practical implementation strategies*. On each count, we have achieved only qualified success.

Datafun possesses a simple denotational semantics that captures Datalog's ability to manipulate finite relations: Datalog's recursively-defined relations become Datafun's bottom-up fixed points, and Datalog's stratification is enforced by Datafun's monotonicity types. However, least fixed points also require an ascending chain condition; here Datafun and Datalog diverge. Datalog makes a clear distinction between relations and the terms they range over, and enforces *constructor-freedom:* programs may not construct new terms not present in the source program. This ensures a finite universe of terms, and thus an ascending chain condition for relations over this universe.

By contrast, in Datafun relations and terms are simply types of values; this adds flexibility but requires a different way to guarantee the ascending chain condition. In theory we require

the *type* at which we take a fixed point to satisfy the ACC; for set types, this requires the element type be finite. In practice we have hand-waved this condition away – for instance, our regular expression combinators from §2.2.3 use **fix** at type {int}, representing sets of indices into a string. Although valid indices into a particular string form a finite set, the integer type int is infinite. Semantically, this is a serious flaw in the foundations of our approach. Practically, it is on par with existing approaches: real Datalog engines routinely permit constructors or arithmetic, putting the onus on the programmer to avoid infinite relations and non-termination.

As for *practical implementation strategies,* we have constructed a Datafun implementation supporting seminaïve evaluation, a central Datalog technique without which recursive relations are impractical to compute; but this required a significant novel development of theory, and represents only one of many techniques necessary for an efficient implementation. It is plausible, but hardly certain, that other standard techniques – in particular query planning and optimization (necessary to replace nested **for**-loops with efficient relational joins) and demand transforms such as magic sets (which make queries over large recursively defined relations practical by computing only a smaller relevant subset of the relation) – could be extended to Datafun; this remains future work.

6.1 Directions forward

In this section we sketch some directions for future work on Datafun to address the shortcomings identified in the previous section.

The ascending chain condition As we have discussed, Datafun's semantics for **fix** require ACC; our motivating examples satisfy this in principle, but Datafun's simple type system cannot capture this. For instance, we might represent the nodes of a graph as integers; but while the graph may be finite, the integers are infinite. To remedy this, we must either (a) accept nontermination and adjust our semantics or (b) reject nontermination and adjust our language.

Accepting nontermination is simpler, but it removes potential optimizations by invalidating many program equalities, for instance *loop interchange:*

$$\textbf{for}\ (x \in e_1,\ y \in e_2)\ e_3 = \textbf{for}\ (y \in e_2,\ x \in e_1)\ e_3 \quad (\text{when } x, y \text{ not used in } e_1, e_2)$$

This equation fails if e_1 is nonterminating but $e_2 = \emptyset$ or vice-versa. Perhaps some adjustment to how Datafun expresses queries like this, such as ditching monadic set-comprehensions in favor of applicative ones (see *query planning and optimization* below), might resolve this.

Rejecting nontermination requires creatively re-thinking how we guarantee ACC. We might, for instance, capture reasoning about the finiteness of sets at the type level, mechanizing our hand-waving about taking a fixed point over finite sets of *nodes* rather than of *integers*. Or, we might try to capture the spirit of Datalog's no-constructor restriction by singling out a class of "uncreative" functions that do not create new data; perhaps by treating "creating data" as an effectful operation,[1] or perhaps by exploiting parametricity to guarantee all data in our output came from our input.

[1] Here is a simple system along these lines: introduce a monad Bless and type constructor Holy with the methods:

Query planning and optimization To implement Datafun efficiently, we need to be able to identify relational joins. Identifying joins is also a necessary first step if we wish to apply standard query planning and optimization techniques. Datafun expresses joins as nested loops; for instance, relational composition (a simple equijoin):

$$\_ \bullet \_ : \{\underset{EQ}{A} \times \underset{EQ}{B}\} \to \{\underset{EQ}{B} \times \underset{EQ}{C}\} \to \{\underset{EQ}{A} \times \underset{EQ}{C}\}$$
$$S \bullet T = \textbf{for}\ ((a, b) \in S)\ \textbf{for}\ ((!b, c) \in T)\ \{(a, c)\}$$

Implementing this as it is written, using a nested loop, has time-complexity $O(|S| \cdot |T|)$. An on-the-fly hash-join, building an index on the $\underset{EQ}{B}$ column of either S or T, takes $O(|S|+|T|+|S \bullet T|)$. Unfortunately, nested **for**-loop expressions are in general not reducible to joins, because the inner loop can loop over a function of the outer loop variable: $(\textbf{for}\ (x \in S)\ \textbf{for}\ (y \in f(x))\ \ldots)$. This does not happen in Datalog and makes query planning significantly harder: in database parlance, we've expressed joins (easy) as subqueries (hard). We could address this in the compiler by heuristically identifying nested loops which can be implemented as joins; or, we could force our comprehensions to be applicative rather than monadic, banning the problematic nesting entirely.

Moreover, there remains the question of what to do once we have identified joins. We could again attempt to lift standard database techniques to apply to a higher-order setting; or, we could attempt to sidestep this work by compiling Datafun into an existing Datalog dialect by using some variety of defunctionalisation.

Aggregations and the Boom hierarchy Aggregations can be added to Datafun as primitive higher-order functions and pose no semantic issues so long as their types properly capture their (non-)monotonicity. For instance, consider counting and summation:

$$\text{size} : \{\underset{EQ}{A}\} \to \text{int} \qquad \text{sum} : (\square\underset{EQ}{A} \to \text{int}) \to \square\{\underset{EQ}{A}\} \to \text{int}$$
$$\text{size}\ s = |s| \qquad \text{sum}\ f\ s = \sum_{x \in s} f(x)$$

Throwing in each aggregation we need as a primitive function is, however, somewhat ad-hoc; we might unlock a richer approach if we pay attention to the *semantics* of aggregations. Many aggregations arise from free functors into categories of algebraic structures. For instance, bags (finite multisets) form the free commutative monoid; and given a map from a bag's elements into a commutative monoid $(M, +, 0)$, we can aggregate the bag's contents into M:

$$\text{bagsum} : (A \to M) \to \text{Bag}\ A \to M$$
$$\text{bagsum}\ f\ b = \sum_{x \in b} f(x)$$

Modulo monotonicity, sum arises from bagsum by letting $M = (\mathbb{Z}, +, 0)$. Many common aggregations and some of the most common data structures used in programming arise in a

---

$$\text{anoint} : A \to \text{Bless}\ (\text{Holy}\ A) \qquad \text{disregard} : \text{Holy}\ A \to A$$

such that map disregard (anoint x) = pure x. The idea is that Holy A represents the *finite* subset of values of A which have been anointed. Since functions passed to **fix** are pure, they cannot anoint new values; so we may treat Holy A as a finite type, and thus (for instance) use **fix** at the type {Holy int} even though we may not at {int}. Alas, actually programming with anoint/disregard is an exercise in boilerplate.

similar way from three free functors in particular: lists as the free monoid, bags as the free commutative monoid, and finite sets as the free idempotent commutative monoid (equivalent to a semilattice). These are three layers of the "Boom hierarchy" (Backhouse and Hoogendijk, 1993).[2] Each free functor forms half of an adjunction between **Set** and the respective category of algebraic structures (**Mon**, **CMon**, and **Semilat**). Our semantics for Datafun is built over the adjunctions between **Set**, **Poset**, and **Semilat**; can we extend this to cover the other adjunctions of the Boom hierarchy, and uncover a language for expressing queries over these three data structures and their corresponding varieties of aggregation?

6.2 Lessons and surprises

Research is a process of discovery: sometimes you find what you expected, sometimes you do not; sometimes you find more questions. This section presents some insights gained and questions raised in the course of writing this dissertation which, while not entirely novel, at least surprised this author.

Change minimization and precise differences The need to minimize changes to avoid junk piling up during seminaïve fixed point iteration and eliminating any asymptotic speed-up is obvious in retrospect (see §4.3), but was overlooked by Arntzenius and Krishnaswami (2020), their reviewers, and the examiners of the initial version of this dissertation. The author only realized it when an undergraduate student pointed out an instance of it in the Q & A for a talk given at POPL 2020.

The root of this oversight is an early decision, when applying the incremental λ-calculus to Datafun, to *not* compute precise changes – in particular, to let $\delta(e \cup f) = \delta e \cup \delta f$ instead of the more precise $(\delta e \setminus f) \cup (\delta f \setminus e)$, because the former, simpler expression does less work and avoids recomputing e and f. In light of the need for change minimization, this decision is questionable. But because our efforts in chapter 3 focused almost entirely on proving correctness, and because our initial benchmark in chapter 4 (a linear graph) has at most one path between two nodes and thus fails to trigger this issue, we failed to notice this inefficiency. Change minimization addresses this issue, but the question remains: might it be better to just compute precise changes in the first place?

Expressiveness versus tractability There is a tradeoff in programming language design between *expressiveness* and *tractability:* the more powerful the language, the more complex it becomes to reason about programs in that language. Datalog's power comes from its limitations: it restricts logic programming to the bare minimum – finite relations – and in return gains a rich theory of implementation and optimisation techniques. Datafun is an attempt to loosen Datalog's restrictions; in hindsight it's clear we might run into issues of tractability.

For instance, we have already seen that by allowing monadic set-comprehensions, a natural choice from a functional programming standpoint and one made without careful consideration on the part of this author, we complicated the task of query planning, ignoring

[2] The other layer of the Boom hierarchy is *trees,* specifically binary trees with data only at the leaves: the free magma. Few useful aggregations form magmas, and although trees are ubiquitous in computing, few are of this particular form.

hard-won wisdom from the field of databases incorporated implicitly into the design of Datalog.

More generally, by deeply integrating functional and logic programming, Datafun extends Datalog's expressivity – but is it worth the price? There are more conservative ways to address Datalog's shortcomings which may require less reinvention of existing techniques; for instance, using code generation or staged programming to allow modular code re-use; or the approach taken by Flix: two interlocking but separate logic and functional languages. The only examples we have given that deeply intertwine the functional and logic features of Datafun are the regular expression combinators from §2.2.2–2.2.3, which represent regular expressions as functions producing relations and regular expression constructions as higher-order functions (combinators). This is cute, but as justifications for a dissertation's worth of work go it is fairly thin. Higher-order programming sometimes unlocks unusually powerful or concise solutions to existing problems; can we find other motivating examples that take advantage of Datafun's unique feature set?

