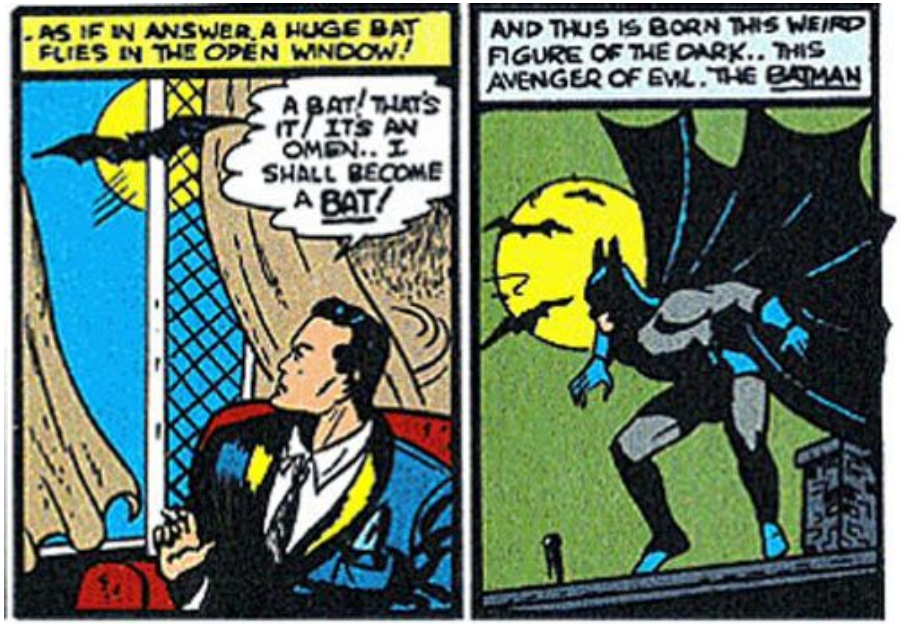
(technology presentation)

2021



Origin story

6 years ago...

in a powerpoint far, far away

The Haskell Spreadsheet

goals

streams
(Reflex)

correctness

content-addressable
(Unison)

reactive
(spreadsheets)

user-friendly FP
(Elm)

real data types
(trees/graphs/lists)

pure functional

SaaS

browser-based
(Excel/Goog/AirTable)

polymorphic variants
(OCaml)

row-types
(PureScript)

re-use Haskell vocab
(take/any/filter/scan)

stepper
(Duet)

ad-hoc typing

june 2017

Duet is an educational dialect of Haskell aimed at interactivity. This is a demonstration page of the work-in-progress implementation, compiled to JavaScript, consisting of a type-checker and interpreter.

Input program

Arithmetic ▾

```
main = 2 * (10 - (5 + -3))
```

Steps

Complete output ▾ ☐ **Show dictionaries**

```
2 * (10 - (5 + -3))
2 * (10 - 2)
2 * 8
16
```

value1 = 115

some_table =

12
6
9
115

value1 = 115

some_table =

```
[2 * 6, 6, 9, value1]
```

(prototype)

march 2020

value1 = 115

some_table =

12
value1 * 3
9
115

INFLEX

(real implementation)

New declaration

table2

age	name
38	"Giulia"
21	"Chris"
53	"Dave"

row

age	40
name	"Giulia"

nested rows

after	**age**	40
	name	"Giulia"
before	**age**	40
	name	"Giulia"

may 2020

table

age	name
38	"Giulia"
21	"Chris"
53	"Dave"

inflex

Undo Redo New Formula Text List Table Import Logout

server_expenses ×

	expense ×	price ×	+
1 ×	db	15.00	
2 ×	droplet	40.00	
3 ×	loadbalancer	10.00	
+			

expenses_pie ×

$schema: https://vega.github.io/schema/vega-lite/v4.json

description: A simple pie chart with embedded data.

