Finding fixed points faster

Michael Arntzenius
University of Birmingham
daekharel@gmail.com

Abstract

I propose to talk about work-in-progress on generalising the classic Datalog optimisation *seminaïve evaluation* to the higher-order functional language Datafun.

1 Introduction

Functional programmers have learned to emulate logic programming using the effect of *nondeterminism*, usually implemented as backtracking. However, backtracking search is often inefficient; logic programmers have explored other useful strategies. Datalog [7] takes an extreme approach, allowing only predicates with finite extent. This allows bottom-up evaluation, which easily handles queries (such as transitive closure) that are inefficient to solve by brute-force search.

Datafun [2] shows that higher-order functional programs can emulate Datalog using a *bottom-up nondeterminism effect* (a *finite set monad*) combined with *monotone fixed points*. Here, we sketch the translation to Datafun of a classic Datalog optimisation, seminaïve evaluation, which avoids needlessly re-deducing facts when evaluating a recursive predicate.

Datalog folklore suggests seminaïve evaluation can be understood in terms of *derivatives* [4, 5]; we substantiate this by showing that its analogue in Datafun can be defined by applying recent work by Cai et al. [6] on derivatives for incremental computation in a higher-order setting.

2 Datalog, naïvely and seminaïvely

This simple Datalog program computes reachability in a graph, given its edge relation:

```
path(X,Y) :- edge(X,Y).
path(X,Z) :- edge(X,Y), path(Y,Z).
```

A path is either an edge, or an edge followed by a path. But how does Datalog find these paths? Let's identify a predicate with the set of argument-tuples it holds of. Then we can compute `path` as the least fixed point of this function:

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

The naïve approach is to iterate the step function, computing the sequence $\emptyset, \mathsf{step}^1(\emptyset), \mathsf{step}^2(\emptyset), \ldots$ until it reaches a fixed point $\mathsf{step}^k(\emptyset) = \mathsf{step}^{k+1}(\emptyset)$. This works, but observe that $\mathsf{step}^i(\emptyset) \subseteq \mathsf{step}^{i+1}(\emptyset)$. This means we are doing *redundant computation* — if step i appends an edge (x,y) to a path (y,z) to discover the path (x,z), step $i+1$ re-discovers it the same way. We really want to compute the *change* between iterations:

$$\begin{aligned}\mathsf{path} &= \mathsf{iterate}\ \emptyset\ \mathsf{edge} \\ \mathsf{iterate}\ x\ dx &= \mathbf{if}\ dx \subseteq x\ \mathbf{then}\ x\ \mathbf{else} \\ &\quad\ \mathsf{loop}\ (x \cup dx)\ (\delta\mathsf{step}\ dx) \\ \delta\mathsf{step}\ \mathit{dpaths} &= \{(x,z) \mid (x,y) \in \mathsf{edge}, (y,z) \in \mathit{dpaths}\}\end{aligned}$$

In Datalog, it's long been known how to safely approximate this change using a static transformation on Datalog rules (we omit its definition for space reasons); this is known as *seminaïve evaluation* [4, 5].

$$\begin{aligned}A, B &::= \mathsf{bool} \mid \{A\} \mid A \to B \mid A \Rightarrow B \\ e &::= \lambda x.\, e \mid e_1\ e_2 \mid \{\vec{e}\} \mid e_1 \cup e_2 \mid \textstyle\bigcup(x \in e_1)\ e_2 \\ &\quad\ \mathbf{fix}\ e \mid \mathbf{when}\ (e_1)\ e_2 \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3\end{aligned}$$

Figure 1. A fragment of Datafun

3 Datafun, naïvely

You've actually already seen some Datafun code: the step function in section 2! Datafun is a typed higher-order functional language equipped with a *finite set monad*, which supports set-comprehension syntax sugar in the usual way [8]. Like Datalog, Datafun is *total*: all programs terminate.

