Datafun

a functional query language

Michael Arntzenius
daekharel@gmail.com
`http://www.rntz.net/datafun`

Strange Loop, September 2017
Recurse Center, March 2018

Early stage work

What if programming languages were more like query languages?

1. **What's a functional query language?**
2. From Datalog to Datafun
3. Incremental Datafun

SQL

Parent	Child
Arathorn	Aragorn
Drogo	Frodo
Eärwen	Galadriel
Finarfin	Galadriel
⋮	⋮

```
SELECT parent
FROM parentage
WHERE child = "Galadriel"
```

Tables as sets

Parent	**Child**
Arathorn	Aragorn
Drogo	Frodo
Eärwen	Galadriel
Finarfin	Galadriel
⋮	⋮

=

// set of (parent, child) pairs
{(Arathorn, Aragorn)
, (Drogo, Frodo)
, (Eärwen, Galadriel)
, (Finarfin, Galadriel)
...}

Tuples and sets are just datatypes!

Tuples and sets are just datatypes!

If tables are sets, what are queries?

Queries as set comprehensions

```
SELECT parent
FROM parentage
WHERE child = "Galadriel"
```

Queries as set comprehensions

```
SELECT parent
FROM parentage
WHERE child = "Galadriel"
```

$$\Longrightarrow$$

```
{ parent | (parent, child) in parentage
         , child = "Galadriel" }
```

Queries as set comprehensions: finding siblings

```
SELECT DISTINCT A.child, B.child
FROM parentage A INNER JOIN parentage B
ON A.parent = B.parent
WHERE A.child <> B.child
```

$$\Longrightarrow$$

```
{ (a,b) | (parent, a) in parentage
        , (parent, b) in parentage
        , not (a = b) }
```

Queries as set comprehensions: finding siblings

```
SELECT DISTINCT A.child, B.child
FROM parentage A INNER JOIN parentage B
ON A.parent = B.parent
WHERE A.child <> B.child
```

$$\Longrightarrow$$

```
{ (a,b) | (parent, a) in parentage
        , (parent, b) in parentage
        , not (a = b) }
```

Recipe for a functional query language

1. Take a functional language
2. Add sets and set comprehensions
3. ... done?

But can it go fast?

Loop reordering

```
{ ... | x in EXPR1, y in EXPR2 }
```

$$=?$$

```
{ ... | y in EXPR2, x in EXPR1 }
```

Loop reordering

```
{ ... | x in EXPR1, y in EXPR2 }
```

$$\neq$$

```
{ ... | y in EXPR2, x in EXPR1 }
```

1. Side-effects
2. Nontermination

Loop reordering

```
{ print x | x in {"hello"}, y in {0,1} }
```

$$\neq$$

```
{ print x | y in {0,1}, x in {"hello"} }
```

1. Side-effects
2. Nontermination

Loop reordering

$$\{\ \ldots\ |\ \texttt{x in } \{\},\ \texttt{y in } \infty\texttt{-loop}\ \} \implies \{\}$$

$$\neq$$

$$\{\ \ldots\ |\ \texttt{y in } \infty\texttt{-loop},\ \texttt{x in } \{\}\ \} \implies \infty\text{-loop}$$

1. Side-effects
2. Nontermination

Recipe for a functional query language, v2

1. Take a **pure, total** functional language
2. Add sets and set comprehensions
3. **Optimize!**

What have we gained?

- Can factor out repeated patterns with **higher-order functions**
- Sets are **just ordinary values**
- Sets, bags, lists: choose your container semantics!

What have we gained?

- Can factor out repeated patterns with **higher-order functions**
- Sets are **just ordinary values**
- Sets, bags, lists: choose your container semantics!

At what cost?

- **Implementation complexity**:
  GC, closures, nested sets, optimizing comprehensions...
- **Re-inventing the wheel**:
  persistence, transactions, replication...

1. What's a functional query language?
2. **From Datalog to Datafun**
3. Incremental Datafun

Parent	**Child**
Arathorn	Aragorn
Drogo	Frodo
Eärwen	Galadriel
Finarfin	Galadriel
⋮	⋮

Is Eärendil one of Aragorn's ancestors?

Datalog in a nutshell

X is Z's ancestor if X is Z's parent.

X is Z's ancestor if X is Y's parent and Y is Z's ancestor.

Datalog in a nutshell

ancestor(X,Z) if parent(X, Z).

ancestor(X, Z) if parent(X, Y) and ancestor(Y, Z).

