gulcii

Claude Heiland-Allen

2010–2019

Contents

1 doc/encoding.md

```
# Lambda calculus encodings

## Notation

Backslash is used for lambda:

* easier to type
* familiar from Haskell

Neighbouring lambdas can be combined with one backslash:

    \a b s z . a s (b s z)

Corresponds to:

    \a. \b. \s. \z. a s (b s z)


## Church and Scott encodings

* encode data as lambda terms
* continuation passing style
* Church folds vs Scott case analysis


## Continuation passing style

* the datum is a black box that knows itself
* the datum is passed functions that it calls with its deconstruction
* the datum has one argument per constructor
* each continuation has one argument per constructor argument


## Simple types

Church and Scott encoding coincide for simple types.


## Bool

Haskell:

    data Bool
      = False
      | True

Church, Scott:

    true  = \t f . t
    false = \t f . f

    and = \a b . a b a
    or  = \a b . a a b
    not = \a . a false true
```

```
The datum "true" takes two arguments, and returns the first, which (by
convention) denotes the value True.


## Pair

Haskell:

    data Pair a b
      = Pair a b

Church, Scott:

    pair = \a b p . p a b

    fst  = \p . p (\a b . a)
    snd  = \p . p (\a b . b)

The datum "pair x y" takes one argument, which is a function of two arguments,
and passes it the stored values of "x" and "y".


## Maybe

Haskell:

    data Maybe a
      = Nothing
      | Just a

Church, Scott:

    nothing = \n j . n
    just    = \a n j . j a

    maybe   = \n j m . m n j


## Either

Haskell:

    data Either a b
      = Left a
      | Right b

Church, Scott:

    left   = \a l r . l a
    right  = \b l r . r b

    either = \l r e . e l r


## Recursive types

* Church and Scott encoding differ for recursive types.
```

```
* Church encoding uses folds.

    The deconstruction continuation threads throughout
    the structure.

* Scott encoding is similar to case analysis.

    The deconstruction continuation
    unwraps one layer of constructors only.


## Natural numbers

Haskell:

    data Nat
      = Zero
      | Succ Nat

Church:

    zero = \s z . z
    succ = \n s z . s (n s z)

Scott:

    zero = \s z . z
    succ = \n s z . s n


## Nat examples

Haskell:

    Zero, Succ Zero, Succ(Succ Zero), Succ(Succ(Succ Zero))

Church, applying the same "s" "n" times at once:

    \s z . z
    \s z . s z
    \s z . s (s z)
    \s z . s (s (s z))

Scott, applying different "s" "n" times separately:

    \s z . z
    \s z . s (\s z . z)
    \s z . s (\s z . s (\s z . z))
    \s z . s (\s z . s (\s z . s (\s z . z)))


## Church Nat succ

    zero = \s z . z
    succ = \n s z . s (n s z)
```

```
    succ zero
  = {- definition of succ -}
    (\n s z . s (n s z)) zero
  = {- beta -}
    \s z . s (zero s z)
  = {- definition of zero -}
    \s z . s ((\s z . z) s z)
  = {- beta -}
    \s z . s ((\z . z) z)
  = {- beta -}
    \s z . s z
  = {- definition of one -}
    one
```

Scott Nat succ

```
zero = \s z . z
succ = \n s z . s n
```

```
    succ zero
  = {- definition of succ -}
    (\n s z . s n) zero
  = {- beta -}
    \s z . s zero
  = {- definition of zero -}
    \s z . s (\s z . z)
  = {- definition of one -}
    one
```

Nat arithmetic

Church:

```
add = \m n . m succ n
mul = \m n . m (add n) zero
exp = \m n . n m
```

Scott, open terms with ‘letrec‘:

```
add = \m n . m (\p . succ  (add p n)) n
mul = \m n . m (\p . add n (mul p n)) zero
exp = \m n . n (\p . mul m (exp m p)) one
```

Scott, closed terms with ‘fix‘:

```
add = fix (\add . \m n . m (\p . succ  (add p n)) n)
mul = fix (\mul . \m n . m (\p . add n (mul p n)) zero)
exp = fix (\exp . \m n . n (\p . mul m (exp m p)) one)
```

Fixed point combinator

Semantics:

```
fix f = f (fix f)
```

Implementation:

```
    \f . (\x . f (x x)) (\x . f (x x))
```

What it computes:

* The unique least fixed point under the definedness order.
* Allows recursive functions to be defined as closed terms.

