In a functional language, types are propositions, terms are proofs, embody the Curry-Howard correspondence in two different ways. A particular formula is derivable or not. Proof search algorithm on these clauses to figure out whether a

Execution of a Prolog program can be understood as running a

∀

ideas to steal.

language designers, we can look to logic languages to discover new

for sorrow, but we prefer to view it as an opportunity: as functional

Kanren (Friedman et al. 2005)). This could be seen as an occasion

Mercury (Somogyi et al. 1994), Curry (Antoy and Hanus 2010) and

declarative languages. Despite this shared epithet, the logic and

functionality, and declarativeness. However, most use-cases require extending

Datalog in an application-specific manner. In this paper we define

Datafun, an analogue of Datalog supporting higher-order functional

programming. The key idea is to track monotonicity via types.

1. Introduction

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

A Prolog program can be understood as a collection of logical axioms formulated as Horn clauses (i.e., first-order formulas of the form ∀x. P₁ ∧ ... ∧ Pₙ → Q, where Pᵢ and Q are atomic formulas). Execution of a Prolog program can be understood as running a proof search algorithm on these clauses to figure out whether a particular formula is derivable or not.

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

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

This view of Prolog suggests considering other possible design choices, such as restricting the logical language so as to make proof search decidable. One of the oldest such variants is Datalog [Gal- laire and Minker 1978], a syntactic subset of Prolog satisfying three restrictions:

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

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

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

In all of these cases, the use of Datalog permits giving specifications and implementations which are dramatically shorter and clearer than alternatives implemented in more conventional languages. However, while all of these applications are built on a foundation of Datalog, they all also extend it significantly. For example, it is impossible even to implement arithmetic in Datalog, since adding 2 and 3 produces 5, which is a new term not equal to either

Datafun: a Functional Datalog (PREPRINT)

[Copyright notice will appear here once 'preprint' option is removed]
2 or 3! As a result, even though Datalog has a very clean semantics, its metatheory needs to be re-established once again for each application-specific extension to it.

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

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

Contributions

- We describe Datafun, a typed language capturing the expressive power of Datalog and extending it to support higher-order functional programming. Datafun’s key feature is to track monotonicity with types. This permits us to use typing to analyze fixed point computations in a way ensuring their termination.

- We present examples illustrating the expressive power of Datafun, including relational-algebra-style operations, transitive closure, CYK parsing, and dataflow analysis. Some of these examples are familiar from Datalog, but many of them go well beyond what can be expressed in it, illustrating the benefits of our approach.

- We identify the semantic structures underpinning Datalog, and use this to give a denotational semantics for Datafun in terms of a pair of adjunctions between Set, Poset, SemiLat.

- We have a prototype implementation of Datafun in Racket, which has been used to implement all of the examples in this paper, and is available at [link omitted for double-blind review].

2. Datafun, informally

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

1. We add a type of finite sets, \( \{A\} \).

2. We add a type of monotone functions, \( A \downarrow \rightarrow B \). Consequently Datafun has two flavors of variable, ordinary and monotone. We write ordinary variables in script and monotone variables in bold.

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

- Booleans 2 are ordered false < true.
- Natural numbers \( \mathbb{N} \) have the usual order: \( 0 < 1 < 2 < ... \).
- We have no particular use-case for comparing strings \( \mathcal{str} \) in this paper, so we order them discretely: \( a \leq b \) iff \( a = b \).

- Pairs and functions are ordered pointwise:
  - \( (a, x) \leq (b, y) \) iff \( a \leq b \land x \leq y \)
  - \( f \leq g \) iff \( \forall x. f(x) \leq g(x) \)

- Sum types are ordered disjointly: \( \text{in}_1 \alpha \leq \text{in}_1 \beta \) iff \( \alpha \leq \beta \), but \( \text{in}_1 \alpha \) and \( \text{in}_1 \beta \) are never comparable.

- Sets are ordered by inclusion: \( a \leq b \) iff \( a \subseteq b \).

3. We add a term \( \text{fix} \ x \in e \) denoting the least fixed point of the monotone function \( \langle \lambda x. e \rangle \). This is computed (modulo optimizations) by iteration, starting from the smallest value of the desired type and halting once a fixed point is found. This strategy constrains the types of fix terms in several ways:

- The type must have a smallest value. We enforce this using semilattice types (see item 4 below).

- The type must support equality tests, to determine when a fixed point has been reached. We call a type supporting equality tests an eqtype.

- To ensure termination, the type must have finite height. We conservatively approximate this property by limiting fix to finite types.

In summary, fix may only be used at finite semilattice eqtypes.

4. Generalizing the empty set \( \emptyset \) and union \( \cup \), we identify a subset of types that have a least element \( \epsilon \) and a least upper bound operator \( \lor \). We call these semilattice types and denote them by the metavariables \( \Lambda, M \).