The diversity and unity of incremental computation The problem of seminaïve evaluation is an instance of incremental computation – one instance among many. As we explored in §5.2, there appear to be as many approaches to incremental computation as there are applications of it, including a zoo of incremental build systems, several varieties of self-adjusting computation, and multiple approaches to incremental maintenance in database systems. Even user interfaces involve incremental computation: for performance reasons, web UI libraries like React avoid re-rendering UI components whose state has not changed.[3]

These diverse systems nonetheless share a small set of core techniques: tracking dependencies so we know what needs to be updated, caching intermediate values to re-use them when they don't change, and propagating changes – perhaps as all-or-nothing updates, perhaps as diffs. This latter apparent difference (between what we called *dependency tracking* and *finite differencing* approaches) can, however, be resolved by seeing all-or-nothing updates as a degenerate kind of diff, as we showed in §5.2.1. The theory of change structures – originated by Cai et al. (2014) and further developed by Giarrusso (2020), Alvarez-Picallo (2020), and in the present work – hints at the beginnings of a unified theory of incremental computation. However, all these works use slightly but crucially differing notions of *change structure;* perhaps incremental computation is destined never to be unified at all.

6.3 Successes summarized

So far in this chapter we have largely explored how this effort falls short or could be improved on in future work. However, chapters 2 and 3 respectively represent significant successes for this semantic-deconstruction approach; if we were to distill this dissertation into their key ideas, it would be these:

Model monotonicity with modal types. Datalog can be summarized as *relational algebra plus stratified recursive queries.* Modulo implementation subtleties, relational algebra embeds straightforwardly in a functional language via finite sets and set comprehensions. We

[3] https://web.archive.org/web/20220223155533/https://reactjs.org/; see also https://web.archive.org/web/20220907152744/https://blog.janestreet.com/incrementality-and-the-web/ for a discussion of incrementality in web UI libraries.

have shown that stratified recursive queries also embed nicely, so long as we locate our semantics in **Poset** to capture compositional reasoning about monotonicity. The main difficulty is the interaction of monotone and non-monotone functions; this arises from the discreteness comonad $\Box$, and can be handled with a simple modal type system.

To find fixed points faster, incrementalize! Finding a fixed point by iteration involves repeatedly changing a function's input to match its changing output. Doing this naïvely is asymptotically inefficient; to do it efficiently, we must efficiently propagate *changes*. This is not only the essence of seminaïve evaluation in Datalog, but an instance of a greater problem of automatic incremental computation. Prior work on the incremental λ-calculus shows that incremental computation can be achieved in higher-order languages; we have extended it to Datafun and shown that by modifying it to consider only *increasing* changes, it gives rise to seminaïve evaluation.

Appendix A

Proofs omitted from main text

We state these lemmas and theorems in dependency order, so that nothing is used before it has been proven. This is not always the order in which they are stated in the text.

A.1 Datafun

Theorems and lemmas from chapter 2.

Lemma 31. If $X : A \in \Gamma$ or $x :: A \in \Gamma$ and $\Gamma \sqsubseteq \Delta$ then $X : A \in \Delta$ or $x :: A \in \Delta$.

Proof. Recall that although we write them differently we regard X and x as the same variable. Let H stand for the hypothesis in our assumption, either $X : A$ or $x :: A$ respectively. Then by induction on the derivation of $\Gamma \sqsubseteq \Delta$:

Case $\varepsilon \sqsubseteq \varepsilon$. By contradiction, since $H \in \varepsilon$ is impossible.

Case $\dfrac{\Gamma' \sqsubseteq \Delta'}{\Gamma', H' \sqsubseteq \Delta', H'}$. If $H = H'$ we are done. Otherwise, $H \in \Gamma'$, so apply the IH.

Case $\dfrac{\Gamma \sqsubseteq \Delta'}{\Gamma \sqsubseteq \Delta', H}$. Apply the IH.

Case $\dfrac{\Gamma' \sqsubseteq \Delta'}{\Gamma',\ X : A \sqsubseteq \Delta', x :: A}$. If $H = X : A$ we are done. Otherwise, $H \in \Gamma'$, so apply the IH.

□

Lemma 32. If $\Gamma \sqsubseteq \Delta$ then $\lceil \Gamma \rceil \sqsubseteq \lceil \Delta \rceil$.

Proof. By induction on $\Gamma \sqsubseteq \Delta$:

Case $\varepsilon \sqsubseteq \varepsilon$. Immediate.

Case $\dfrac{\Gamma' \sqsubseteq \Delta'}{\Gamma', H \sqsubseteq \Delta', H}$.

Either H is discrete $x :: A$, in which case $\lceil \Gamma', H \rceil = \lceil \Gamma' \rceil, H \sqsubseteq \lceil \Delta' \rceil, H = \lceil \Delta', H \rceil$ by CONS and our inductive hypothesis; or H is monotone $X : A$, in which case $\lceil \Gamma', H \rceil = \lceil \Gamma' \rceil \sqsubseteq \lceil \Delta' \rceil = \lceil \Delta', H \rceil$ by our inductive hypothesis alone.

Case $\dfrac{\Gamma \sqsubseteq \Delta'}{\Gamma \sqsubseteq \Delta', H}$.

Then $\lceil \Gamma \rceil \sqsubseteq \lceil \Delta' \rceil$ by our inductive hypothesis. Depending upon whether H is monotone or discrete, we have either $\lceil \Delta', H \rceil = \lceil \Delta' \rceil$ (in which case our inductive hypothesis suffices) or $\lceil \Delta', H \rceil = \lceil \Delta' \rceil, H$, in which case by CONS and transitivity $\lceil \Gamma \rceil \sqsubseteq \lceil \Delta' \rceil, H$.

Case $\dfrac{\Gamma' \sqsubseteq \Delta'}{\Gamma', X : A \sqsubseteq \Delta', x :: A}$.

Then $\lceil \Gamma', X : A \rceil = \lceil \Gamma' \rceil$ while $\lceil \Delta', x :: A \rceil = \lceil \Delta' \rceil, x :: A$. By our IH, we have $\lceil \Gamma' \rceil \sqsubseteq \lceil \Delta' \rceil$, and therefore by DROP we have $\lceil \Gamma' \rceil \sqsubseteq \lceil \Delta' \rceil, x :: A$ as desired.

□

A.2 Seminaïve evaluation

Theorems and lemmas from chapter 3.

Lemma 19. $\Phi \underset{EQ}{A} = \underset{EQ}{A}$ for all equality types $\underset{EQ}{A}$.

Proof. Induct on $\underset{EQ}{A}$ applying the equations in figure 3.1, recalling from figure 2.1 that the grammar of equality types is $\underset{EQ}{A} ::= 1 \mid \underset{EQ}{A} \times \underset{EQ}{B} \mid \underset{EQ}{A} + \underset{EQ}{B} \mid \{\underset{EQ}{A}\}$. □

Lemma 20. At each semilattice type L, we have $\Delta L = L$.

Proof. Induct on L applying the equations in figure 3.1, recalling from figure 2.1 that the grammar of semilattice types is $L ::= 1 \mid L_1 \times L_2 \mid \{\underset{EQ}{A}\}$. □

Theorem 22 (Weakening)**.** If $\Delta \sqsupseteq \Gamma$ and $\Gamma \vdash e : A$ then $\Delta \vdash e : A$.

Proof. By induction on the derivation of $\Gamma \vdash e : A$.

Cases $\dfrac{X : A \in \Gamma}{\Gamma \vdash X : A}$ $\dfrac{x :: A \in \Gamma}{\Gamma \vdash x : A}$. By lemma 31.

Cases $\dfrac{}{\Gamma \vdash () : 1}$ $\dfrac{}{\Gamma \vdash \bot : L}$. Trivial.

Cases where the premises have the same context as the conclusion, namely:

$$\frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash f : A}{\Gamma \vdash e\ f : B} \qquad \frac{(\Gamma \vdash e_i : A_i)_i}{\Gamma \vdash (e_1, e_2) : A_1 \times A_2} \qquad \frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \pi_i\ e : A_i}$$

$$\frac{(\Gamma \vdash e_i : L)_i}{\Gamma \vdash e_1 \vee e_2 : L} \qquad \frac{\Gamma \vdash e : \Box(A + B)}{\Gamma \vdash \mathsf{split}\ e : \Box A + \Box B} \qquad \frac{\Gamma \vdash e : \Box(\underset{\text{FIX}}{L} \to \underset{\text{FIX}}{L})}{\Gamma \vdash \mathsf{fix}\ e : \underset{\text{FIX}}{L}}$$

Apply the same typing rule to our inductive hypotheses.

Cases where the premises add hypotheses to the context, namely:

$$\frac{\Gamma, X : A \vdash e : B}{\Gamma \vdash \lambda X.\ e : A \to B} \qquad \frac{\Gamma \vdash e : \Box A \qquad \Gamma, x :: A \vdash f : B}{\Gamma \vdash \textbf{let}\ [x] = e\ \textbf{in}\ f : B}$$

$$\frac{\Gamma \vdash e : A_1 + A_2 \qquad (\Gamma, X_i : A_i \vdash f_i : B)_i}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ (\mathsf{in}_i\ X_i \to f_i)_i : B} \qquad \frac{\Gamma \vdash e : \{A\} \qquad \Gamma, x :: A \vdash f : L}{\Gamma \vdash \textbf{for}\ (x \in e)\ f : L}$$

Apply the inductive hypotheses, using CONS when necessary to show that the modified contexts also satisfy our precondition, for example, $\Delta, X : A \sqsupseteq \Gamma, X : A$.

Case where the premises strip the context, namely:

$$\frac{\lceil\Gamma\rceil \vdash e : A}{\Gamma \vdash [e] : \Box A} \qquad \frac{(\lceil\Gamma\rceil \vdash e_i : \underset{\text{EQ}}{A})_i}{\Gamma \vdash \{e_i\}_i : \{\underset{\text{EQ}}{A}\}} \qquad \frac{(\lceil\Gamma\rceil \vdash e_i : \underset{\text{EQ}}{A})_i}{\Gamma \vdash e_1 = e_2 : \mathsf{bool}} \qquad \frac{\lceil\Gamma\rceil \vdash e : \{1\}}{\Gamma \vdash \mathsf{empty?}\ e : 1 + 1}$$

Then $\lceil\Delta\rceil \sqsupseteq \lceil\Gamma\rceil$ by lemma 32, so we apply the inductive hypotheses.

