

A close-up photograph of various tools on a light-colored wooden surface. At the top, a metal ruler with inch markings is visible. Below it, a black metal square is positioned vertically, featuring a brass-colored adjustment knob and a spirit level. To the right of the square, a single, small metal nail lies horizontally. The text "Tiny Type Tools" is overlaid in the center in a white, sans-serif font, enclosed in a thin white rectangular border.

Tiny Type Tools

Haskell

Tools



GHC

Type holes



Hoogle

Type search



Pointfree

Remove lambdas



Djinn

Code synthesis



QuickSpec

Equational laws



FitSpec

Refining property tests



Pointful

Insert lambdas



Wingman

Case splits, code synthesis

Libraries

- Expose the core algorithm
- Reuse across different languages
- Returns useful data structures
- Enables rich user experience with types
 - Tell me cardinality, i.e. how many inhabitants
 - Show me similar functions which already exist
 - Synthesise code
 - Gives equational laws



Type Algebra

- $\text{Void} \sim 0$
- $() \sim 1$
- $\text{Bool} \sim 2$
- $(\text{forall } a. a \rightarrow a) \sim 1$
- $(\text{forall } a. a \rightarrow a \rightarrow a) \sim 2$
- $\text{Either } a \ b \sim (a + b)$
- $(a, b) \sim (a * b)$
- $[()] \sim \infty$

Counting type inhabitants.

Last modified on April 20, 2019 by Alex

Preamble

For Scala readers: Haskell's `Void` type can be assumed to be roughly equivalent to `Nothing`. Neither has any inhabitants other than diverging or erroneous expressions that fail at runtime. `Maybe` is the same as `Option`, except its empty case is called `Nothing` and its non-empty case is called `Just`.

For Haskell readers: `Any` in Scala is roughly equivalent to

```
data Any = forall a. Any a
```

the least informative existential type.

<https://web.archive.org/web/2020111192703/https://alexknvl.com/posts/counting-type-inhabitants.html>

Type Formulas

- $(a, b) \sim (b, a)$
- $\text{Either } a \ b \sim \text{Either } b \ a$

- $((a, b) \rightarrow c) \sim (a \rightarrow b \rightarrow c)$
- $(a \rightarrow b \rightarrow c) \sim ((a, b) \rightarrow c)$
- $(\text{Either } a \ b \rightarrow c) \sim (a \rightarrow c, b \rightarrow c)$

- $(\text{forall } x. (a \rightarrow x) \rightarrow f \ x) \sim f \ a$
- $(\text{forall } x. (x \rightarrow a) \rightarrow f \ x) \sim f \ a$

Rewrite Rules - Commutative

```
commutative :: Algebra x -> Maybe (Algebra x)
commutative (Product a b) =
  Just (Product b a)
commutative (Sum a b) =
  Just (Sum b a)
commutative _ =
  Nothing
```

Rewrite Rules - Curry Product

```
curryProduct :: Algebra x -> Maybe (Algebra x)
curryProduct (Exponent c (Product a b)) =
  Just (Exponent (Exponent c b) a)
curryProduct _ =
  Nothing
```


Rewrite Rules - Combined

```
rules :: Ord x => [(RewriteLabel, Rule [] (Algebra x))]
rules =
  [ (RewriteYonedaCovariant, rule yonedaCovariant),
    (RewriteYonedaContravariant, rule yonedaContravariant),
    (RewriteMoveForall, rule moveForall),
    (RewriteRemoveForall, rule removeForall),
    (RewriteArithmetic, rule arithmetic),
    (RewriteCurrySum, rule currySum),
    (RewriteCurryProduct, rule curryProduct),
    (RewriteUncurryProduct, rule uncurryProduct),
    (RewriteAssociative, rule associative),
    (RewriteDistributive, rule distributive),
    (RewriteCommutative, rule commutative),
    (RewriteIntroduceArity, rule introduceArity)
  ]
```

Heuristics

- Disincentivise terms
 - Terms have cost 5
 - Functions have cost 10
 - Foralls have cost 20
- Disincentivise rules
 - Each rule has cost 1
 - Commutative has cost 2