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

Second, they provide a natural eliminator for sets. Given \( e : \{A\} \), we write \( \forall (x \in e) \in \epsilon_\Lambda \) for the least upper bound, over all elements \( x \in e \), of \( \epsilon_\Lambda \), for some semilattice type \( \Lambda \). If \( \epsilon_\Lambda \) is a set, for example, this provides the set type’s monadic “bind” operation. For example, \( \forall (x \in \{1, 2, 3\}) \{10 \cdot x, x^2\} \) denotes the set \( \{1, 4, 9, 10, 20, 30\} \).

3. Examples

For purposes of these examples, we use a simple Haskell-like syntax for top-level type and function definitions. We also permit ourselves infix notation, let-binding, n-ary tuples, n-ary sum types with named constructors, pattern-matching (including non-linear patterns), and additional syntax sugar given in Figure 2. All of

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

2. Technically, the partial orderings on these types form join-semilattices with a least element. For brevity’s sake, we call these structures simply “semilattices.”
3.1 Filtering, mapping, and cross products

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

- **map**: \( \{ A \to B \} \to \{ A \to \{ B \} \} \)
  
- **filter**: \( \{ A \to 2 \} \to \{ A \to \{ A \} \} \)
  
- **cross product**: \( \{ x : \{ A \times B \} \to \{ A \times B \} \} \)

These conveniences are supported (with slightly different concrete syntax) in our implementation.

For clarity, we set the names of top-level variables in sans-serif; ordinary variables in script or italic (for long variable names); and monotone variables in **bold**.

Although Datafun as presented does not have polymorphism, we give our examples their most general possible type schemes.

### 3.2 Membership, intersection

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

\[
\{ e \in A \to 2 \} (x \in A) x = y
\]

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

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

\[
\{ \land : \{ A \to 2 \} \to \{ A \to \{ A \} \} \}
\]

\[
A \cap B = \{ x \in A, y \in B, x = y \}
\]

However, explicitly testing for equality can become tedious, so we usually use **nonlinear pattern-matching** instead — that is, we bind the same-named variable multiple times, which indicates it must have an equal value at each occurrence:

\[
\{ \land : \{ A \to 2 \} \to \{ A \to \{ A \} \} \}
\]

\[
A \cap B = \{ x \in A, x \in B \}
\]

This is merely syntax sugar for an equality test, so the condition that the set’s element type support equality remains in force.

### 3.3 Composition of relations

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

\[
\{ \circ : \{ A \times B \} \to \{ B \times C \} \to \{ A \times C \} \}
\]

\[
R \circ S = \{ (a, b) | (a, c) \in R, (b, c) \in S \}
\]

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

\[
r(X, Y) :- (\ldots).
\]

\[
s(X, Y) :- (\ldots).
\]

\[
rs(A, C) :- r(A, B), s(B, C).
\]

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

### 3.4 Transitive closure

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

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

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

\[
data person = \text{EÄRENDEL} | \text{ELROND} | \text{ARWEN}
\]

\[
\text{parent, ancestor : } \{ \text{person} \times \text{person} \}
\]

\[
\text{parent} = \{ (\text{EÄRENDEL}, \text{ELROND}), (\text{ELROND}, \text{ARWEN}) \}
\]

\[
\text{ancestor} = \text{fix X is person } \lor (X \circ X)
\]
The type `person` represents the domain of our parent and ancestor relations. `parent` is simply a list of parent-child pairs. `ancestor` is where the action is at: since the Datalog predicate `ancestor` is defined recursively, `ancestor` is defined as a least fixed point — in this case, of the following equation:

\[ X = \text{parent} \lor (X \cdot X) \]

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

### 3.4.1 Transitive closure with an upper bound

The above explanation glosses over one critical requirement: `fix` requires that the type at which the fixed-point is taken be a finite semilattice `eqtype`.

The type of `ancestor` is `(person × person)`. Does this suffice? It’s certainly a semilattice, since it’s a set type. Since `person` is effectively a sum of units, it supports equality, and sets and products of `eqtypes` are themselves `eqtypes`. Likewise, `person` is finite, and products and sets of finite types are themselves finite.

So! We find ourselves in the clear, for now. However, in practice, the restriction of `fix` to finite types can be quite limiting. So Datafun provides a more general way to take a fixed-point: provide an upper bound which the desired fixed point will not exceed. For this we write `(fix x ≤ e)` where `e` is our upper bound.

Suppose, for example, we wish to represent our `drumatis personae` as strings `str` rather than defining a person type. Then, making use of bounded fixed points, we could write:

```plaintext
person : [str]
person = \{"eärendil", "elrond", "arwen"\}
parent, ancestor : (str × str)
parent = \{(eärendil, elrond), (elrond, arwen)\}
ancestor = fix X ≤ (person × person) is parent \lor (X \cdot X)
```

Instead of a person type, we have person `set`, which we use to construct an upper bound on our fixed-point: `(person × person)`, the complete binary relation. Since every string in `parent` is also in person, the transitive closure of `parent` cannot exceed this upper bound.

However, this invariant is left to the programmer to check. What if a sloppy programmer should mistakenly include a person in `parent` not present in `person`? More generally, what if the “fixed point” `(fix x ≤ e)` is trying to compute exceeds `e`? (Or indeed, no such fixed point exists?)