□

Theorem 21 (Well-typedness of ϕ, δ)**.** If $\Gamma \vdash e : A$, then ϕe and δe have the following types:

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

Proof. By induction on the derivation of $\Gamma \vdash e : A$, although as we'll see shortly we will need weakening (theorem 22) in some places. □

Lemma 25 (Equality changes)**.** If $dx \rhd_{\underset{\text{EQ}}{A}} x \lightning a \to y \lightning b$ then $x = a$ and $y = b$.

Proof. By induction on $\underset{\text{EQ}}{A}$, applying the definition from figure 3.6:

Case 1. Trivial.

Case $\underset{EQ}{A} \times \underset{EQ}{B}$. Then our assumption is equivalent to

$$(dx_1, dx_2) \rhd_{\underset{EQ}{A} \times \underset{EQ}{B}} (x_1, x_2) \lightning (a_1, a_2) \to (y_1, y_2) \lightning (b_1, b_2)$$

and by unfolding this we have $dx_1 \rhd_{\underset{EQ}{A}} x_1 \lightning a_1 \to y_1 \lightning b_1$ and $dx_2 \rhd_{\underset{EQ}{B}} x_2 \lightning a_2 \to y_2 \lightning b_2$, which by our inductive hypotheses show $x_1 = a_1$, $y_1 = b_1$ and $x_2 = a_2$, $y_2 = b_2$, which suffices.

Case $\underset{EQ}{A_1} + \underset{EQ}{A_2}$. Then for some $i \in \{1, 2\}$ our assumption is equivalent to

$$\mathsf{in}_i\, dx \rhd_{\underset{EQ}{A_1} + \underset{EQ}{A_2}} \mathsf{in}_i\, x \lightning \mathsf{in}_i\, a \to \mathsf{in}_i\, y \lightning \mathsf{in}_i\, b$$

and by unfolding this we have $dx \rhd_{\underset{EQ}{A_i}} x \lightning a \to y \lightning b$, which by our inductive hypothesis implies $x = a$ and $y = b$, which suffices.

Case $\{\underset{EQ}{A}\}$. Then our assumption unfolds to $(x, y, x \cup dx) = (a, b, y)$, which suffices.

□

Lemma 26 (Dummy is zero at eqtypes)**.** If $x \in [\![\underset{EQ}{A}]\!]$ then $\mathsf{dummy}\, x \rhd_{\underset{EQ}{A}} x \lightning x \to x \lightning x$.

Proof. By induction on $\underset{EQ}{A}$, applying the definitions of dummy and $\mathsf{dummy}\, x \rhd_{\underset{EQ}{A}} x \lightning x \to x \lightning x$ (figures 3.5 and 3.6).

Case 1. Trivial.

Case $\underset{EQ}{A} \times \underset{EQ}{B}$. Letting $x = (y, z)$, we have $\mathsf{dummy}\, x = \mathsf{dummy}\, (y, z) = (\mathsf{dummy}\, y, \mathsf{dummy}\, z)$. By our inductive hypotheses, we have $\mathsf{dummy}\, y \rhd_{\underset{EQ}{A}} y \lightning y \to y \lightning y$ and likewise for z. By definition this shows that

$$(\mathsf{dummy}\, y, \mathsf{dummy}\, z) \rhd_{\underset{EQ}{A} \times \underset{EQ}{B}} (y, z) \lightning (y, z) \to (y, z) \lightning (y, z)$$

as desired.

Case $\underset{EQ}{A_1} + \underset{EQ}{A_2}$. Without loss of generality we have $x = \mathsf{in}_i\, y$ for some $i \in \{1, 2\}$. Applying the definition of dummy we have $\mathsf{dummy}\, x = \mathsf{in}_i\, (\mathsf{dummy}\, y)$. By our inductive hypothesis we have $\mathsf{dummy}\, y \rhd_{\underset{EQ}{A_i}} y \lightning y \to y \lightning y$, which suffices to show

$$\mathsf{in}_i\, (\mathsf{dummy}\, y) \rhd_{\underset{EQ}{A_i}} \mathsf{in}_i\, y \lightning \mathsf{in}_i\, y \to \mathsf{in}_i\, y \lightning \mathsf{in}_i\, y$$

as desired.

Case $\{\underset{EQ}{A}\}$. Unfolding our theorem's definition, we need to show that $(x, x, x \cup \mathsf{dummy}\, x) = (x, x, x)$, or in other words $x = x \cup \mathsf{dummy}_{\{\underset{EQ}{A}\}}\, x$, which is trivial since $\mathsf{dummy}_{\{\underset{EQ}{A}\}}\, x = \{\}$.

□

Lemma 28 (Discrete contexts don't change)**.** If $() \rhd_{\lceil\Gamma\rceil} \gamma \lightning \rho \to \gamma' \lightning \rho'$ then $\gamma = \gamma'$ and $\rho = \rho'$.

Proof. All variables in the stripped contexts are discrete, and therefore the logical relation for discrete variables in contexts, which invokes the logical relation at $\Box$ type, requires their corresponding components be equal. □

Lemma 29 (Context stripping)**.** If $d\gamma \rhd_\Gamma \gamma \lightning \rho \to \gamma' \lightning \rho'$ then

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

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

Proof. Immediate from the definitions. □

Lemma 33 (Applying box)**.** Given $\lceil\Gamma\rceil \vdash e : A$ and $\gamma : [\![\Gamma]\!]$,

$$[\![[e]]\!]\ \gamma = \mathsf{box}_\Gamma([\![e]\!])(\gamma) = [\![e]\!]\ (\mathsf{strip}_\Gamma(\gamma))$$

Proof. Recall that the box comonad $\Box$'s functorial action, duplication map $\delta_A : \Box A \to \Box\Box A$, and distribution $\mathsf{dist}^\times_\Box : \prod_i \Box A_i \to \Box \prod_i A_i$ are all no-ops. Then:

$$\begin{aligned}
[\![[e]]\!]\ \gamma &= \mathsf{box}_\Gamma([\![e]\!])(\gamma) && \text{definition of } [\![[e]]\!] \\
&= \Box[\![e]\!](\mathsf{dist}^\times_\Box(\delta_A(\gamma_x)_{x::A\in\Gamma})) && \text{definition of box} \\
&= [\![e]\!](\gamma_x)_{x::A\in\Gamma} && \text{no-ops} \\
&= [\![e]\!](\mathsf{strip}_\Gamma(\gamma)) && \text{definition of strip}
\end{aligned}$$

□

Lemma 34 (Correctness of semifix)**.** If $g' \rhd_{\underset{\mathsf{FIX}}{\mathrm{L}} \to \underset{\mathsf{FIX}}{\mathrm{L}}} g \lightning f \to g \lightning f$, then $\mathsf{semifix}\ (g, g') = \mathsf{fix}\ f$.

Proof. First, let's expand our assumption:

$$(\forall dx \rhd_{\underset{\mathsf{FIX}}{\mathrm{L}}} x \lightning a \to y \lightning b)\ g'\ x\ dx \rhd_{\underset{\mathsf{FIX}}{\mathrm{L}}} g\ x \lightning f\ a \to g\ y \lightning f\ b$$

If we apply lemma 27 and simplify slightly, this is equivalent to:

$$(\forall x, dx : \underset{\mathsf{FIX}}{\mathrm{L}})\ g\ x = f\ x \text{ and } g\ (x \vee dx) = f\ (x \vee dx) = g\ x \vee g'\ x\ dx \tag{A.1}$$

This in particular implies that $f = g$.

Now, recall that $\mathsf{fix}\ f$ is defined as the limit $\bigvee_i f^i\ \bot$ of the iterations of f, while $\mathsf{semifix}\ (g, g')$ is the limit $\bigvee_i x_i$ of the sequence x_i given by:

$$\begin{aligned}
x_0 &= \bot & x_{i+1} &= x_i \vee dx_i \\
dx_0 &= g\ \bot & dx_{i+1} &= g'\ x_i\ dx_i
\end{aligned}$$

Thus it suffices to show that $x_i = f^i\ \bot$, which we will show inductively, along with $x_i \vee dx_i = f\ x_i$. To establish the base case, $x_0 = \bot = f^0\ \bot$ by definition and $x_0 \vee dx_0 = \bot \vee g\ \bot = f\ \bot$ because $g = f$. Inductively assuming that $x_i = f^i\ \bot$ and $x_i \vee dx_i = f\ x_i$, we have that $x_{i+1} = x_i \vee dx_i = f\ x_i = f\ (f^i\ \bot) = f^{i+1}\ \bot$ as desired, and finally:

$$\begin{aligned}
x_{i+1} \vee dx_{i+1} &= (x_i \vee dx_i) \vee g'\ x_i\ dx_i && \text{expanding definitions} \\
&= f\ x_i \vee g'\ x_i\ dx_i && \text{inductive hypothesis} \\
&= g\ x_i \vee g'\ x_i\ dx_i && \text{because } f = g \\
&= f\ (x_i \vee dx_i) && \text{by equation A.1} \\
&= f\ x_{i+1} && \text{definition of } x_{i+1}
\end{aligned}$$

□

Theorem 24 (Fundamental property)**.** If $\Gamma \vdash e : A$ and $d\gamma \rhd_\Gamma \gamma \lightning \rho \to \gamma' \lightning \rho'$ then

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

Proof. By induction on the derivation of $\Gamma \vdash e : A$. We will refer to the other premise $d\gamma \rhd_\Gamma \gamma \lightning \rho \to \gamma' \lightning \rho'$ as simply "the assumption".

Case $\dfrac{X : A \in \Gamma}{\Gamma \vdash X : A}$. We wish to show:

$$\begin{aligned}
&[\![\delta X]\!]\ (\gamma, d\gamma) \rhd_A [\![\phi X]\!]\ \gamma \lightning [\![X]\!]\ \rho \to [\![\phi X]\!]\ \gamma' \lightning [\![X]\!]\ \rho' \\
\iff &[\![DX]\!]\ (\gamma, d\gamma) \rhd_A [\![X]\!]\ \gamma \lightning [\![X]\!]\ \rho \to [\![X]\!]\ \gamma' \lightning [\![X]\!]\ \rho' \\
\iff &d\gamma_{DX} \rhd_A \gamma_X \lightning \rho_X \to \gamma'_X \lightning \rho'_X
\end{aligned}$$

which follows from the definition of our assumption.