Datalog in a nutshell

ancestor(X,Z) :- parent(X, Z).

ancestor(X, Z) :- parent(X, Y), ancestor(Y, Z).

Datalog is **deductive**: it chases rules to their logical conclusions.

Can we capture this feature **functionally**?

Procedure:

1. Pick a rule.
2. Find facts satisfying its premises.
3. Add its conclusion to the known facts.

Rules:

```
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

Facts:

```
parent(Idril, Eärendil).
parent(Eärendil, Elros).
```

Procedure:

1. Pick a rule.
2. Find facts satisfying its premises.
3. Add its conclusion to the known facts.

Rules:

ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).

Facts:

parent(Idril, Eärendil).
parent(Eärendil, Elros).

Procedure:

1. Pick a rule.
2. Find facts satisfying its premises.
3. Add its conclusion to the known facts.

Rules:

> **ancestor(X,Z) :- parent(X,Z).**
> ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).

Facts:

> parent(Idril, Eärendil).
> parent(Eärendil, Elros).

Procedure:

1. Pick a rule.
2. Find facts satisfying its premises.
3. Add its conclusion to the known facts.

Rules:

ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).

Facts:

parent(Idril, Eärendil).
parent(Eärendil, Elros).
ancestor(Idril, Eärendil). *(new!)*

Procedure:

1. Pick a rule.
2. Find facts satisfying its premises.
3. Add its conclusion to the known facts.

Rules:

```
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

Facts:

```
parent(Idril, Eärendil).
parent(Eärendil, Elros).
ancestor(Idril, Eärendil).
```

Procedure:

1. Pick a rule.
2. Find facts satisfying its premises.
3. Add its conclusion to the known facts.

Rules:

ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).

Facts:

parent(Idril, Eärendil).
parent(Eärendil, Elros).
ancestor(Idril, Eärendil).
ancestor(Eärendil, Elros). *(new!)*

Procedure:

1. Pick a rule.
2. Find facts satisfying its premises.
3. Add its conclusion to the known facts.

Rules:

```
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

Facts:

```
parent(Idril, Eärendil).
parent(Eärendil, Elros).
ancestor(Idril, Eärendil).
ancestor(Eärendil, Elros).
```

Procedure:

1. Pick a rule.
2. Find facts satisfying its premises.
3. Add its conclusion to the known facts.

Rules:

ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).

Facts:

parent(Idril, Eärendil).
parent(Eärendil, Elros).
ancestor(Idril, Eärendil).
ancestor(Eärendil, Elros).
ancestor(Idril, Elros). *(new!)*

Procedure:

1. Pick a rule.
2. Find facts satisfying its premises.
3. Add its conclusion to the known facts.

Rules:

```
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

Facts:

```
parent(Idril, Eärendil).
parent(Eärendil, Elros).
ancestor(Idril, Eärendil).
ancestor(Eärendil, Elros).
ancestor(Idril, Elros).
```

Repeatedly apply a **set of rules**
until **nothing changes**

Repeatedly apply a **function**
until **nothing changes**

Repeatedly apply a **function**
until **its output equals its input**

Repeatedly apply a **function**
until **its output equals its input**
i.e. it reaches a **fixed point**

Repeatedly apply a **function**
until **its output equals its input**
i.e. it reaches a **fixed point**

$$\text{fix x} = \ldots\ \textit{function of x}\ \ldots$$

```
// Datalog
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).

// Datafun
fix ancestor = parent
             ∪ {(x,z) | (x,y) in parent
                      , (y,z) in ancestor}
```

```
// Datalog
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).

// Datafun
fix ancestor = parent
             ∪ {(x,z) | (x,y) in parent
                      , (y,z) in ancestor}
```

```
// Datalog
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