In that case, the value of `(fix x ≤ e)` is clamped to the upper bound `e`. This ensures Datafun programs terminate even in the presence of sloppy programmers, and although they may not have the value you expect, that value is at least predictable.

### 3.4.2 Generic transitive closure

Thus far we have only considered taking the transitive closure of a relation we have already defined. But consider: for any finite `eqtype` `A`, we may write:

\[
\text{trans} : (A \times A) \rightarrow (A \times A) \\
\text{trans} E = \text{fix} X \leq (A \cdot A)
\]

Similarly, for any `eqtype` `A`, we may write:

\[
\text{trans} : A \rightarrow (A \times A) \rightarrow (A \times A) \\
\text{trans} V E = \text{fix} S \leq (V \cdot V) \text{ is } E \lor (S \cdot S)
\]

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

### 3.5 CYK parsing

Parsing can be understood logically, with a parse tree representing a proof that a certain string belongs to a language described by a context-free grammar. As a result, it is possible to formulate parsing in terms of proof search (Sieber et al. 1995). One of the simplest algorithms for parsing context-free grammars is the Cocke-Younger-Kasami (CYK) algorithm for parsing with grammars in Chomsky normal form.

Given a grammar `G`, we begin by introducing a family of predicates (sometimes called facts or items) `A(i, j)`, with one `A` for each nonterminal, and `i` and `j` representing indices into a string. Given a word `w`, we write `w[i, n]` for the `n`-element substring of `w` beginning at position `i`. Then, we can specify the CYK algorithm with the following two inference rules:

\[
\begin{align*}
B(i, j) & \quad C(j, k) \quad (A \rightarrow B C) \in G \\
A(i, k) & \quad \text{if } A(i, i + n) \in \text{G}
\end{align*}
\]

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

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

```plaintext
type sym = str
data rule = STRING str | CONCAT sym
type grammar = [sym × rule]
type fact = sym × N × N
```

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

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

```plaintext
iter : str → grammar → {fact} → {fact}
iter text G chart =
\{
\langle a, i, k \rangle | \langle a, CONCAT b c \rangle \in G, \\
\langle b, j, k \rangle \in \text{chart}, \langle c, j, k \rangle \in \text{chart}\}
\lor \{(a, i, i + \text{length } s) | \langle a, \text{STRING } s \rangle \in G, i + \text{length } s \in \text{chart}\}
```

This function works by taking a string `text` and a grammar `G`, and then taking a set of facts `chart`, and taking a union. The first clause is a set comprehension, saying that we return \( \langle a, i, k \rangle \) if \( \langle b, i, j \rangle \).

\footnote{In Chomsky normal form, each production is of the form \( A \rightarrow B \cdot C \) or \( A \rightarrow \alpha \), with \( A, B, C \) ranging over nonterminals, and \( \alpha \) over nonempty strings of terminals.}
and \((c, j, k)\) are in chart – this corresponds to applications of the first rule. The second clause corresponds to the second rule above, saying that \((a, i, i + \text{length } s)\) is a generated fact if \(s\) is a substring of text at position \(i\).

We can then use \(\text{iter}\) to implement the parse function.

\[
\begin{align*}
\text{parse} &: \text{str} \rightarrow \text{grammar} \xrightarrow{+} \{\text{sym}\} \\
\text{parse text} \ G &= \\
\text{let } n = \text{length text} \\
\text{bound} &= \{(a, i, j) \mid (a, \_j) \in G, \ i \in \text{range } 0 \ n, \ \_j \in \text{range } i \ n\} \\
\text{chart} &= \text{fix } C \leq \text{bound} \ \text{is \ iter \ text \ G} \ C \\
\text{in } \{a \mid (a, 0, n) \in \text{chart}\}
\end{align*}
\]

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

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

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

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

### 3.6 Dataflow analysis

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

\[
\begin{align*}
\text{type var} &= \text{str} \\
\text{type label} &= \mathbb{N} \\
\text{data oper} &= \text{EQ} \mid \text{LE} \mid \text{ADD} \mid \text{SUB} \mid \text{MUL} \mid \text{DIV} \\
\text{data atom} &= \text{VAR \ var} \mid \text{NUM \ N} \\
\text{data expr} &= \text{ATOM \ atom} \mid \text{APPLY \ oper \ atom} \\
\text{data stmt} &= \text{ASSIGN \ var \ expr} \mid \text{IF \ expr \ label} \\
\text{type program} &= \{\text{label} \times \text{stmt}\}
\end{align*}
\]

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

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

\[
\begin{align*}
\text{labels} &: \text{program} \rightarrow \{\text{label}\} \\
\text{vars} &: \text{program} \rightarrow \{\text{var}\} \\
\text{uses} &: \text{stmt} \rightarrow \{\text{var}\} \\
\text{defines} &: \text{stmt} \rightarrow \{\text{var}\}
\end{align*}
\]