Case $\dfrac{x :: A \in \Gamma}{\Gamma \vdash x : A}$. We wish to show:

$$\begin{aligned}
&[\![\delta x]\!]\ (\gamma, d\gamma) \rhd_A [\![\phi x]\!]\ \gamma \lightning [\![x]\!]\ \rho \to [\![\phi x]\!]\ \gamma' \lightning [\![x]\!]\ \rho' \\
\iff &[\![dx]\!]\ (\gamma, d\gamma) \rhd_A [\![x]\!]\ \gamma \lightning [\![x]\!]\ \rho \to [\![x]\!]\ \gamma' \lightning [\![x]\!]\ \rho' \\
\iff &\gamma_{dx} \rhd_A \gamma_x \lightning \rho_x \to \gamma'_x \lightning \rho'_x
\end{aligned}$$

If we apply our assumption we get:

$$\begin{aligned}
&d\gamma \rhd_\Gamma \gamma \lightning \rho \to \gamma' \lightning \rho' \\
\implies &() \rhd_{\Box A} (\gamma_x, \gamma_{dx}) \lightning \rho_x \to (\gamma'_x, \gamma'_{dx}) \lightning \rho'_x \\
\implies &\gamma_{dx} \rhd_A \gamma_x \lightning \rho_x \to \gamma'_x \lightning \rho'_x
\end{aligned}$$

as desired.

Case $\dfrac{\Gamma, X : A \vdash e : B}{\Gamma \vdash \lambda X.\ e : A \to B}$ Recall that $\phi(\lambda X.\ e) = \lambda X.\ \phi e$ and $\delta(\lambda X.\ e) = \lambda[x].\ \lambda DX.\ \delta e$. We wish to show

$$[\![\lambda[x].\ \lambda DX.\ \delta e]\!]\ (\gamma, d\gamma) \rhd_{A\to B} [\![\lambda X.\ \phi e]\!]\ \gamma \lightning [\![\lambda X.\ e]\!]\ \rho \to [\![\lambda X.\ \phi e]\!]\ \gamma' \lightning [\![\lambda X.\ e]\!]\ \rho'$$

Applying the definition of the logical relation at $A \to B$, it suffices to assume (a) $dx \rhd_A x \lightning a \to y \lightning b$ and prove

$$[\![\lambda[x].\ \lambda DX.\ \delta e]\!]\ (\gamma, d\gamma)\ x\ dx \rhd_B [\![\lambda X.\ \phi e]\!]\ \gamma\ x \lightning [\![\lambda X.\ e]\!]\ \rho\ a \to [\![\lambda X.\ \phi e]\!]\ \gamma'\ y \lightning [\![\lambda X.\ e]\!]\ \rho'\ b$$

which, by calculation, is:

$$[\![\delta e]\!]\ \sigma \rhd_B [\![\phi e]\!]\ (\gamma, X \mapsto x) \lightning [\![e]\!]\ (\rho, X \mapsto a) \to [\![\phi e]\!]\ (\gamma, X \mapsto y) \lightning [\![e]\!]\ (\rho, X \mapsto b)$$

where $\sigma = (\gamma, d\gamma, x \mapsto x, DX \mapsto dx)$. This follows from our inductive hypothesis if we can show that:

$$(d\gamma, DX \mapsto dx) \rhd_{\Gamma, X:A} (\gamma, X \mapsto x) \lightning (\rho, X \mapsto a) \to (\gamma', X \mapsto y) \lightning (\rho', X \mapsto b)$$

and this follows from our assumption and (a).

Case $\dfrac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash f : A}{\Gamma \vdash e\ f : B}$. Recall that $\phi(e\ f) = \phi e\ \phi f$ and $\delta(e\ f) = \delta e\ [\phi f]\ \delta f$. Thus we wish to show:

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

Let:

$$\begin{array}{llll} dx = [\![\delta e]\!]\ (\gamma, d\gamma) & & dy = [\![\delta f]\!]\ (\gamma, d\gamma) & \\ x = [\![\phi e]\!]\ \gamma & a = [\![e]\!]\ \rho & y = [\![\phi f]\!]\ \gamma & b = [\![f]\!]\ \rho \\ x' = [\![\phi e]\!]\ \gamma' & a' = [\![e]\!]\ \rho' & y' = [\![\phi f]\!]\ \gamma' & b' = [\![f]\!]\ \rho' \end{array}$$

By our IH for f we have $dy \rhd_A y \lightning b \to y' \lightning b'$. By this and our IH for e we have $dx\ y\ dy \rhd_B x\ y \lightning a\ b \to x'\ y' \lightning a'\ b'$. By calculation, this is equal to what we wish to show.

Case $\dfrac{}{\Gamma \vdash () : 1}$. We wish to show:

$$\begin{aligned} & [\![()]\!]\ (\gamma, d\gamma) \rhd_1 [\![()]\!]\ \gamma \lightning [\![()]\!]\ \rho \to [\![()]\!]\ \gamma' \lightning [\![()]\!]\ \rho' \\ \iff & ()\ \rhd_1\ ()\ \lightning\ () \to ()\ \lightning\ () \\ \iff & \top \end{aligned}$$

Case $\dfrac{(\Gamma \vdash e_i : A_i)_i}{\Gamma \vdash (e_1, e_2) : A_1 \times A_2}$: Recall that $\phi((e_1, e_2)) = (\phi e_1, \phi e_2)$ and $\delta((e_1, e_2)) = (\delta e_1, \delta e_2)$. Thus we wish to show:

$$[\![(\delta e_1, \delta e_2)]\!]\ (\gamma, d\gamma) \rhd_{A_1 \times A_2} [\![(\phi e_1, \phi e_2)]\!]\ \gamma \lightning [\![(e_1, e_2)]\!]\ \rho \to [\![(\phi e_1, \phi e_2)]\!]\ \gamma' \lightning [\![(e_1, e_2)]\!]\ \rho'$$

Since in general $[\![(f_1, f_2)]\!]\ \sigma = ([\![f_1]\!]\ \sigma, [\![f_2]\!]\ \sigma)$, applying the definition of the LR at $A_1 \times A_2$ this is equivalent to:

$$(\forall i)\ [\![\delta e_i]\!]\ (\gamma, d\gamma) \rhd_{A_i} [\![\phi e_i]\!]\ \gamma \lightning [\![e_i]\!]\ \rho \to [\![\phi e_i]\!]\ \gamma' \lightning [\![e_i]\!]\ \rho'$$

which holds by our IH.

Case $\dfrac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \pi_i\ e : A_i}$. Recall that $\phi(\pi_i\ e) = \pi_i\ \phi e$ and $\delta(\pi_i\ e) = \pi_i\ \delta e$ and observe that $[\![\pi_i\ f]\!]\ \sigma = \pi_i([\![f]\!]\ \sigma)$. Applying this, what we wish to show is

$$\pi_i\ ([\![\delta e]\!]\ (\gamma, d\gamma)) \rhd_{A_i} \pi_i\ ([\![\phi e]\!]\ \gamma) \lightning \pi_i\ ([\![e]\!]\ \rho) \to \pi_i\ ([\![\phi e]\!]\ \gamma') \lightning \pi_i\ ([\![e]\!]\ \rho')$$

which is a direct consequence of our IH.

Case $\dfrac{\Gamma \vdash e : A_i}{\Gamma \vdash in_i\ e : A_1 + A_2}$. Recall that $\phi(\mathsf{in}_i\ e) = \mathsf{in}_i\ \phi e$ and $\delta(\mathsf{in}_i\ e) = \mathsf{in}_i\ \delta e$. Observe that in general $[\![\mathsf{in}_i\ f]\!]\ \sigma = \mathsf{in}_i\ ([\![f]\!]\ \sigma)$ and therefore what we wish to show is equivalent to:

$$\mathsf{in}_i\ ([\![\delta e]\!]\ (\gamma, d\gamma)) \rhd_{A_1 + A_2} \mathsf{in}_i\ ([\![\phi e]\!]\ \gamma) \lightning \mathsf{in}_i\ ([\![e]\!]\ \rho) \to \mathsf{in}_i\ ([\![\phi e]\!]\ \gamma') \lightning \mathsf{in}_i\ ([\![e]\!]\ \rho')$$

which is by definition equivalent to our inductive hypothesis.

Case $\dfrac{}{\Gamma \vdash \bot : L}$. Recall that $\phi\bot = \bot$ and $\delta\bot = \bot$ and $[\![\bot]\!]\ \sigma = \bot$. Thus it STS $\bot \rhd_L \bot \lightning \bot \to \bot \lightning \bot$, which holds by lemma 27.

Case $\dfrac{(\Gamma \vdash e_i : L)_i}{\Gamma \vdash e_1 \vee e_2 : L}$. Recall that $\phi(e_1 \vee e_2) = \phi e_1 \vee \phi e_2$. We wish to show:

$$[\![\delta e_1 \vee \delta e_2]\!]\ (\gamma, d\gamma) \rhd_L [\![\phi e_1 \vee \phi e_2]\!]\ \gamma \lightning [\![e_1 \vee e_2]\!]\ \rho \to [\![\phi e_1 \vee \phi e_2]\!]\ \gamma' \lightning [\![e_1 \vee e_2]\!]\ \rho'$$

Let:

$$dx_i = [\![\delta e_i]\!]\ (\gamma, d\gamma) \quad x_i = [\![\phi e_i]\!]\ \gamma \quad x_i' = [\![\phi e_i]\!]\ \gamma' \quad a_i = [\![e_i]\!]\ \rho \quad a_i' = [\![e_i]\!]\ \rho'$$

Since $[\![f \vee g]\!]\ \sigma = [\![f]\!]\ \sigma \vee [\![g]\!]\ \sigma$, what we wish to show is equivalent to:

$$dx_1 \vee dx_2 \rhd_L x_1 \vee x_2 \lightning a_1 \vee a_2 \to x_1' \vee x_2' \lightning a_1' \vee a_2'$$

Applying lemma 27 this is equivalent to:

$$x_1 \vee x_2 = a_1 \vee a_2 \qquad x_1' \vee x_2' = a_1' \vee a_2' = x_1 \vee x_2 \vee dx_1 \vee dx_2$$

By our IH and assumption and lemma 27 we have:

$$x_i = a_i \qquad x_i' = a_i' = x_i \vee dx_i$$

which suffices by associativity and commutativity of $\vee$.