data

values

	price ×	expense ×	+
1 ×	18.00	db	
2 ×	48.00	droplet	
3 ×	12.00	loadbalancer	
+			

mark: arc

encoding

theta: field: price; type: quantitative

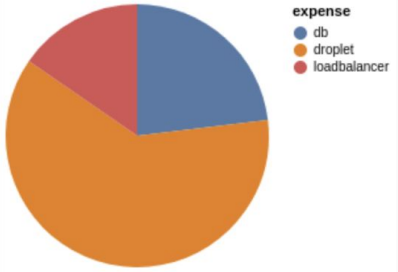
color: field: expense; type: nominal

view: stroke: null

taxed ×

	price ×	expense ×	+
1 ×	18.00	db	
2 ×	48.00	droplet	
3 ×	12.00	loadbalancer	
+			

expenses ×



total ×

#ok
78.00

feb 2021

(latest version)

```
{x: 1, y: "foo"}.x
```

(records)

```
#ok(123)  #none
```

(variants)

syntax

```
map(x:x*2,[1,2,3])
```

(func() syntax and shorter lambdas, arrays)

fidelity

(this is new)

server_expenses

	expense	price
1	db	15.00
2	droplet	40.00
3	loadbalancer	10.00

Changes go in both directions

server_expenses


```
[{"expense": "db", "price": 15.00}, {"expense": "droplet", "price": 40}, {"expense": "loadbalancer", "price": 10}]
```


(tables are just lists of records)

This is what Elmologists actually believe.

Equality

```
(==) : a -> a -> Bool
```

Check if values are “the same”.

Note: Elm uses structural equality on tuples, records, and user-defined union types. This means the values `(3, 4)` and `(3, 4)` are definitely equal. This is not true in languages like JavaScript that use reference equality on objects.

Note: Do not use `(==)` with functions, JSON values from `elm/json`, or regular expressions from `elm/regex`. It does not work. It will crash if possible. With JSON values, decode to Elm values before doing any equality checks!

Why is it like this? Equality in the Elm sense can be difficult or impossible to compute. Proving that functions are the same is undecidable, and JSON values can come in through ports and have functions, cycles, and new JS data types that interact weirdly with our equality implementation. In a future release, the compiler will detect when `(==)` is used with problematic types and provide a helpful error message at compile time. This will require some pretty serious infrastructure work, so the stopgap is to crash as quickly as possible.

```
1 * 2 :: Multiply a => a
```

(numbers are overloaded)

type classes

Type classes > TDNR

and also <, <=, =, /=

```
#foo :: <foo({})|v>

#ok("hi") :: <ok(Text)|v>
```

polymorphic variants

#ok(x)/#none,
#true/#false,
#red/#black#/blue

duality of row types

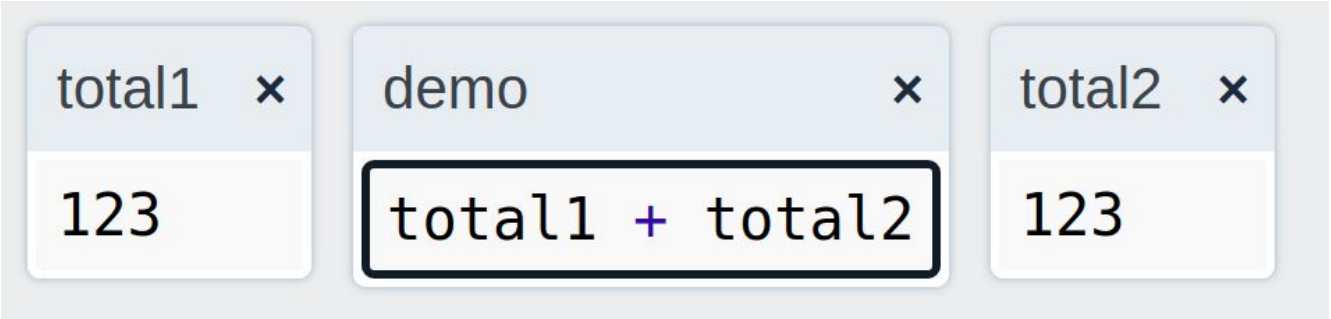
	Record (products)	**Variant** (sums)
Construct	`{x:1}` `{x::Int}` (closed)	`#x(1)` `<x::Int|v>` (open)
Deconstruct	`r.x` `r :: {x::a|v}` (open)	`case v { #x(i): i*2 }` `v :: <x::a>` (closed)