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

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

\[
\begin{align*}
\text{type flow} &= \{\text{label} \times \text{label}\} \\
\text{flow} &: \text{program} \rightarrow \text{flow} \\
\text{flow} c &= \bigvee\{(i, s) \in c\} \\
\text{case } s \text{ of } \text{IF}_{j} k &\rightarrow \{(i, j), (i, k)\} \\
&\rightarrow \{[i], i \in \text{stmt}\} \{i, i + 1\} \{i + 1 \in \text{labels } c\}
\end{align*}
\]

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

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

\[
\begin{align*}
\text{live} &: \text{program} \rightarrow \text{flow} \rightarrow \{\text{label} \times \text{var}\} \\
\text{live code flow} &= \\
\text{fix } \text{Live} \leq \text{labels code} \times \text{vars code} \text{ is} \\
\bigvee\{(i, \text{stmt}) \in \text{code}\} \\
&\bigvee\{(i, v) \mid v \in \text{uses stmt}\} \\
&\bigvee\{(i, v) \mid (i, j) \in \text{flow}, \ (j, v) \in \text{Live}, \ (v \in \text{? defines stmt})\}
\end{align*}
\]

For a statement \(\text{stmt}\) at label \(i\), we say that the variable \(v\) is live at \(i\) if \(v\) is used by \(\text{stmt}\). The variable \(v\) is also live at \(i\) if control flows from \(i\) to \(j\), and \(v\) is live at \(j\), assuming that \(\text{stmt}\) isn’t a definition site for \(v\).

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

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

\[
\begin{align*}
\text{reachingDefinitions} &: \text{program} \rightarrow \text{flow} \rightarrow \{\text{goals code} \times \text{labels code}\} \\
\text{reachingDefinitions code flow} &= \\
\text{fix } \text{RD} \leq \{\text{goals code} \times \text{labels code}\} \times \text{labels code} \text{ is} \\
\bigvee\{(i, \text{stmt}) \in \text{code}\} \\
&\bigvee\{(i, v) \mid v \in \text{defines stmt}\} \\
&\bigvee\{(i, v) \mid (i, j) \in \text{flow}, \ (v \in \text{? defines stmt})\}
\end{align*}
\]

We define a function reachingDefinitions which takes a program and a set of flows as arguments, and returns a relation of type \(\{\text{goals code} \times \text{labels code}\}\). An entry \(\{(i, v), i\}\) in this relation means the definition of \(v\) at \(i\) reaches program point \(i\).
This is then computed as a fixed point of two clauses. First, if there is a definition \( v \) at program point \( i \), then \( i \) is reached by that definition. Second, if \( \{i, v\} \) reaches \( j \), and \( j \) flows to \( i \), then \( \{i, v\} \) reaches \( i \) as long as \( v \) is not re-defined at \( i \).

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

4. Typing rules

Datafun’s typing judgment \( \Delta; \Gamma \vdash e : A \) is defined by the inference rules given in Figure 4. We gloss \( \Delta; \Gamma \vdash e : A \) as follows: “expression \( e \) has type \( A \) using variables from \( \Delta \cup \Gamma \), and moreover the value of \( e \) is monotone with respect to the variables in \( \Gamma \).”

The context \( \Delta \) types ordinary variables; \( \Gamma \), monotone variables. Both admit the usual structural rules of exchange, weakening, and contraction. Variables from either context may be used freely (rules \( \text{VAR}, \text{VAR}' \)).

4.1 Functions and application

Two function types require two function introduction rules: the ordinary \( \lambda \) and the monotone \( \lambda^+ \). These simply introduce variables into their respective contexts. Monotone function application \( \text{APP}^+ \) is perfectly standard, but ordinary function application \( \text{APP} \) has a peculiarity: the argument \( e_2 \) gets an empty monotone context.

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

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

4.2 Products and sums

The pairing and projection rules, \( \text{PAIR} \) and \( \pi_i \), are completely standard, as is the \( \text{IN} \) rule for sum introduction. Sum elimination, however, splits into two rules, \( \text{CASE}^+ \) and \( \text{CASE}^\emptyset \). \( \text{CASE}^+ \) requires its branches to be monotone in the variable \( x \) it introduces, and consequently its subject \( e \) is permitted access to the monotone context \( \Gamma \). \( \text{CASE} \), however, analyses its subject \( e \) as a constant — wiping clean its monotone context — and thus is allowed to introduce the variable \( x \) into the ordinary context \( \Delta \).

4.3 Booleans

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

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

Second, \( \text{IF}' \), where the subject \( e \) has full access to \( \Gamma \), but the if-expression must have \( \text{semilattice type} \), and the else-branch is constrained to be \( e \) — the least value, thus smaller than \( e_1 \).

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

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

4.4 Semilattices and sets

The semilattice \( \epsilon \) and \( \vee \) operations are typed by their correspondingly-named rules. As \( \vee \) is monotone, its arguments have full access to the monotone context \( \Gamma \).

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

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

4.5 Fixed points

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

\( \text{FIX}_\emptyset \) loosens this restriction by letting us take fixed points at (not-necessarily-finite) semilattice equatypes \( \text{L}_\infty \) as long as we provide an upper bound \( e_1 : \text{L}_\infty \) which we can check we do not exceed.