Case $\dfrac{(\Gamma \vdash e_i : A)_{i\ \text{EQ}}}{\Gamma \vdash \{e_i\}_i : \{A\}_{\text{EQ}}}$. Recall that $\phi\{e_i\}_i = \{\phi e_i\}_i$ and $\delta\{e_i\}_i = \bot$. Noting that $[\![\bot]\!]\ (\gamma, d\gamma) = \bot$, by lemma 27 what we want to show is equivalent to:

$$[\![\{\phi e_i\}_i]\!]\ \gamma = [\![\{e_i\}_i]\!]\ \rho = [\![\{\phi e_i\}_i]\!]\ \gamma' = [\![\{e_i\}_i]\!]\ \rho'$$

Note that $[\![\{f_i\}_i]\!]\ \sigma = \bigvee_i \{\text{box}_\Gamma([\![f_i]\!])(\sigma)\} = \{[\![f_i]\!]\ (\text{strip}\ \sigma)\}_i$. Also observe that by lemmas 28 and 29 and our assumption, we have

$$() \rhd_{\lceil\Gamma\rceil} \text{strip}\ \gamma \lightning \text{strip}\ \rho \to \text{strip}\ \gamma' \lightning \text{strip}\ \rho' \tag{A.2}$$

$$\text{strip}\ \gamma = \text{strip}\ \gamma' \quad \text{and} \quad \text{strip}\ \rho = \text{strip}\ \rho' \tag{A.3}$$

So applying equation A.3 it suffices to show that $\{[\![\phi e_i]\!]\ (\text{strip}\ \gamma)\}_i = \{[\![e_i]\!]\ (\text{strip}\ \rho)\}_i$, for which it suffices to show $[\![\phi e_i]\!]\ (\text{strip}\ \gamma) = [\![e_i]\!]\ (\text{strip}\ \rho)$. This holds by our IH for e_i and equation A.2 and lemma 25.

Case $\dfrac{(\lceil\Gamma\rceil \vdash e_i : A)_{i\ \text{EQ}}}{\Gamma \vdash e_1 = e_2 : \mathit{bool}}$. Recall that $\phi(e_1 = e_2) = \phi e_1 = \phi e_2$ and $\delta(e_1 = e_2) = \bot$. Thus what we wish to show is:

$$[\![\bot]\!]\ (\gamma, d\gamma) \rhd_{\text{bool}} [\![\phi e_1 = \phi e_2]\!]\ \gamma \lightning [\![e_1 = e_2]\!]\ \rho \to [\![\phi e_1 = \phi e_2]\!]\ \gamma' \lightning [\![e_1 = e_2]\!]\ \rho'$$

Observing that $[\![\bot]\!]\ (\gamma, d\gamma) = \emptyset$ and applying the definition of the logic relation for $\text{bool} = \{1\}$, this is equivalent to:

$$[\![\phi e_1 = \phi e_2]\!]\ \gamma = [\![\phi e_1 = \phi e_2]\!]\ \gamma' = [\![e_1 = e_2]\!]\ \rho = [\![e_1 = e_2]\!]\ \rho'$$

Observe by calculation that

$$[\![f_1 = f_2]\!]\ \sigma = \begin{cases} \{()\} & \text{if } [\![f_1]\!]\ (\text{strip}\ \sigma) = [\![f_2]\!]\ (\text{strip}\ \sigma) \\ \emptyset & \text{otherwise} \end{cases}$$

Thus it suffices to show $[\![\phi e_i]\!]\ (\text{strip}\ \gamma) = [\![\phi e_i]\!]\ (\text{strip}\ \gamma') = [\![e_i]\!]\ (\text{strip}\ \rho) = [\![e_i]\!]\ (\text{strip}\ \rho')$. By our assumption and lemmas 28 and 29 we know $\text{strip}\ \gamma = \text{strip}\ \gamma'$ and $\text{strip}\ \rho = \text{strip}\ \rho'$ and $() \rhd_{\lceil\Gamma\rceil} \text{strip}\ \gamma \not\downarrow \text{strip}\ \rho \to \text{strip}\ \gamma' \not\downarrow \text{strip}\ \rho'$. By the first two equalities it now suffices to show $[\![\phi e_i]\!]\ (\text{strip}\ \gamma) = [\![e_i]\!]\ (\text{strip}\ \rho)$. Applying our IH to the remaining third proposition we have

$$[\![\delta e_i]\!]\ (\text{strip}\ \gamma) \rhd_{\underset{\text{EQ}}{A}} [\![\phi e_i]\!]\ (\text{strip}\ \gamma) \not\downarrow [\![e_i]\!]\ (\text{strip}\ \rho) \to [\![\phi e_i]\!]\ (\text{strip}\ \gamma') \not\downarrow [\![e_i]\!]\ (\text{strip}\ \rho')$$

which by lemma 25 implies $[\![\phi e_i]\!]\ (\text{strip}\ \gamma) = [\![e_i]\!]\ (\text{strip}\ \rho)$ as desired.

Case $\dfrac{\lceil\Gamma\rceil \vdash e : \{1\}}{\Gamma \vdash \textit{empty?}\ e : 1 + 1}$. Recall that $\phi(\text{empty? } e) = \delta(\text{empty? } e) = \text{empty? } \phi e$. Note that by lemmas 28 and 29 we have $\text{strip}\ \gamma = \text{strip}\ \gamma'$ and $\text{strip}\ \rho = \text{strip}\ \rho'$ and $() \rhd_{\lceil\Gamma\rceil} \text{strip}\ \gamma \not\downarrow \text{strip}\ \rho \to \text{strip}\ \gamma' \not\downarrow \text{strip}\ \rho'$; applying this to our inductive hypothesis and invoking lemma 25 on the result, we have

$$[\![e]\!]\ (\text{strip}\ \rho) = [\![e]\!]\ (\text{strip}\ \rho') = [\![\phi e]\!]\ (\text{strip}\ \gamma) = [\![\phi e]\!]\ (\text{strip}\ \gamma')$$

Thus all are equal to the same value. Now, observe by calculation that

$$[\![\text{empty? } f]\!]\ \sigma = \begin{cases} \text{in}_i\ () & \text{if } [\![f]\!]\ (\text{strip}\ \sigma) = \emptyset \\ \text{in}_2\ () & \text{otherwise} \end{cases}$$

Thus, there is some i such that $[\![\text{empty? } e]\!]\ \rho = [\![\text{empty? } e]\!]\ \rho' = [\![\text{empty? } \phi e]\!]\ \gamma = [\![\text{empty? } \phi e]\!]\ \gamma' = \text{in}_i\ ()$ and we have $\text{in}_i\ () \rhd_{1+1} \text{in}_i\ () \not\downarrow \text{in}_i\ () \to \text{in}_i\ () \not\downarrow \text{in}_i\ ()$ as desired.

Case $\dfrac{\Gamma \vdash e : A_1 + A_2 \qquad (\Gamma, X_i : A_i \vdash f_i : B)_i}{\Gamma \vdash \textbf{\textit{case}}\ e\ \textbf{\textit{of}}\ (\textit{in}_i\ X \to f_i)_i : B}$. Recall that

$$\begin{aligned}
\phi(\textbf{case}\ e\ \textbf{of}\ (\text{in}_i\ X \to f_i)_i) &= \textbf{case}\ \phi e\ \textbf{of}\ (\text{in}_i\ X \to \phi f_i)_i \\
\delta(\textbf{case}\ e\ \textbf{of}\ (\text{in}_i\ X \to f_i)_i) &= \textbf{case}\ \text{split}\ [\phi e]\ \textbf{of} \\
&\qquad (\text{in}_i\ Y \to \textbf{let}\ [x] = Y\ \textbf{in} \\
&\qquad\quad (\lambda DX.\ \delta f_i) \\
&\qquad\quad (\textbf{case}\ \delta e\ \textbf{of}\ \text{in}_i\ DX \to DX \\
&\qquad\qquad \text{in}_{i+1 \bmod 2}\ _ \to \text{dummy}\ x))_i
\end{aligned}$$

Let $\rho_1 = \rho$, $\rho_2 = \rho'$, $\gamma_1 = \gamma$, $\gamma_2 = \gamma'$. By our inductive hypothesis for e there must be some $k \in \{1, 2\}$ and some dx, x_i, a_i such that $dx \rhd_{A_k} x_1 \not\downarrow a_1 \to x_2 \not\downarrow a_2$ and:

$$[\![e]\!]\ \rho_i = \text{in}_k\ a_i \qquad [\![\phi e]\!]\ \gamma_i = \text{in}_k\ x_i \qquad [\![\delta e]\!]\ (\gamma, d\gamma) = \text{in}_k\ dx$$

Applying this and our inductive hypothesis for f_k it will suffice to show that:

$$[\![\textbf{case } e \textbf{ of } (\mathsf{in}_i\, X \to f_i)_i]\!]\ \rho_i = [\![f_k]\!]\ (\rho_i, X \mapsto a_i) \tag{A.4}$$

$$[\![\phi(\textbf{case } e \textbf{ of } (\mathsf{in}_i\, X \to f_i)_i)]\!]\ \gamma_i = [\![\phi f_k]\!]\ (\gamma_i, X \mapsto x_i) \tag{A.5}$$

$$[\![\delta(\textbf{case } e \textbf{ of } (\mathsf{in}_i\, X \to f_i)_i)]\!]\ (\gamma, d\gamma) = [\![\delta f_k]\!]\ (\gamma, d\gamma, x \mapsto x_1, DX \mapsto dx) \tag{A.6}$$

Showing this is a matter of calculation. Equations A.4 and A.5 are fairly straightforward to calculate, so we only show equation A.6 in detail. Before starting, it will be useful to give some abbreviations for subterms of $\delta(\textbf{case } e \textbf{ of } (\mathsf{in}_i\, X \to f_i)_i)$:

$$h_i = \textbf{case } \delta e \textbf{ of } \mathsf{in}_i\, DX \to DX;\ \mathsf{in}_{i+1 \bmod 2}\ \_ \to \mathsf{dummy}\ x$$
$$g_i = \textbf{let } [x] = Y \textbf{ in } (\lambda DX.\ \delta f_i)\ h_i$$

It will also be useful to note the type of ϕe as it occurs in the sub-expression split $[\phi e]$ being immediately analyzed by $\delta(\textbf{case } e \textbf{ of } (\mathsf{in}_i\, X \to f_i)_i)$; it has been *weakened* to the type $\Box\Phi\Gamma \vdash \phi e : \Phi A_1 + \Phi A_2$.