Pretty printed

```
∀ a. ∀ b. (a -> b) -> a + 1 -> b + 1
= ∀ a. a + 1 -> a + 1    -- via covariant yoneda lemma
= ∀ a. (a -> a + 1) * (1 -> a + 1)    -- via curry sum
= ∀ a. (a -> a + 1) * (a + 1)    -- via arithmetic
= (∀ a. a -> a + 1) * (∀ a. a + 1)    -- via distributive
= (∀ a. (1 -> a) -> a + 1) * (∀ a. a + 1)    -- via introduce
arity
= (1 + 1) * (∀ a. a + 1)    -- via covariant yoneda lemma
= 2 * (∀ a. a + 1)    -- via arithmetic
= 2 * (∀ a. (0 -> a) -> a + 1)    -- via introduce arity
= 2 * (0 + 1)    -- via covariant yoneda lemma
= 2 * 1    -- via arithmetic
= 2    -- via arithmetic
```

$\forall a. \forall b. (a \rightarrow b) \rightarrow ((a + 1) \rightarrow (b + 1))$	$= \forall a. (a + 1) \rightarrow (a + 1)$	(covariant yoneda lemma)
	$= \forall a. a \rightarrow (a + 1) * 1 \rightarrow (a + 1)$	(curry sum)
	$= \forall a. a \rightarrow (a + 1) * (a + 1)$	(arithmetic)
	$= (\forall a. a \rightarrow (a + 1)) * (\forall a. a + 1)$	(distributive)
	$= (\forall a. (1 \rightarrow a) \rightarrow (a + 1)) * (\forall a. a + 1)$	(introduce arity)
	$= (1 + 1) * (\forall a. a + 1)$	(covariant yoneda lemma)
	$= 2 * (\forall a. a + 1)$	(arithmetic)
	$= 2 * (\forall a. (0 \rightarrow a) \rightarrow (a + 1))$	(introduce arity)
	$= 2 * (0 + 1)$	(covariant yoneda lemma)
	$= 2 * 1$	(arithmetic)
	$= 2$	(arithmetic)

Rendered via MathJax

type-algebra, next steps

<https://github.com/puffnfresh/type-algebra>

- Publish on Hackage
- Properly handle infinite types
- Quick/dirty translation from Haskell types
- General problem:
 - Run a bunch of rewrites
 - Apply a heuristic
 - Search for solution(s)
- Are there useful rewrites that are missing?

Packages

- is:exact
- base
- ghc
- haskell-gi-base
- relude
- xmonad-contrib
- Cabal
- base-prelude
- rio
- numeric-prelude
- dimensional
- pqueue
- ghc-lib-parser
- rebase
- numhask
- LambdaHack
- mixed-types-num
- ye
- br
- dis
- factory
- hledger-web

:: (a -> b) -> [a] -> [b]

map :: (a -> b) -> [a] -> [b]

base Prelude Data.List GHC.Base GHC.List GHC.OldList, ghc GHC.Prelude, haskell-gi-base Data.GI.Base.ShortPrelude, relude Relude.List.Reexport, xmonad-contrib XMonad.Config.Prime

map f xs is the list obtained by applying f to each element of xs, i.e.,

map :: (a -> b) -> [a] -> [b]

Cabal Distribution.Compat.Prelude.Internal, base-prelude BasePrelude, rio RIO.List RIO.Prelude, numeric-prelude NumericPrelude NumericPrelude.Base, dimensional Numeric.Units.Dimensional.Prelude, pqueue Data.PQueue.Max, ghc-lib-parser GHC.Prelude, rebase Rebase.Prelude, numhask NumHask.Prelude, LambdaHack Game.LambdaHack.Core.Prelude Game.LambdaHack.Core.Prelude, mixed-types Numeric.MixedTypes.PreludeHiding, yesod-paginator Yesod.Paginator.Prelude, brittany Language.Haskell.Brittany.Internal.Prelude, distribution-opensuse OpenSuse.Prelude, factory Factory.Prelude, hledger-web Hledger.Web.Import

map f xs is the list obtained by applying f to each element of xs, i.e.,

strictMap :: (a -> b) -> [a] -> [b]

ghc GHC.Utils.Misc, ghc-lib-parser GHC.Utils.Misc

map :: (a -> b) -> [a] -> [b]

llvm-hs-pure LLVM.Prelude

map f xs is the list obtained by applying f to each element of xs, i.e.,

Type Search using Hoogle

Tuesday, June 09, 2020

Hoogle Searching Overview

Summary: Hoogle 5 has three interesting parts, a pipeline, database and search algorithm.