5. Semantics and metatheory

We give a denotational semantics for Datafun in terms of three categories (Set, Poset, and SemiLat) and two adjunctions between them (see Figure 2). We present the notation we use in Figure 2 we take care to distinguish between sets and posets, and since posets are more central to our semantics, most of our notation concerns them. We take less care to distinguish between sets and posets.

5.1 The category SemiLat

SemiLat is the category of join-semilattices with least elements, which we call simply “semilattices”. Directly, a semilattice is a poset \( L \), with a least element \( \epsilon \), in which any two elements \( a, b \) have a least-upper-bound \( a \lor b \). From \( \epsilon \) and \( \lor \) it follows that any finite subset \( X \subseteq_{\text{fin}} L \) has a least upper bound, written \( \bigvee X \).

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

\[
\begin{align*}
\text{f}(a \lor_B b) &= f(a) \lor f(b) \\
\text{f}(\epsilon_A) &= \epsilon_B
\end{align*}
\]
\[
\begin{align*}
\frac{x:A \in \Delta \quad \text{VAR}}{\Delta; \Gamma \vdash x: A} & & \frac{x:A \in \Gamma \quad \text{VAR}^+}{\Delta; \Gamma \vdash x: A} & & \frac{\Delta, x:A; \Gamma \vdash e:B}{\Delta; \Gamma \vdash \lambda x.e : A \rightarrow B} & & \frac{\Delta; \Gamma \vdash e_1:A \rightarrow B \quad \Delta; \Gamma \vdash e_2:A}{\Delta; \Gamma \vdash e_1 \, e_2 : A} \\
\frac{\Delta; \Gamma \vdash e : e_1 : A \quad \Delta; \Gamma \vdash e : B}{\Delta; \Gamma \vdash e : A \rightarrow B} & & \frac{\Delta; \Gamma \vdash e_1 : A \quad \Delta; \Gamma \vdash e_2 : A}{\Delta; \Gamma \vdash e_1 \, e_2 : A} & & \frac{\Delta; \Gamma \vdash \Gamma_1 \rightarrow \Gamma_2}{\Delta; \Gamma \vdash \pi_1.e : A_1} & & \frac{\Delta; \Gamma \vdash e : A_1 \times A_2}{\Delta; \Gamma \vdash \pi_1.e : A_1} \\
\frac{\Delta; \Gamma \vdash e : A_1}{\Delta; \Gamma \vdash \text{in}_1.e : A_1 \rightarrow A_2} & & \frac{\Delta; \Gamma \vdash \text{case} e : \text{in}1.x \rightarrow e_1; \text{in}2.x \rightarrow e_2 : C}{\Delta; \Gamma \vdash \text{case} e \, \text{of} \, \text{in}1.x \rightarrow e_1; \text{in}2.x \rightarrow e_2 : C} & & \frac{\Delta; \Gamma \vdash e : A_1 \quad \Delta; \Gamma \vdash e : A_2}{\Delta; \Gamma \vdash e : A_1 \times A_2} & & \frac{\Delta; \Gamma \vdash e : \text{in}1.e : e_1; \text{in}2.e : e_2}{\Delta; \Gamma \vdash \text{in}1.e \, \text{in}2.e : e_1 \times e_2} \\
\frac{\Delta; \Gamma \vdash \text{true} : 2}{\Delta; \Gamma \vdash \text{false} : 2} & & \frac{\Delta; \Gamma \vdash e : A \quad \Delta; \Gamma \vdash e_1 : A \rightarrow B \quad \Delta; \Gamma \vdash e_2 : B}{\Delta; \Gamma \vdash \text{if} \, e \, \text{then} \, e_1 \, \text{else} \, e_2 : B} & & \frac{\Delta; \Gamma \vdash e : A \quad \Delta; \Gamma \vdash e : B}{\Delta; \Gamma \vdash \text{if} \, e \, \text{then} \, e_1 \, \text{else} \, e_2 : B} & & \frac{\Delta; \Gamma \vdash e : E}{\Delta; \Gamma \vdash \text{fix} \, x \, \text{is} \, e : L} & & \frac{\Delta; \Gamma \vdash \text{fix} \, x \, \leq \, e_1 \, \text{is} \, e_2 : L}{\Delta; \Gamma \vdash \text{fix} \, x : L} \\
\frac{\Delta; \Gamma \vdash e : L}{\Delta; \Gamma \vdash e_1 \, \lor \, e_2 : L} & & \frac{\Delta; \Gamma \vdash e : A \\ \Delta; \Gamma \vdash \{e\} : A}{\Delta; \Gamma \vdash e : A} & & \frac{\Delta; \Gamma \vdash \text{fix} \, x : L}{\Delta; \Gamma \vdash \text{fix} \, x \, \leq \, e_1 \, \text{is} \, e_2 : L} & & \frac{\Delta; \Gamma \vdash \text{fix} \, x : L}{\Delta; \Gamma \vdash \text{fix} \, x \, \leq \, e_1 \, \text{is} \, e_2 : L} & & \frac{\Delta; \Gamma \vdash \text{fix} \, x : L}{\Delta; \Gamma \vdash \text{fix} \, x \, \leq \, e_1 \, \text{is} \, e_2 : L} \\
\end{align*}
\]