$$\begin{aligned}
&[\![\delta(\textbf{case } e \textbf{ of } (\mathsf{in}_i\, X \to f_i)_i)]\!]\ (\gamma, d\gamma) \\
&= [[\![g_i]\!]]_i\ (\mathsf{dist}_+^{\times}((\gamma, d\gamma), [\![\mathsf{split}\ [\phi e]]\!]\ (\gamma, d\gamma))) \\
&= [[\![g_i]\!]]_i\ (\mathsf{dist}_+^{\times}((\gamma, d\gamma), \mathsf{dist}_+^{\Box}([\![[\phi e]]\!]\ (\gamma, d\gamma)))) \\
&= [[\![g_i]\!]]_i\ (\mathsf{dist}_+^{\times}((\gamma, d\gamma), [\![[\phi e]]\!]\ (\gamma, d\gamma))) && \mathsf{dist}_+^{\Box} \text{ is the identity} \\
&= [[\![g_i]\!]]_i\ (\mathsf{dist}_+^{\times}((\gamma, d\gamma), [\![\phi e]\!]\ (\mathsf{strip}_{\Box\Phi\Gamma}\ (\gamma, d\gamma)))) && \text{lemma 33} \\
&= [[\![g_i]\!]]_i\ (\mathsf{dist}_+^{\times}((\gamma, d\gamma), [\![e]\!]\ \gamma)) \\
&= [[\![g_i]\!]]_i\ (\mathsf{dist}_+^{\times}((\gamma, d\gamma), \mathsf{in}_k\, x_1)) \\
&= [[\![g_i]\!]]_i\ (\mathsf{in}_k\, (\gamma, d\gamma, Y \mapsto x_1)) \\
&= [\![g_k]\!]\ (\gamma, d\gamma, Y \mapsto x_1) \\
&= [\![\textbf{let } [x] = Y \textbf{ in } (\lambda DX.\ \delta f_k)\ h_k]\!]\ (\gamma, d\gamma, Y \mapsto x_1) \\
&= [\![(\lambda DX.\ \delta f_k)\ h_k]\!]\ (\gamma, d\gamma, x \mapsto x_1) \\
&= [\![(\lambda DX.\ \delta f_k)]\!]\ (\gamma, d\gamma, x \mapsto x_1)\ ([\![h_k]\!]\ (\gamma, d\gamma, x \mapsto x_1)) \\
&= (dx \mapsto [\![\delta f_k]\!]\ (\gamma, d\gamma, x \mapsto x_1, DX \mapsto dx)) \\
&\quad\ ([\![h_k]\!]\ (\gamma, d\gamma, x \mapsto x_1)) \\
&= [\![\delta f_k]\!]\ (\gamma, d\gamma, x \mapsto x_1, DX \mapsto ([\![h_k]\!]\ (\gamma, d\gamma, x \mapsto x_1)))
\end{aligned}$$

And therefore it suffices to show that $[\![h_k]\!]\ (\gamma, d\gamma, x \mapsto x_1) = dx$. Without loss of generality, assume $k = 1$:

$$\begin{aligned}
&[\![h_1]\!]\ (\gamma, d\gamma, x \mapsto x_1) \\
&= [\![\textbf{case } \delta e \textbf{ of } \mathsf{in}_1\, DX \to DX;\ \mathsf{in}_2\ _ \to \mathsf{dummy}\ x]\!]\ (\gamma, d\gamma, x \mapsto x_1) \\
&= [[\![DX]\!], [\![\mathsf{dummy}\ x]\!]]\ (\mathsf{dist}_+^{x}((\gamma, d\gamma, x \mapsto x_1), [\![\delta e]\!]\ (\gamma, d\gamma, x \mapsto x_1))) \\
&= [[\![DX]\!], [\![\mathsf{dummy}\ x]\!]]\ ((\gamma, d\gamma, x \mapsto x_1), (\mathsf{in}_1\, dx)) \\
&= [[\![DX]\!], [\![\mathsf{dummy}\ x]\!]]\ (\mathsf{in}_1\, (\gamma, d\gamma, x \mapsto x_1, DX \mapsto dx)) \\
&= [\![DX]\!]\ (\gamma, d\gamma, x \mapsto x_1, DX \mapsto dx) \\
&= dx
\end{aligned}$$

Which is what we wished to show.

Case $\dfrac{\lceil\Gamma\rceil \vdash e : A}{\Gamma \vdash [e] : \Box A}$, $\phi[e] = [(\phi e, \delta e)]$, $\delta[e] = ()$.

For brevity, let

$$\gamma_s = \mathsf{strip}_{\Phi\Gamma}(\gamma) \qquad \gamma'_s = \mathsf{strip}_{\Gamma}(\gamma') \qquad \rho_s = \mathsf{strip}_{\Phi\Gamma}(\rho) \qquad \rho'_s = \mathsf{strip}_{\Gamma}(\rho')$$

By applying lemma 29 to our assumption, we have

$$() \rhd_\Gamma \gamma_s \lightning \rho_s \to \gamma'_s \lightning \rho'_s \tag{A.7}$$

By applying lemma 28 we further know

$$\gamma_s = \gamma'_s \quad \text{and} \quad \rho_s = \rho'_s \tag{A.8}$$

We wish to show:

$$[\![()]\!]\ (\gamma, d\gamma) \rhd_{\Box A} [\![[(\phi e, \delta e)]]\!]\ \gamma \lightning [\![[e]]\!]\ \rho \to [\![[(\phi e, \delta e)]]\!]\ \gamma' \lightning [\![[e]]\!]\ \rho'$$

Applying lemma 33 and further simplifying, this is equivalent to:

$$() \rhd_{\Box A} ([\![\phi e]\!]\ \gamma_s, [\![\delta e]\!]\ \gamma_s) \lightning [\![e]\!]\ \rho_s \to ([\![\phi e]\!]\ \gamma_s, [\![\delta e]\!]\ \gamma_s) \lightning [\![e]\!]\ \rho_s$$

Applying the definition of the logical relation at $\Box A$, this requires that $[\![e]\!]\ \rho_s = [\![e]\!]\ \rho'_s$ and $[\![\phi e]\!]\ \gamma_s = [\![\phi e]\!]\ \gamma'_s$ and $[\![\delta e]\!]\ \gamma_s = [\![\delta e]\!]\ \gamma'_s$, which hold by equation A.8, and that:

$$[\![\delta e]\!]\ \gamma_s \rhd_A [\![\phi e]\!]\ \gamma_s \lightning [\![e]\!]\ \rho_s \to [\![\phi e]\!]\ \gamma'_s \lightning [\![e]\!]\ \rho'_s$$

which follows from our inductive hypothesis applied to equation A.7.

Case $\dfrac{\Gamma \vdash e : \Box A \qquad \Gamma, x :: A \vdash f : B}{\Gamma \vdash \textbf{\textit{let}}\ [x] = e\ \textbf{\textit{in}}\ f : B}$. Observe that

$$\phi(\textbf{let}\ [x] = e\ \textbf{in}\ f) = \textbf{let}\ [(x, dx)] = \phi e\ \textbf{in}\ \phi f$$
$$\delta(\textbf{let}\ [x] = e\ \textbf{in}\ f) = \textbf{let}\ [(x, dx)] = \phi e\ \textbf{in}\ \delta f$$

Further observe that

$$[\![\textbf{let}\ [(x, dx)] = \phi e\ \textbf{in}\ \phi f]\!]\ \gamma = [\![\phi f]\!]\ (\gamma, x, dx)$$

and likewise for δf in place of ϕf and/or γ' in place of γ. For brevity, let

$$x, dx = [\![\phi e]\!]\ \gamma \qquad x', dx' = [\![\phi e]\!]\ \gamma'$$
$$a = [\![e]\!]\ \rho \qquad a' = [\![e]\!]\ \rho'$$

Using this observation and these abbreviations, we have

$$\begin{aligned}[\![\textbf{let } [x] = e \textbf{ in } f]\!]\ \rho &= [\![f]\!]\ (\rho, a)\\ [\![\textbf{let } [(x, dx)] = \phi e \textbf{ in } \phi f]\!]\ \gamma &= [\![\phi f]\!]\ (\gamma, x, dx)\\ [\![\textbf{let } [(x, dx)] = \phi e \textbf{ in } \delta f]\!]\ \gamma &= [\![\delta f]\!]\ (\gamma, x, dx)\end{aligned}$$

and likewise for γ', ρ'. Then what we wish to show is that

$$[\![\delta f]\!]\ (\gamma, x, dx) \rhd_B [\![\phi f]\!]\ (\gamma, x, dx) \lightning [\![f]\!]\ (\rho, a) \to [\![\phi f]\!]\ (\gamma', x', dx') \lightning [\![f]\!]\ (\rho', a')$$

By our inductive hypothesis for f, it suffices to show that

$$d\gamma \rhd_{\Gamma, x :: A}\ (\gamma, x, dx) \lightning (\rho, a) \to (\gamma', x', dx') \lightning (\rho', a')$$

Since our assumption tells us that $d\gamma \rhd_\Gamma \gamma \lightning \rho \to \gamma' \lightning \rho'$, it suffices to show that $() \rhd_{\Box A}\ (x, dx) \lightning a \to (x', dx') \lightning a'$, which follows directly from our inductive hypothesis for e.

Case $\dfrac{\Gamma \vdash e : \{\underset{EQ}{A}\} \qquad \Gamma, x :: \underset{EQ}{A} \vdash f : L}{\Gamma \vdash \textbf{\textit{for}}\ (x \in e)\ f : L}$. Recall that:

$$\begin{aligned}\phi(\textbf{for } (x \in e)\ f) &= \textbf{for } (x \in \phi e) \textbf{ let } [dx] = [\mathbb{0}\ x] \textbf{ in } \phi f\\ \delta(\textbf{for } (x \in e)\ f) &= (\textbf{for } (x \in \delta e) \textbf{ let } [dx] = [\mathbb{0}\ x] \textbf{ in } \phi f)\\ &\quad \vee (\textbf{for } (x \in \phi e \vee \delta e) \textbf{ let } [dx] = [\mathbb{0}\ x] \textbf{ in } \delta f)\end{aligned}$$

This case of the proof is quite complex; it will help to have a few abbreviations. First, we will be considering various definitons which differ only in whether they use the primed or un-primed versions of γ, ρ, so it will help to refer to these by subscript: $\gamma_1 = \gamma$ and $\gamma_2 = \gamma'$ and $\rho_1 = \rho$ and $\rho_2 = \rho'$.