```
// Datafun
fix ancestor = parent
             ∪ {(x,z) | (x,y) in parent
                      , (y,z) in ancestor}
```

Repeatedly applying:

$$X \longmapsto \text{parent} \cup \{(\text{x,z}) \mid (\text{x,y}) \text{ in parent, } (\text{y,z}) \text{ in } X\}$$

Where parent $=$ {(Idril, Eärendil), (Eärendil, Elros)}

Steps:

$$\emptyset$$

Repeatedly applying:

$$X \longmapsto \text{parent} \cup \{(\text{x,z}) \mid (\text{x,y}) \text{ in parent}, (\text{y,z}) \text{ in X}\}$$

Where parent = {(Idril, Eärendil), (Eärendil, Elros)}

Steps:

$$\emptyset$$

$$\longmapsto \quad \text{parent} \cup \{(\text{x,z}) \mid (\text{x,y}) \text{ in parent}, (\text{y,z}) \text{ in } \emptyset\}$$

Repeatedly applying:

$$X \longmapsto \text{parent} \cup \{(\text{x,z}) \mid (\text{x,y}) \text{ in parent, } (\text{y,z}) \text{ in } X\}$$

Where parent = {(Idril, Eärendil), (Eärendil, Elros)}

Steps:

$$\begin{aligned} & \emptyset \\ \longmapsto \quad & \text{parent} \cup \{(\text{x,z}) \mid (\text{x,y}) \text{ in parent, } (\text{y,z}) \text{ in } \emptyset\} \\ = \quad & \text{parent} \end{aligned}$$

Repeatedly applying:

$$X \longmapsto \text{parent} \cup \{(\text{x,z}) \mid (\text{x,y}) \text{ in parent, } (\text{y,z}) \text{ in } X\}$$

Where parent = {(Idril, Eärendil), (Eärendil, Elros)}

Steps:

$$\begin{aligned} & \emptyset \\ \longmapsto \quad & \text{parent} \cup \{(\text{x,z}) \mid (\text{x,y}) \text{ in parent, } (\text{y,z}) \text{ in } \emptyset\} \\ = \quad & \text{parent} \\ \longmapsto \quad & \text{parent} \cup \{(\text{x,z}) \mid (\text{x,y}) \text{ in parent, } (\text{y,z}) \text{ in parent}\} \end{aligned}$$

Repeatedly applying:

$$X \longmapsto \text{parent} \cup \{(\text{x,z}) \mid (\text{x,y}) \text{ in parent, } (\text{y,z}) \text{ in } X\}$$

Where parent $=$ {(Idril, Eärendil), (Eärendil, Elros)}

Steps:

$$\begin{aligned} & \emptyset \\ \longmapsto\quad & \text{parent} \cup \{(\text{x,z}) \mid (\text{x,y}) \text{ in parent, } (\text{y,z}) \text{ in } \emptyset\} \\ =\quad & \text{parent} \\ \longmapsto\quad & \text{parent} \cup \{(\text{x,z}) \mid (\text{x,y}) \text{ in parent, } (\text{y,z}) \text{ in parent}\} \\ =\quad & \{(\text{Idril, Eärendil}), (\text{Eärendil, Elros}), (\text{Idril, Elros})\} \end{aligned}$$

But can it go fast?

1. What's a functional query language?

2. From Datalog to Datafun

3. **Incremental Datafun**

Three problems

1. **View maintenance:**
   How do we update a cached query efficiently after a mutation?

Three problems

1. **View maintenance:**
   How do we update a cached query efficiently after a mutation?

2. **Seminaïve evaluation in Datalog:**
   How do we avoid re-deducing facts we already know?

Three problems

1. **View maintenance:**
   How do we update a cached query efficiently after a mutation?
2. **Seminaïve evaluation in Datalog:**
   How do we avoid re-deducing facts we already know?
3. **Incremental computation:**
   How do we efficiently recompute a function as its inputs change?

Three problems

1. **View maintenance:**
   How do we update a cached query efficiently after a mutation?

2. **Seminaïve evaluation in Datalog:**
   How do we avoid re-deducing facts we already know?

3. **Incremental computation:**
   How do we efficiently recompute a function as its inputs change?

"A Theory of Changes for Higher-Order Languages: Incrementalizing λ-calculi by Static Differentiation"
[PLDI 2014]

by Yufei Cai, Paolo G Giarrusso, Tillmann Rendel, and Klaus Ostermann

Static differentiation

Every type A has a *type of changes*, ΔA.

Static differentiation

Every type A has a *type of changes*, ΔA.

$$\begin{aligned} \Delta\mathbb{N} &= \mathbb{Z} \\ \Delta(A \times B) &= \Delta A \times \Delta B \end{aligned}$$