**Figure 4.** Typing rules for core Datafun

\[
\begin{align*}
\text{Set} & \xrightarrow{\perp} \text{Poset} & \text{Poset} & \xrightarrow{\cup} \text{SemiLat} \\
\text{FS} & \xrightarrow{\downarrow} \text{Disc} & \xrightarrow{\mathcal{F}} \text{Disc} & \xrightarrow{\mathcal{F}} \text{SemiLat} \\
\end{align*}
\]

**Figure 5.** Semantic categories of Datafun

| P | Underlying set of the poset P |
| S | Set of strings |
| 1 | One-element poset \(\{\}\) |
| 2 | Two-element poset \([f, f]\), with \(ff < tt\) |
| \(\mathbb{N} \leq\) | The naturals \(\mathbb{N}\), as a (totally ordered) poset |
| \(P + Q\) | Disjointly-ordered poset on disjoint union of \(P, Q\) |
| \(P \times Q\) | Pointwise poset on pairs of \(P_s, Q_s\) |
| \(P \rightarrow Q\) | Pointwise poset on monotone maps \(\text{Poset}(P, Q)\) |
| \(\mathcal{F} P\) | Free semilattice on a poset \(P\) |
| \(\mathcal{U} L\) | Underlying poset of a semilattice \(L\) |
| \(\text{Disc} A\) | Discrete poset underlying \(A\) |
| \(\text{FS} A\) | Free semilattice on a set \(A\); same as \(\mathcal{F} (\text{Disc} A)\) |
| \(\downarrow(x : P)\) | The sub-poset of \(P\) below \(x\): \([y \in P \mid y \leq x]\) |

**Figure 6.** Semantic notation

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

5.2 Denotation of Datafun types

Datafun types and contexts denote posets as shown in Figure 7. To complete our semantics, we will need a few simple lemmas about the denotations of Datafun types. First, we need to know that our semilattice types are semilattices, and that any function types are finite:

\[
[A] \in \text{Poset}_0 \\
[2] = 2 \\
[N] = \mathbb{N} \leq \\
[str] = \text{Disc } S \\
[A \times B] = [A] \times [B] \\
[A + B] = [A] + [B] \\
[A \rightarrow B] = \text{Disc } [A] \rightarrow [B] \\
[[A]] = \mathcal{FS} [[A]]
\]

\[
[A], [\Gamma] \in \text{Poset}_0 \\
[[\Gamma]] = 1
\]

\[
[[A, x:A]] = [[A] \times [A]] \\
[[\Gamma, x:A]] = [[\Gamma] \times [A]]
\]

**Figure 7.** Denotations of Datafun types and contexts

**Lemma 1.** The denotation \([L]\) of a semilattice type \(L\) is a semilattice.

**Lemma 2.** The poset \([A]\) denoted by a finite ectype \(A\) is finite.

Second, to show that bounded fixed-points \((\text{fix } x \leq e \; \forall \; e\) is \(e\)) terminate, we need possible \(e\) \(\forall \; e\) to pick out a finite-height sub-poset:

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

All of these are trivial to prove by induction over types.

5.3 Denotation of Datafun terms

In Figure 9 we give a denotation for typing derivations with the following signature:

\[
[[A], \Gamma \vdash e : A] \in \text{Set}([[A]], \text{Poset}([[\Gamma]], [[A]]))
\]

Colloquially, \(\Delta; \Gamma \vdash e : A\) denotes a function \(A \rightarrow A\) to \(A\) that must be monotone in \(\Gamma\) (but not in \(\Delta\)).

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

\[
\frac{\Delta; \Gamma \vdash e : A}{\Delta; \Gamma \vdash \text{fix} \, x : L}
\]

\[
\frac{\Delta; \Gamma \vdash e \leq e_1 \text{ is } e_2 : L}{\Delta; \Gamma \vdash \text{fix} \, x \leq e_1 \text{ is } e_2 : L}
\]
Derivation