With this in mind, apply lemma 27 to our inductive hypothesis for e:

$$[\![\delta e]\!]\ (\gamma_1, d\gamma) \rhd_{\{\underset{EQ}{A}\}} [\![\phi e]\!]\ \gamma_1 \lightning [\![e]\!]\ \rho_1 \to [\![\phi e]\!]\ \gamma_2 \lightning [\![e]\!]\ \rho_2 \tag{A.9}$$

$$\iff [\![\phi e]\!]\ \gamma_1 = [\![e]\!]\ \rho_1 \text{ and } [\![\phi e]\!]\ \gamma_2 = [\![e]\!]\ \rho_2 = [\![\phi e]\!]\ \gamma_1 \vee [\![\delta e]\!]\ (\gamma_1, d\gamma) \tag{A.10}$$

Moving to our inductive hypothesis for f, for any $x \in [\![\underset{EQ}{A}]\!]$, let:

$$\rho_i^x = (\rho_i, x \mapsto x) \qquad \gamma_i^x = (\gamma_i, x \mapsto x, dx \mapsto \text{dummy } x) \qquad d\gamma^x = (\gamma_1^x, d\gamma)$$

By our assumption and lemma 26 we have $d\gamma^x \rhd_{\Gamma, x :: \underset{EQ}{A}} \gamma_1^x \lightning \rho_1^x \to \gamma_2^x \lightning \rho_2^x$. From this and our inductive hypothesis for f, applying lemma 27:

$$[\![\delta f]\!]\ d\gamma^x \rhd_L [\![\phi f]\!]\ \gamma_1^x \lightning [\![f]\!]\ \rho_1^x \to [\![\phi f]\!]\ \gamma_2^x \lightning [\![f]\!]\ \rho_2^x \tag{A.11}$$

$$\iff [\![\phi f]\!]\ \gamma_1^x = [\![f]\!]\ \rho_1^x \text{ and } [\![\phi f]\!]\ \gamma_2^x = [\![f]\!]\ \rho_2^x = [\![\phi f]\!]\ \gamma_1^x \vee [\![\delta f]\!]\ d\gamma^x \tag{A.12}$$

We summarize equations A.10 and A.12 as follows, introducing variables s_i, ds, F_i, dF:

$$s_i = [\![e]\!]\ \rho_i = [\![\phi e]\!]\ \gamma_i \qquad F_i(x) = [\![f]\!]\ \rho_i^x = [\![\phi f]\!]\ \gamma_i^x \tag{A.13}$$

$$ds = [\![\delta e]\!]\ (\gamma_1, d\gamma) \qquad dF(x) = [\![\delta f]\!]\ d\gamma^x \tag{A.14}$$

$$s_1 \cup ds = s_2 \qquad F_1(x) \vee dF(x) = F_2(x) \tag{A.15}$$

Now let's give abbreviations to the denotations about which we are trying to prove something:

$$\begin{aligned} l_i &= [\![\textbf{for}\ (x \in e)\ f]\!]\ \rho_i \\ m_i &= [\![\phi(\textbf{for}\ (x \in e)\ f)]\!]\ \gamma_i \\ dm &= [\![\delta(\textbf{for}\ (x \in e)\ f)]\!]\ (\gamma_1, d\gamma) \end{aligned}$$

Then what we wish to show is that $dm \rhd_L m_1 \not\downarrow l_1 \to m_2 \not\downarrow l_2$, or equivalently by applying lemma 27, that $m_1 = l_1$ and $m_2 = l_2 = m_1 \vee dm$. We will do this by showing that

$$l_i = m_i = \bigvee_{x \in s_i} F_i(x) \tag{A.16}$$

$$dm = \Big(\bigvee_{x \in ds} F_1(x)\Big) \vee \Big(\bigvee_{x \in s_1 \cup ds} dF(x)\Big) \tag{A.17}$$

from which our result follows because:

$$\begin{aligned} m_2 &= \bigvee_{x \in s_2} F_2(x) \\ &= \bigvee_{x \in s_1 \cup ds} (F_1(x) \vee dF(x)) && \text{equation A.15} \\ &= \Big(\bigvee_{x \in s_1} F_1(x)\Big) \vee \Big(\bigvee_{x \in ds} F_1(x)\Big) \vee \Big(\bigvee_{x \in s_1 \cup ds} dF(x)\Big) && \text{reassociate} \\ &= m_1 \vee dm \end{aligned}$$

So it suffices to show equations A.16 and A.17. This is mostly a matter of pushing through denotations. For instance, starting with l_i:

$$\begin{aligned} l_i &= [\![\textbf{for}\ (x \in e)\ f]\!]\ \rho_i \\ &= \text{collect}([\![f]\!])(\rho_i, [\![e]\!]\ \rho_i) && \text{definition of } [\![\textbf{for}\ldots]\!] \\ &= \bigvee_{x \in [\![e]\!]\ \rho_i} [\![f]\!]\ (\rho_i, x \mapsto x) && \text{definition of collect} \\ &= \bigvee_{x \in s_i} F_i(x) && \text{equation A.13} \end{aligned}$$

And m_i:

$$\begin{aligned}
m_i &= [\![\textbf{for } (x \in \phi e) \textbf{ let } [dx] = [\mathbb{0}\ x] \textbf{ in } \phi f]\!]\ \gamma_i \\
&= \bigvee_{x \in [\![\phi e]\!]\ \gamma_i} [\![\textbf{let } [dx] = [\mathbb{0}\ x] \textbf{ in } \phi f]\!]\ (\gamma_i, x \mapsto x) \\
&= \bigvee_{x \in s_i} [\![\phi f]\!]\ (\gamma_i, x \mapsto x, dx \mapsto [\![[\mathbb{0}\ x]]\!]\ (\gamma_i, x \mapsto x)) && \text{equation A.13} \\
&= \bigvee_{x \in s_i} [\![\phi f]\!]\ (\gamma_i, x \mapsto x, dx \mapsto [\![\mathbb{0}\ x]\!]\ (\text{strip}\ (\gamma_i, x \mapsto x))) && \text{lemma 33} \\
&= \bigvee_{x \in s_i} [\![\phi f]\!]\ (\gamma_i, x \mapsto x, dx \mapsto \text{dummy}\ x) && \text{pushing definitions} \\
&= \bigvee_{x \in s_i} F_i(x)
\end{aligned}$$

And finally, dm. By expanding in the same manner as directly above we have that:

$$[\![\textbf{for } (x \in \delta e) \textbf{ let } [dx] = [\mathbb{0}\ x] \textbf{ in } \phi f]\!]\ (\gamma_1, d\gamma) = \bigvee_{x \in ds} F_1(x)$$

$$[\![\textbf{for } (x \in \phi e \vee \delta e) \textbf{ let } [dx] = [\mathbb{0}\ x] \textbf{ in } \delta f]\!]\ (\gamma_1, d\gamma) = \bigvee_{x \in s_1 \cup ds} dF(x)$$

And therefore:

$$\begin{aligned}
dm &= [\![\delta(\textbf{for } (x \in e)\ f)]\!]\ (\gamma_1, d\gamma) \\
&= [\![\textbf{for } (x \in \delta e) \textbf{ let } [dx] = [\mathbb{0}\ x] \textbf{ in } \phi f]\!]\ (\gamma_1, d\gamma) \\
&\vee [\![\textbf{for } (x \in \phi e \vee \delta e) \textbf{ let } [dx] = [\mathbb{0}\ x] \textbf{ in } \delta f]\!]\ (\gamma_1, d\gamma) \\
&= \Big(\bigvee_{x \in ds} F_1(x)\Big) \vee \Big(\bigvee_{x \in s_1 \cup ds} dF(x)\Big)
\end{aligned}$$

Which is what we wished to show.

Case $\dfrac{\Gamma \vdash e : \square(\underset{\text{FIX}}{\mathbb{L}} \to \underset{\text{FIX}}{\mathbb{L}})}{\Gamma \vdash \mathit{fix}\ e : \underset{\text{FIX}}{\mathbb{L}}}$. Recall that

$$\phi(\text{fix } e) = \text{semifix } \phi e \qquad \Phi \underset{\text{FIX}}{\mathbb{L}} = \underset{\text{FIX}}{\mathbb{L}} \quad \text{(lemma 19)}$$

$$\delta(\text{fix } e) = \bot \qquad \Delta\Phi \underset{\text{FIX}}{\mathbb{L}} = \underset{\text{FIX}}{\mathbb{L}} \quad \text{(lemmas 19 and 20)}$$

For brevity, let

$$f = [\![e]\!]\ \rho \qquad f' = [\![e]\!]\ \rho'$$

$$(g_1, g_2) = [\![\phi e]\!]\ \gamma \qquad (g_1', g_2') = [\![\phi e]\!]\ \gamma'$$

What we wish to show is:

$$\begin{aligned}
&[\![\bot]\!]\ (\gamma, d\gamma) \rhd_{\underset{\text{FIX}}{\mathbb{L}}} [\![\text{semifix } \phi e]\!]\ \gamma \not\downarrow [\![\text{fix } e]\!]\ \rho \to [\![\text{semifix } \phi e]\!]\ \gamma' \not\downarrow [\![\text{fix } e]\!]\ \rho' \\
\iff &\bot \rhd_{\underset{\text{FIX}}{\mathbb{L}}} \text{semifix}\ ([\![\phi e]\!]\ \gamma) \not\downarrow \text{fix}\ ([\![e]\!]\ \rho) \to \text{semifix}\ ([\![\phi e]\!]\ \gamma') \not\downarrow \text{fix}\ ([\![e]\!]\ \rho') \\
\iff &\bot \rhd_{\underset{\text{FIX}}{\mathbb{L}}} \text{semifix}\ (g_1, g_2) \not\downarrow \text{fix}\ f \to \text{semifix}\ (g_1', g_2') \not\downarrow \text{fix}\ f' \\
\iff &\text{semifix}\ (g_1, g_2) = \text{fix}\ f = \text{semifix}\ (g_1', g_2') = \text{fix}\ f' \qquad \text{(applying lemma 27)}
\end{aligned}$$

By our inductive hypothesis we have $[\![\delta e]\!]\ (\gamma, d\gamma) \rhd_{\Box(\underset{\text{FIX}}{L} \to \underset{\text{FIX}}{L})} (g_1, g_2) \lightning f \to (g_1', g_2') \lightning f'$ and expanding gives us:

$$f = f' \text{ and } g_1 = g_1' \text{ and } g_2 = g_2' \tag{A.18}$$

$$g_2 \rhd_{\underset{\text{FIX}}{L} \to \underset{\text{FIX}}{L}} g_1 \lightning f \to g_1' \lightning f' \tag{A.19}$$

Applying equation A.18 reduces our goal to showing $\text{semifix}\ (g_1, g_2) = \text{fix}\ f$, which holds by lemma 34 applied to equations A.18 and A.19.