Figure 1 gives the fragment of Datafun we consider here. For the full language, see Arntzenius and Krishnaswami [2]. We write monadic bind $\bigcup(x \in e_1)\ e_2$, meaning "the union of the sets e_2 for each $x \in e_1$". Datafun can also compute *fixed points* of functions on finite sets, $\mathbf{fix}\ f$.[1]

To ensure $\mathbf{fix}\ f$ terminates, f must be (among other things) monotone ($x \subseteq y$ implies $f\ x \subseteq f\ y$), so Datafun's type system tracks monotonicity of functions and expressions. The type $A \Rightarrow B$ represents monotone functions, a subtype of all functions $A \to B$. The expression $\mathbf{when}\ (e_1)\ e_2$ yields the set e_2 if e_1 is true, and $\emptyset$ otherwise; unlike **if**, this is always monotone in e_1.

As we've seen, Datalog programs can be expressed using a combination of set operations and fixed points. For example, `path` is ($\mathbf{fix}$ step). However, the seminaïve evaluation transformation, formulated on Datalog, does not handle higher-order functions. Can we lift this limitation?

4 Derivatives for Datafun

Naïve evaluation iterates a function f. Seminaïve evaluation approximates the *change* between iterations — how does

[1] The full language generalizes both monadic bind and fixed points to *semilattice types*; for simplicity we here consider only finite sets.

$$
\begin{aligned}
\Delta\mathrm{bool} &= \mathrm{bool} \\
\Delta\{A\} &= \{A\} \\
\Delta(A \to B) &= A \to \Delta A \to \Delta B \\
\Delta(A \Rightarrow B) &= A \to \Delta A \Rightarrow \Delta B
\end{aligned}
$$

$$
\begin{aligned}
\delta x &= dx \\
\delta(\lambda x.\, e) &= \lambda x.\, \lambda dx.\, \delta e \\
\delta(e_1\ e_2) &= \delta e_1\ e_2\ \delta e_2 \\
\delta\{\vec{e}\} &= \emptyset \\
\delta(e_1 \cup e_2) &= \delta e_1 \cup \delta e_2 \\
\delta(\textstyle\bigcup(x \in e_1)\ e_2) &= \textstyle\bigcup(x \in \delta e_1)\ e_2 \\
&\cup\ \textstyle\bigcup(x \in e_1 \cup \delta e_1)\ [\mathbf{0}\, x/dx]\, \delta e_2 \\
\delta(\mathbf{fix}\ f) &= \mathbf{fix}\ (\delta f\ (\mathbf{fix}\ f)) \\
\delta(\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) &= \mathbf{if}\ e\ \mathbf{then}\ \delta e_1\ \mathbf{else}\ \delta e_2 \\
\delta(\mathbf{when}\,(e_1)\ e_2) &= \mathbf{if}\ e_1\ \mathbf{then}\ \delta e_2\ \mathbf{else} \\
&\quad\ \mathbf{when}\,(\delta e_1)\ e_2 \cup \delta e_2
\end{aligned}
$$

Figure 2. Derivatives for a fragment of Datafun

$f(x)$ change as x goes from $f^i(\emptyset)$ to $f^{i+1}(\emptyset)$? To answer this question for Datafun, we build on the *incremental λ-calculus* of Cai et al. [6], which shows how to compute the change to a function's result given a change to its input. Here we summarize their approach and how we apply it to Datafun.

To capture what change means, we assign to every type A a *change structure* $(\Delta A, \oplus, \mathbf{0})$.[2] The type ΔA represents changes to values of type A. Since the steps of a fixed point computation increase monotonically, we need only represent *increasing changes.* The function $\oplus : A \to \Delta A \to A$ applies a change to a value. Finally, $\mathbf{0} : A \to \Delta A$ gives a *zero change* such that $x \oplus \mathbf{0}\, x = x$.

In fig. 2 we give a transformation from an expression $e : A$ to its derivative $\delta e : \Delta A$, which computes how e changes as its free variables change. By convention, the change to a variable $x_i : A_i$ is given by a variable named $dx_i : \Delta A_i$.

For our purpose, the most important change structures are those on finite sets and on functions. Sets are ordered by inclusion, so increasing a set means is simply it with a set of added elements. Thus $\Delta\{A\} = \{A\}$, $\oplus = \cup$, and $\mathbf{0}\, x = \emptyset$.