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

\[ \begin{align*}
\delta \gamma &= \pi_1 \delta \\
\delta \gamma &= \pi_1 \gamma \\
\delta \gamma &= x \mapsto [e] \langle \delta, x \rangle \\
\delta \gamma &= \pi_1 (\|e\|) \\
\delta \gamma &= \text{in}_1 (\|e\|) \\
\delta \gamma &= \begin{cases} \{e_1\} \langle \delta, x \rangle & \text{if } \|e\| = \text{in}_1 x \\
\{e_2\} \langle \delta, x \rangle & \text{if } \|e\| = \text{in}_2 x \\
\end{cases} \\
\delta \gamma &= \begin{cases} \{e_1\} \{e_2\} \langle \delta, \gamma \rangle & \text{if } \|e\| = \text{in}_1 x \\
\{e_2\} \{e_2\} \langle \delta, \gamma \rangle & \text{if } \|e\| = \text{in}_2 x \\
\end{cases} \\
\delta \gamma &= \text{tt} \\
\delta \gamma &= \text{ff} \\
\delta \gamma &= \begin{cases} \{e_1\} \langle \delta, \gamma \rangle & \text{if } \|e\| = \text{tt} \\
\{e_2\} \langle \delta, \gamma \rangle & \text{if } \|e\| = \text{ff} \\
\end{cases} \\
\delta \gamma &= \epsilon_{\lambda L} \\
\delta \gamma &= \{e_1\} \langle \delta, \gamma \rangle \\
\delta \gamma &= \langle [e_1], \delta \rangle \\
\delta \gamma &= \{e_2\} \{e_2\} \langle \delta, \gamma \rangle \\
\delta \gamma &= \text{ifp}(x \mapsto [e] \langle \delta, x \rangle) \in \text{L}_{\lambda} \\
\delta \gamma &= \text{ifp}(\lambda x \mapsto \text{if } s \leq [e_1] \delta \gamma \text{ otherwise } [e_2] \delta \gamma) \in \text{L}_{\lambda} \\
\end{align*} \]

Figure 8. Denotations of Datafun typing derivations
Lemma 4 (Fixed points in finite-height pointed posets). Any monotone map \( f : P \to P \) on a poset \( P \) of finite height with a least element \( \varepsilon \) has a least fixed point of the form \( f^n(\varepsilon) \).

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

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

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

5.5 Discussion

Theorem 3

- Substitution, monotone
- Substitution, ordinary

Theorem 2

- Substitution, ordinary

Theorem 1 (Weakening and exchange). The rules

\[
\frac{\Delta; \Gamma \vdash e : A}{\Delta, \Delta' ; \Gamma, \Gamma' \vdash e : A} \quad \text{WEAK} \quad \frac{\Delta_2, \Delta_1 ; \Gamma_2, \Gamma_1 \vdash e : A}{\Delta_1, \Delta_2 ; \Gamma_2, \Gamma_1 \vdash e : A} \quad \text{XCHG}
\]

are admissible.

Theorem 2 (Substitution, ordinary). From

- \( \Delta ; \vdash e_1 : A \)
- \( \text{and} \Delta, x : A ; \Gamma \vdash e_2 : B \),

it follows that

- \( \Delta ; \Gamma \vdash [e_1/x] e_2 \)
- \( \text{and} \lbrack [e_1/x] e_2 \rbrack \delta \Gamma = [e_2] \lbrack [e_1] \delta \Gamma \text{ } \Gamma \).

Theorem 3 (Substitution, monotone). From

- \( \Delta ; \Gamma \vdash e_1 : A \)
- \( \text{and} \Delta, x : A ; \Gamma \vdash e_2 : B \),

it follows that

- \( \Delta ; \Gamma \vdash [e_1/x] e_2 \)
- \( \text{and} \lbrack [e_1/x] e_2 \rbrack \delta \Gamma = [e_2] \lbrack [e_1] \delta \Gamma \text{ } \Gamma \).

5.5 Discussion

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

However, our decomposition of this adjunction into two smaller adjunctions, with an intermediate way-station in Posetis new. By interpreting our types in the intermediate category Poset, we gain access to the comonad Disc \([A]\). This lets us distinguish between monotone and non-monotone computations, which is the critical property letting us interpret fixed points in a sensible way. Indeed, it would also have been possible to directly reflect the adjunctions in the syntax (in the style of Benton and Wadler (1996)), but we chose not to because the explicit coercions were somewhat noisy in practice. However, the ghost of this logic persists, as can be seen in the context-clearing actions in our typing rules.

6. Comparing Datalog and Datafun

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

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

This is, essentially, a fixed point computation—each stage of execution of a Datalog program takes a database and returns an extended database, until a fixed point is reached. The stratified negation restriction essentially ensures that the database transformer defined by a Datalog program is a monotone function on the set of facts. This is why the type system of Datafun tracks the monotonicity of functions — since we permit both higher-order definitions and taking fixed points, we need to ensure that the body of a fixed point definition is monotone in order to guarantee that the recursion is well-founded.

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

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

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

7. Implementation

We have built a proof-of-concept implementation of Datafun in Racket, available at [link omitted for double-blind review]. In addition to core Datafun, it supports pattern-matching, variant types, record types, dictionaries, subtyping, antitone functions, and unbounded (potentially nonterminating) fixed points. We implement everything in a naive style, and perform no optimizations.