The Haskell search engine [Hoogle](#) has gone through five major designs, the first four of which are described in [these slides from TFP 2011](#). [Hoogle version 5](#) was designed to be a complete rewrite which simplified the design and allowed it to scale to all of [Hackage](#). All versions of Hoogle have had some preprocessing step which consumes Haskell definitions, and writes out a data file. They then have the search phase which uses that data file to perform searches. In this post I'll go through three parts -- what the data file looks like, how we generate it, and how we search it. When we consider these three parts, the evolution of Hoogle can be seen as:

- Versions 1-3, produce fairly simple data files, then do an expensive search on top. Fails to scale to large sizes.
- Version 4, produce a very elaborate data files, aiming to search quickly on top. Failed because producing the data file required a lot of babysitting and a long time, so was updated very rarely (yearly). Also, searching a complex data file ends up with a lot of corner cases which have terrible complexity (e.g. `a -> a -> a -> a -> a` would kill the server).
- Version 5, generate very simple data files, then do $O(n)$ but small-constant multiplier searching on top. Update the files daily and automatically. Make search time very consistent.

Version 5 data file

By version 5 I had realised that deserialising the data file was both time consuming and memory hungry. Therefore, in version 5, the data file consists of chunks of data that can be memory-mapped into `Vector` and `ByteString` chunks using a `ForeignPtr` underlying storage. The OS figures out which bits of the data file should be paged in, and there is very little overhead or complexity on the H

9 <http://neilmitchell.blogspot.com/2020/06/hoogle-searching-overview.html>

```
data NamesSize a where NamesSize :: NamesSize Int
```

```
data NamesItems a where NamesItems :: NamesItems (V.Vector TargetId)
```

Type Search

- Fingerprints types into a database
 - Arity: $(a \rightarrow b) = 1$
 - Terms: $(a \rightarrow b) = 2$
 - Rares: $(\text{Settings } a \rightarrow \text{Program } b) = [\text{“Settings”}, \text{“Program”}]$
- Applies some rewrites to the query
- Finds nearest fingerprints
- Unifies the signatures

Type for Searching

runSearch

`:: Monad m`

`=> Search m a a' b b' c`

`-> [(b, a)]`

`-> a`

`-> b`

`-> SerialT m (b, c)`

Structured Type for Searching

Search

(Writer [SearchLog a n])

(Fingerprint a)

(FingerprintCost a)

(Signature n)

SignatureCost

(StructuredCosts a)

Query string: `Int -> String`

Results:

1. `Int -> String`

2. `a -> String`

3. `Int -> b`

4. `a -> b`

type-search

<https://github.com/puffnfresh/type-search>

- Found and fixed bugs in Hoogle
- Could almost be swapped into Hoogle
- Has commonalities with type-algebra:
 - Ranking by cost
 - Rewrites
 - Searching for solutions

Djinn

```
f :: a -> Maybe a -> a
```

Submit

```
f :: a -> Maybe a -> a
f a b =
  case b of
    Nothing -> a
    Just c -> c
-- or
f a _ = a
```

<https://www.hedonisticlearning.com/djinn/>

For example, give QuickSpec the functions `reverse`, `++` and `[]`, and it will find six laws:

```
reverse [] == []
xs ++ [] == xs
[] ++ xs == xs
reverse (reverse xs) == xs
(xs ++ ys) ++ zs == xs ++ (ys ++ zs)
reverse xs ++ reverse ys == reverse (ys ++ xs)
```

<https://hackage.haskell.org/package/quickspec>

```
-- Evaluate `repeat`
main = do
    print 1
    foldr (>>) (return ()) (take 2 (print 1:repeat (print 1)))

-- Evaluate `take`
main = do
    print 1
    foldr (>>) (return ()) (print 1:take 1 (repeat (print 1)))

-- Evaluate `foldr`
main = do
    print 1
    print 1
    foldr (>>) (return ()) (take 1 (repeat (print 1)))
```

<https://www.haskellforall.com/2013/12/equational-reasoning.html>



World Domination