functions

```
map      :: (a -> b) -> [a] -> [b]
filter   :: (a -> <#true|#false>) -> [a] -> [a]
sum      :: Addable a => [a] -> <#ok(a)|#sum_empty>
average  :: Addable a, Divisible a => [a] -> <#ok(a)|#average_empty>
vega     :: a -> VegaChart
null     :: [a] -> <#true|#false>
length   :: FromInteger number => [a] -> number
distinct :: Comparable a => [a] -> [a]
minimum  :: Comparable a => [a] -> <#ok(a)|#minimum_empty>
maximum  :: Comparable a => [a] -> <#ok(a)|#maximum_empty>
sort     :: Comparable a => [a] -> [a]
find     :: (a -> <#true|#false>) -> [a] -> <#find_empty|#find_failed|#ok(a)>
all      :: (a -> <#true|#false>) -> [a] -> <#all_empty|#ok(a)>
any      :: (a -> <#true|#false>) -> [a] -> <#any_empty|#ok(a)>
from_ok  :: a -> <#ok(a)|v> -> a
```

CAS can cache

(original code)



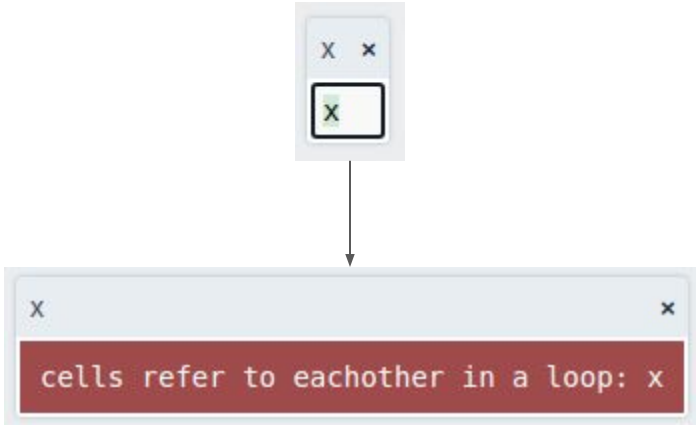
(compiled code)

#abcsdkfj

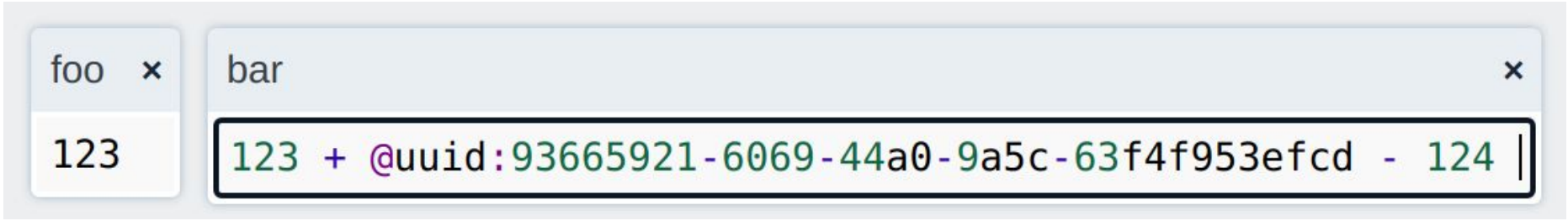
#abcsdkfj + **#abcsdkfj**

#abcsdkfj

CAS can't self-reference



(hint: you need `let` for recursion)

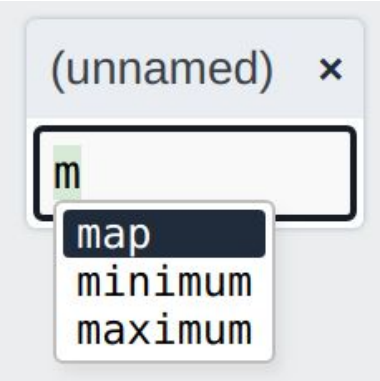


Overlays make the UUIDs pretty in the editor (CodeMirror).