Case $\dfrac{\Gamma \vdash e : \Box(A_1 + A_2)}{\Gamma \vdash \mathit{split}\ e : \Box A_1 + \Box A_2}$. Recall that

$$\begin{aligned}
\phi(\text{split } e) = \ & \textbf{let } [z] = \phi e \textbf{ in} \\
& \textbf{case } \text{split } [\pi_1\ z] \textbf{ of} \\
& \quad (\text{in}_i\ Y \to \textbf{let } [x] = Y \textbf{ in} \\
& \qquad \textbf{case } \text{split } [\pi_2\ z] \textbf{ of} \\
& \qquad\quad \text{in}_i\ DY \to \textbf{let } [dx] = DY \textbf{ in } \text{in}_i\ [(x, dx)] \\
& \qquad\quad \text{in}_{i+1 \bmod 2}\ _ \to \text{in}_i\ [(x, \text{dummy } x)])_i \\
\delta(\text{split } e) = \ & \textbf{let } [y] = \phi e \textbf{ in} \\
& \textbf{case } \pi_1\ y \textbf{ of } (\text{in}_i\ _ \to \text{in}_i\ ())_{i \in \{1,2\}}
\end{aligned}$$

By unpacking our inductive hypothesis there must be some $k \in \{1, 2\}$ and some dx, x, a such that $dx \rhd_{A_k} x \lightning a \to x \lightning a$ and:

$$[\![e]\!]\ \rho = [\![e]\!]\ \rho' = \text{in}_k\ a \qquad [\![\phi e]\!]\ \gamma = [\![\phi e]\!]\ \gamma' = (\text{in}_k\ x,\ \text{in}_k\ dx) \qquad [\![\delta e]\!]\ (\gamma, d\gamma) = ()$$

Applying the definition of what we wish to show, it therefore suffices to show that:

$$[\![\text{split } e]\!]\ \rho = [\![\text{split } e]\!]\ \rho' = \text{in}_k\ a \tag{A.20}$$

$$[\![\phi(\text{split } e)]\!]\ \gamma = [\![\phi(\text{split } e)]\!]\ \gamma' = \text{in}_k\ (x,\ dx) \tag{A.21}$$

$$[\![\delta(\text{split } e)]\!]\ (\gamma, d\gamma) = \text{in}_k\ () \tag{A.22}$$

Showing this is a matter of calculation. Equation A.20 is immediate upon noting that $\text{dist}_+^{\Box} = \text{id}$ and therefore $[\![\text{split } e]\!]\ \rho = [\![e]\!]\ \rho$. The other two are no more difficult but considerably more tedious, so we omit the details of the calculations.

□

Bibliography

Umut Acar. *Self-adjusting Computation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, May 2005.

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In John Launchbury and John C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 247–259. ACM, 2002. doi: 10.1145/503272.503296. URL https://doi.org/10.1145/503272.503296.

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

Mario Alvarez-Picallo. *Change actions: From incremental computation to discrete derivatives*. PhD thesis, University of Oxford, 2020. URL https://arxiv.org/abs/2002.05256.

Mario Alvarez-Picallo, Alex Eyers-Taylor, Michael Peyton Jones, and C.-H. Luke Ong. Fixing incremental computation: Derivatives of fixpoints, and the recursive semantics of Datalog. In Luís Caires, editor, *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*, volume 11423 of *Lecture Notes in Computer Science*, pages 525–552. Springer, 2019. ISBN 978-3-030-17183-4. doi: 10.1007/978-3-030-17184-1_19. URL https://doi.org/10.1007/978-3-030-17184-1_19.

Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors, *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*, pages 262–281. Springer, 2010. ISBN 978-3-642-24205-2. doi: 10.1007/978-3-642-24206-9_16. URL http://dx.doi.org/10.1007/978-3-642-24206-9_16.

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR 2011, Fifth Biennial Confer-*

ence on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings, pages 249–260, 2011.

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

Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1371–1382, 2015.

Michael Arntzenius. $\delta(\text{fix f}) = \text{fix } (\delta \text{f } (\text{fix f}))$: or, static differentiation of monotone fixed points. http://www.rntz.net/files/fixderiv.pdf, May 2017. Accessed: 7 June 2018.

Michael Arntzenius and Neel Krishnaswami. Seminaïve evaluation for a higher-order functional language. *Proc. ACM Program. Lang.*, 4(POPL):22:1–22:28, 2020. doi: 10.1145/3371090. URL https://doi.org/10.1145/3371090.

Michael Arntzenius and Neelakantan R. Krishnaswami. Datafun: A functional Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 214–227, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951948. URL http://doi.acm.org/10.1145/2951913.2951948.

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: object-oriented queries on relational data. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 2:1–2:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi: 10.4230/LIPIcs.ECOOP.2016.2. URL https://doi.org/10.4230/LIPIcs.ECOOP.2016.2.

Roland Carl Backhouse and Paul F. Hoogendijk. Elements of a relational theory of datatypes. In Bernhard Möller, Helmuth Partsch, and Stephen A. Schuman, editors, *Formal Program Development - IFIP TC2/WG 2.1 State-of-the-Art Report*, volume 755 of *Lecture Notes in Computer Science*, pages 7–42. Springer, 1993. doi: 10.1007/3-540-57499-9_15. URL https://doi.org/10.1007/3-540-57499-9_15.

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

Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In Moshe Y. Vardi, editor, *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 23-25, 1987, San Diego, California, USA*, pages 269–284. ACM, 1987. doi: 10.1145/28659.28689. URL https://doi.org/10.1145/28659.28689.

Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In Shail Arora and Gary T. Leavens, editors, *Proceedings of the 24th*

Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA, pages 243–262. ACM, 2009. doi: 10.1145/1640089.1640108. URL https://doi.org/10.1145/1640089.1640108.

Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. 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, pages 145–155, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594304. URL http://doi.acm.org/10.1145/2594291.2594304.

Angelos Charalambidis, Konstantinos Handjopoulos, Panagiotis Rondogiannis, and William W. Wadge. Extensional higher-order logic programming. *ACM Trans. Comput. Log.*, 14(3): 21:1–21:40, 2013. doi: 10.1145/2499937.2499942. URL https://doi.org/10.1145/2499937.2499942.

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

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

Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In Michael J. Carey and Steven Hand, editors, *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, page 1. ACM, 2012. doi: 10.1145/2391229.2391230. URL https://doi.org/10.1145/2391229.2391230.

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

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. MIT Press, 2005. ISBN 978-0-262-56214-0.

Paolo G. Giarrusso. *Optimizing and Incrementalizing Higher-order Collection Queries by AST Transformation*. PhD thesis, University of Tübingen, Germany, 2020. URL https://publikationen.uni-tuebingen.de/xmlui/handle/10900/97998/.

Paolo G. Giarrusso, Yann Régis-Gianas, and Philipp Schuster. Incremental λ-calculus in cache-transfer style: Static memoization by program transformation. In *ESOP*, volume 11423 of *Lecture Notes in Computer Science*, pages 553–580. Springer, 2019.

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

Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. Adapton: composable, demand-driven incremental computation. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 156–166. ACM, 2014. doi: 10.1145/2594291.2594324. URL https://doi.org/10.1145/2594291.2594324.

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

Christoph Koch. Incremental query evaluation in a ring of databases. In Jan Paredaens and Dirk Van Gucht, editors, *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 87–98. ACM, 2010. doi: 10.1145/1807085.1807100. URL https://doi.org/10.1145/1807085.1807100.

Christoph Koch. Incremental query evaluation in a ring of databases. 2013. URL http://infoscience.epfl.ch/record/183766. Revised version of PODS 2010 paper.

Vassilis Kountouriotis, Panos Rondogiannis, and William W Wadge. Extensional higher-order Datalog. In *Short Paper Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 1–5. Citeseer, 2005.

Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, 2009. doi: 10.1145/1592761.1592785. URL https://doi.org/10.1145/1592761.1592785.

Magnus Madsen and Ondrej Lhoták. Safe and sound program analysis with Flix. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 38–48. ACM, 2018. doi: 10.1145/3213846.3213847. URL https://doi.org/10.1145/3213846.3213847.

Magnus Madsen and Ondrej Lhoták. Fixpoints for the masses: programming with first-class Datalog constraints. *Proc. ACM Program. Lang.*, 4(OOPSLA):125:1–125:28, 2020. doi: 10.1145/3428193. URL https://doi.org/10.1145/3428193.

Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. From Datalog to Flix: A declarative language for fixed points on lattices. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 194–208. ACM, 2016. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908096. URL http://doi.acm.org/10.1145/2908080.2908096.

Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013. URL http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf.

Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. Build systems à la carte: Theory and practice. *Journal of Functional Programming*, 30:e11, 2020. doi: 10.1017/S0956796820000088.

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

Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, 1982. doi: 10.1145/357172.357177. URL https://doi.org/10.1145/357172.357177.

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

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

Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1&2):3–36, 1995.

Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors, *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*, pages 245–251. Springer, 2010. doi: 10.1007/978-3-642-24206-9_14. URL https://doi.org/10.1007/978-3-642-24206-9_14.

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

Terrance Swift and David Scott Warren. XSB: extending prolog with tabled logic programming. *Theory Pract. Log. Program.*, 12(1-2):157–187, 2012. doi: 10.1017/S1471068411000500. URL https://doi.org/10.1017/S1471068411000500.

Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.*, 2(OOPSLA):139:1–139:29, 2018. doi: 10.1145/3276509. URL https://doi.org/10.1145/3276509.

K. Tuncay Tekle and Yanhong A. Liu. More efficient Datalog queries: subsumptive tabling beats magic sets. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 661–672. ACM, 2011. doi: 10.1145/1989323.1989393. URL https://doi.org/10.1145/1989323.1989393.

William W. Wadge. Higher-order Horn logic programming. In Vijay A. Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991*, pages 289–303. MIT Press, 1991.

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

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

John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005. doi: 10.1007/11575467_8. URL https://doi.org/10.1007/11575467_8.

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