Putting monotonicity aside, the change type for functions is $\Delta(A \to B) = A \to \Delta A \to \Delta B$. Why not simply $A \to \Delta B$? Because in the derivative of function application $\delta(e_1\ e_2)$, it isn't only the function that may change, but its argument!

4.1 How to compute fixed points faster

We promised an analogue of seminaïve evaluation — a way to find fixed points faster. How do derivatives help us? Well, given $\mathbf{fix}\ f$, the expression $\delta f\ x\ dx$ tells us how f changes as its argument x changes to $x \cup dx$. This is exactly what we need: to compute the change between steps in our fixed point iteration. We give the naïve and seminaïve algorithms for computing fixed points in fig. 3.

[2] Cai et al. also include an operator $\ominus$ from which $\mathbf{0}$ is derived; our restriction to just $\mathbf{0}$ is suggested by Atkey [3].

$$
\begin{aligned}
\mathbf{fix}\ f &= \mathsf{na\ddot{i}ve}_f\ \emptyset\ (f\ \emptyset) \\
\mathsf{na\ddot{i}ve}_f\ x\ \mathit{next} &= \mathbf{if}\ x = \mathit{next}\ \mathbf{then}\ x\ \mathbf{else} \\
&\quad\ \mathsf{na\ddot{i}ve}_f\ \mathit{next}\ (f\ \mathit{next})
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{fix}\ f &= \mathsf{semina\ddot{i}ve}_f\ \emptyset\ (f\ \emptyset) \\
\mathsf{semina\ddot{i}ve}_f\ x\ dx &= \mathbf{if}\ dx \subseteq x\ \mathbf{then}\ x\ \mathbf{else} \\
&\quad\ \mathsf{semina\ddot{i}ve}_f\ (x \cup dx)\ (\delta f\ x\ dx)
\end{aligned}
$$

Figure 3. Naïve and seminaïve fixed point computation

4.2 Defining $\delta(\mathbf{fix}\ f)$

The definition of $\delta(\mathbf{fix}\ f)$ in fig. 2 may seem a little mysterious. However, it's easy to show that $\delta(\mathbf{fix}\ f)$ must be a fixed point of $\delta f\ (\mathbf{fix}\ f)$:

$$
\begin{aligned}
\delta(\mathbf{fix}\ f) &= \delta(f\ (\mathbf{fix}\ f)) && \text{expand fixed point} \\
&= \delta f\ (\mathbf{fix}\ f)\ \delta(\mathbf{fix}\ f) && \text{rule for } \delta(e_1\ e_2)
\end{aligned}
$$

Which suggests the following motto:

THE DERIVATIVE OF A FIXED POINT
IS THE FIXED POINT OF ITS DERIVATIVE.

This is so beautiful it must be true. Nevertheless, we have proven it correct [1].

5 Contributions

Our main contributions are:

1. Generalising seminaïve evaluation to a higher-order functional language, Datafun, giving an optimisation for finding fixed points faster.
2. Substantiating folklore that seminaïve evaluation can be understood in terms of derivatives.
3. Extending the work of Cai et al. [6] to handle a finite set monad, fixed points, and interaction with monotonicity.

References

[1] M. Arntzenius. $\delta(\mathrm{fix}\ f) = \mathrm{fix}(\delta f\ (\mathrm{fix}\ f))$: or, static differentiation of monotone fixed points. http://www.rntz.net/files/fixderiv.pdf, May 2017. Accessed: 7 June 2018.

[2] M. Arntzenius and N. 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.

[3] R. Atkey. The incremental λ-calculus and relational parametricity. https://bentnib.org/posts/2015-04-23-incremental-lambda-calculus-and-parametricity.html, April 2015. Accessed: 7 June 2018.

[4] F. Bancilhon. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems (Islamorada)*, pages 165–178, 1985.

[5] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In C. Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986.*, pages 16–52. ACM Press, 1986. doi: 10.1145/16894.16859. URL http://doi.acm.org/10.1145/16894.16859.

[6] Y. Cai, P. G. Giarrusso, T. Rendel, and K. Ostermann. A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 145–155. ACM, 2014. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594304. URL http://doi.acm.org/10.1145/2594291.2594304.

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

[8] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992. doi: 10.1017/S0960129500001560. URL https://doi.org/10.1017/S0960129500001560.