CAS avoids namespacing

(upcoming work)

You want "map" (for lists) and "map" (for streams) and "map" (for text)? Sure, have all three. Completion handles that.



Names in detail

- References to other cells within a doc are UUIDs: *@uuid:sd9f87s-fd9g87df-98sd7f...* (contents is **supposed to change** when dependency cells change)
- References to other document cells are CAS SHA512s: *@sha512:s9df87sd9f87sd9f7sdf9...* (we **don't allow** other documents to break ours by changing a definition)
- References to local names (lambdas) are just plain words: *foo* (but are immediately deBruijn'd by the renamer).
- References to primitives: *@prim:array-length* (these aren't supposed to change behavior, and will get a new name when they do)

time (no "volatile" cells)

push events

buttons

Reflex-like streams (Event/Behavior/ Dynamic)

streams/feeds

databases

(future work)

can't define
your own
types (yet)

```
1.03 :: FromDecimal n 2 => n
```

```
1.03 * 1.004 :: FromDecimal n 3 => n
```

decimal places are important

(aka floating point is bad)

stack

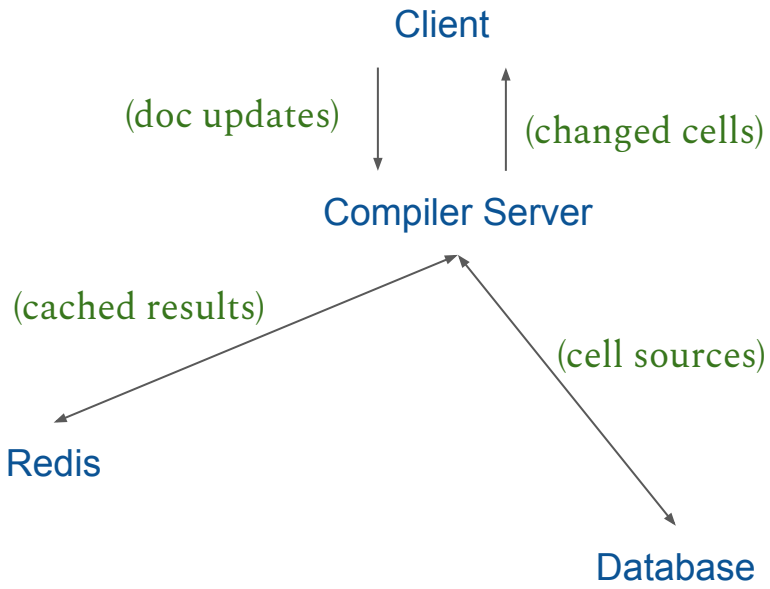
Compiler & Server

Client

Database

Deployment

arch


Client
(doc updates)
(changed cells)
Compiler Server
(cached results)
(cell sources)
Redis
Database

Halogen

CodeMirror

client libs

Vega-Lite

compiler pipeline

Lexer	Generate tokens "[", "1", "]"
Parser	Make an AST e.g. List (Int 1) or Lam "x" (Var "x")
Renamer	Rename local vars "x" => $1 (locals take precedence over globals)
Filler	Fill in globals which are hashes #x9c8vx98sdufs9df.... (remaining names must be globals/other cells)
Generator	Generate type constraints from syntax [a ~ Either b Int, b ~ Text, ...]
Solver	Solve constraints and unify: x -> Either Text Int
Generaliser	Generalise type variables to poly types: forall x. x -> Either Text Int
Resolver	Resolve type classes to dictionaries: Ord a -> a -> Bool
Defaulter	Default type class constraints like numbers: FromInteger a => a to Integer

End