Static differentiation

Every type A has a *type of changes*, ΔA.

$$\begin{aligned}\Delta\mathbb{N} &= \mathbb{Z}\\ \Delta(A \times B) &= \Delta A \times \Delta B\end{aligned}$$

Every type also gets an operator $\oplus_A : A \to \Delta A \to A$.

Static differentiation

Every type A has a *type of changes*, ΔA.

$$\begin{aligned} \Delta\mathbb{N} &= \mathbb{Z} \\ \Delta(A \times B) &= \Delta A \times \Delta B \end{aligned}$$

Every type also gets an operator $\oplus_A : A \to \Delta A \to A$.

$$\begin{aligned} x \oplus_{\mathbb{N}} dx &= x + dx \\ (x, y) \oplus_{A \times B} (dx, dy) &= (x \oplus_A dx, y \oplus_B dy) \end{aligned}$$

Static differentiation

Every type A has a *type of changes*, ΔA.

$$\begin{aligned} \Delta\mathbb{N} &= \mathbb{Z} \\ \Delta(A \times B) &= \Delta A \times \Delta B \end{aligned}$$

Every type also gets an operator $\oplus_A : A \to \Delta A \to A$.

$$\begin{aligned} x \oplus_{\mathbb{N}} dx &= x + dx \\ (x, y) \oplus_{A \times B} (dx, dy) &= (x \oplus_A dx, y \oplus_B dy) \end{aligned}$$

A function $f : A \to B$ gets a *derivative*, $\delta f : A \to \Delta A \to \Delta B$.

Static differentiation

Every type A has a *type of changes*, ΔA.

$$\begin{aligned} \Delta\mathbb{N} &= \mathbb{Z} \\ \Delta(A \times B) &= \Delta A \times \Delta B \end{aligned}$$

Every type also gets an operator $\oplus_A : A \to \Delta A \to A$.

$$\begin{aligned} x \oplus_{\mathbb{N}} dx &= x + dx \\ (x, y) \oplus_{A \times B} (dx, dy) &= (x \oplus_A dx, y \oplus_B dy) \end{aligned}$$

A function $f : A \to B$ gets a *derivative*, $\delta f : A \to \Delta A \to \Delta B$.

$$\begin{aligned} f(x) &= x^2 \\ \delta f(x)(dx) &= 2x \cdot dx + dx^2 \end{aligned}$$

Static differentiation

Every type A has a *type of changes*, ΔA.

$$\begin{aligned}\Delta\mathbb{N} &= \mathbb{Z}\\ \Delta(A \times B) &= \Delta A \times \Delta B\end{aligned}$$

Every type also gets an operator $\oplus_A : A \to \Delta A \to A$.

$$\begin{aligned}x \oplus_{\mathbb{N}} dx &= x + dx\\ (x, y) \oplus_{A \times B} (dx, dy) &= (x \oplus_A dx, y \oplus_B dy)\end{aligned}$$

A function $f : A \to B$ gets a *derivative*, $\delta f : A \to \Delta A \to \Delta B$.

$$\begin{aligned}f(x) &= x^2\\ \delta f(x)(dx) &= 2x \cdot dx + dx^2\\ f(x) + \delta f(x)(dx) &= x^2 + 2x \cdot dx + dx^2 \quad = \quad (x + dx)^2\end{aligned}$$

We've extended this technique to handle all of Datafun!

(As of about three weeks ago.)

Finding fixed points faster with derivatives

The naïve way to find fixed points looks like this:

$$\emptyset \mapsto f(\emptyset) \mapsto f^2(\emptyset) \mapsto f^3(\emptyset) \mapsto \ldots$$

Finding fixed points faster with derivatives

The naïve way to find fixed points looks like this:

$$\emptyset \mapsto f(\emptyset) \mapsto f^2(\emptyset) \mapsto f^3(\emptyset) \mapsto \ldots$$

$f^i(\emptyset)$ and $f^{i+1}(\emptyset)$ **overlap a lot.**

Computing $f^{i+1}(\emptyset)$ from $f^i(\emptyset)$ **does a lot of recomputation.**

Finding fixed points faster with derivatives

The naïve way to find fixed points looks like this:

$$\emptyset \mapsto f(\emptyset) \mapsto f^2(\emptyset) \mapsto f^3(\emptyset) \mapsto \ldots$$

$f^i(\emptyset)$ and $f^{i+1}(\emptyset)$ **overlap a lot.**

Computing $f^{i+1}(\emptyset)$ from $f^i(\emptyset)$ **does a lot of recomputation.**

What if we could only compute **what changed** between iterations?

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

Theorem: $x_i = f^i(x)$

Takeaways

1. Set comprehensions = queries
2. Fixed points = recursive queries *(like Datalog)*
3. Incremental computation = faster fixed points
4. Datafun has all three!*

* In theory.

Michael Arntzenius
daekharel@gmail.com
@arntzenius

rntz.net/datafun