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

1. Bidirectional type inference (Pierce and Turner 2000) determines whether \( \lambda \)s and applications are ordinary or monotone.
∀ so that, for example, \(\lambda f.\lambda x.fx\) want to add support for polymorphism over the tones of function, morphism in the style of Dunfield and Krishnaswami (2013). How-

\[
s_{\text{case } e} = \begin{cases} \text{in}_1 u & \rightarrow e_1; \text{in}_2 u & \rightarrow e_2 \end{cases}
\]

\[
[e/v]_v = e \\
[e/v]_x = x \\
[e/v]_u = \text{case } e \text{ of } \text{in}_1 u & \rightarrow e_1; \text{in}_2 u & \rightarrow e_2
\]

Figure 9. Substitution

2. For if, case, and let, the programmer annotates which form is intended; for example, \((if \ e \ \text{then} \ e_1 \ \text{else} \ e_2)\) is written \((\text{when} \ e \ \text{then} \ e_1)\) to indicate the rule if \(e\) applies.

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

8. Related and Future Work

Aggregation Aggregation of values — for example, taking the sum \(\sum_{x \in A} f x\) of a function \(f\) across a set \(A\) — is a common and ubiquitous database operation. Datafun naturally supports semilat-
tice aggregation via \(\bigvee\), but many natural operations such as sum-

\[
\begin{align*}
[e/v]_{\text{case } e} & = \text{case } [e/v]_e \text{ of } \text{in}_1 u & \rightarrow [e/v]_e; \text{in}_2 u & \rightarrow [e/v]_e \\
[e/v]_e & = e \\
[e/v]_{e_1 \lor e_2} & = [e/v]_{e_1} \lor [e/v]_{e_2} \\
[e/v]_{\forall x \in e_1} & = \forall x \in [e/v]_{e_1} [e/v]_{e_2} \\
[e/v]_{\text{fix } x \text{ is } e'} & = \text{fix } x \text{ is } [e/v]_e \\
[e/v]_{\text{fix } x \leq e_1} & = \text{fix } x \leq [e/v]_{e_1} \\
\end{align*}
\]

Databases Datalog has sometimes been described as “relational algebra plus fixed points”, and there is a long line of work on embed-

\[
\begin{align*}
[e/v]_{\text{case } e} & = \text{case } [e/v]_e \text{ of } \text{in}_1 u & \rightarrow [e/v]_e; \text{in}_2 u & \rightarrow [e/v]_e \\
[e/v]_e & = e \\
[e/v]_{e_1 \lor e_2} & = [e/v]_{e_1} \lor [e/v]_{e_2} \\
[e/v]_{\forall x \in e_1} & = \forall x \in [e/v]_{e_1} [e/v]_{e_2} \\
[e/v]_{\text{fix } x \text{ is } e'} & = \text{fix } x \text{ is } [e/v]_e \\
[e/v]_{\text{fix } x \leq e_1} & = \text{fix } x \leq [e/v]_{e_1} \\
\end{align*}
\]

\[
\begin{align*}
[e/v]_{\text{case } e} & = \text{case } [e/v]_e \text{ of } \text{in}_1 u & \rightarrow [e/v]_e; \text{in}_2 u & \rightarrow [e/v]_e \\
[e/v]_e & = e \\
[e/v]_{e_1 \lor e_2} & = [e/v]_{e_1} \lor [e/v]_{e_2} \\
[e/v]_{\forall x \in e_1} & = \forall x \in [e/v]_{e_1} [e/v]_{e_2} \\
[e/v]_{\text{fix } x \text{ is } e'} & = \text{fix } x \text{ is } [e/v]_e \\
[e/v]_{\text{fix } x \leq e_1} & = \text{fix } x \leq [e/v]_{e_1} \\
\end{align*}
\]

Transplanting this analysis to Datafun would essentially give us a “linear Datafun” corresponding to this style of programming, where we might linear types to model features like deletion. There are many nontrivial semantic issues (e.g., how to define monotonicity), but it seems a promising question for future work.

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

\[
\begin{align*}
\forall x \in e_1 & \forall y \in e_2 \ e = \forall y \in e_2 \forall x \in e_1 \ e
\end{align*}
\]

Datafun: a Functional Datalog (PREPRINT) 10 2016/3/16
when $x, y \notin FV(e_1) \cup FV(e_2)$. But this equation does not always hold in the presence of nontermination; for example, if $e_1 = z$ and $e_2$ diverges.

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

User-Defined Posets and Semilattices The two fundamental semilattice types Datafun provides are booleans and sets; products and functions merely preserve semilattice structure where they find it. One might contemplate allowing the programmer to define their own semilattice types using something like Haskell's newtype/instance. In general, this is a difficult problem, because we may need to do serious mathematical reasoning to prove that a comparison function implements a partial ordering, or that a datatype can be equipped with a semilattice structure obeying this partial ordering which is commutative, associative and idempotent. One example of such a family of types are the lexicographic sum types. Given two posets $P$ and $Q$, their disjoint union $P + Q$ is also a poset, with left values compared by the $P$-ordering, and right values compared by the $Q$-ordering, and no ordering between left and right values. However, this is not the only way that the disjoint union could be equipped with an order structure.

For example, we could define the lexicographic sum $P < Q$, which has the same elements as the sum, but extending the coproduct order relation with the additional facts that $i_1(p) \leq i_2(q)$. Indeed, we already have a special case of this: as we noted earlier, our boolean type is not $1 + 1$, but it is $1 < 1$.

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

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

References