Nat predecessor

Church (courtesy Wikipedia):

```
    pred = \n f x . n (\g h . h (g f)) (\u . x) (\v . v)
```

Scott:

```
    pred = \n . n (\p . p) zero
```

Nat conversion

Church arithmetic is more concise (and doesn't need `fix`).

Scott predecessor is comprehensible.

Mix and match?

```
    churchToScott = \n . n scottSucc scottZero

    scottToChurch = \n . n
        (\p . churchSucc (scottToChurch p))
        churchZero

    scottToChurch = fix (\scottToChurch . \n . n
        (\p . churchSucc (scottToChurch p))
        churchZero)
```

Nat subtract and equality

Church:

```
    sub = \m n . n pred m
```

Scott:

```
    sub = \m n . m
      (\p . n (\q . sub p q) m)
      zero
    equal = \m n . m
      (\p . n (\q . equal p q) false)
      (n (\q . false) true)
```

Unwrap a layer of constructor from each number and recurse.

There is a different ‘equal‘ for booleans:

```
equalBool = \a b . a b (not b)
```

List

Haskell:

```
data List a
  = Nil
  | Cons a (List a)
```

Church:

```
nil  = \c n . n
cons = \x xs c n . c x (xs c n)
```

Scott:

```
nil  = \c n . n
cons = \x xs c n . c x xs
```

List operations

Church, Scott:

```
isnil  = \l . l (\x xs . false) true
head   = \l . l (\x xs . x) error
```

Church (‘tail‘ courtesy Wikipedia):

```
length = \l . l (\x xs . succ xs) zero
tail   = \l c n . l
    (\x xs g . g x (xs c)) (\xs . n) (\x xs . xs)
```

Scott:

```
length = \l . l (\x xs . succ (length xs)) zero
tail   = \l . l (\x xs . xs) nil
```

More Scott functions

```
compose = \f g x . f (g x)

fold = \f e l . l (\x xs . f x (fold f e xs)) e

sum  = fold add zero
ands = fold and true
ors  = fold or false

map = \f . fold (compose cons f) nil
all = \f . compose ands (map f)
```

```
    any = \f . compose ors (map f)

    take = \n l. n(\p. l (\x xs. cons x (take p xs))nil)nil
    drop = \n l. n(\p. l (\x xs. drop p xs) nil) l

    iterate = \f x . cons x (iterate f (f x))


# How to perform lambda calculus

## How to perform lambda calculus

* single step graph reduction

* visualisation of current state

* sonification of changes in state

* open terms vs closed terms


## Graph reduction

    data Term
      = Free String
      | Reference Integer
      | Bound           -- de Bruijn index 0
      | Scope Term      -- see Lambdascope paper
      | Lambda Strategy Term
      | Apply Term Term

    reduce
      :: Definitions   -- Map String  Term
      -> References    -- Map Integer Term
      -> Term
      -> Maybe (References, Term)


## Three kinds of Lambda

* strict (syntax inspired by Haskell's -XBangPatterns):

        (\v ! s) t

    t is fully reduced before substitution into s.

* copy:

        (\v ? s) t

    t is substituted for each occurrence of v in s.

* lazy:

        (\v . s) t
```

```
    a new `Reference` is created for t, and substituted into s.

    `reduce` reduces *inside* the `References` until it is irreducible, at which
    point the `Reference` is replaced with the `Term` it refers to.


## Visualisation

![](visualisation.png){width=100%}\

## Sonification

* count number of nodes of each type

* statistics are forwarded to a Pure-data patch

* changes in each count control a harmonic (one for each type of node) in a
  simple phase modulation synth


## Open terms

* free variables looked up on demand from environment

* allows definitions to be changed at runtime

* easier to write


## Drawbacks of open terms

* no sharing

    subterms can be evaluated many times due to duplication

* exponential work (worst case)

* exponential space (worst case)


## Fixed points

* closed terms with fixed point combinators

* allows evaluation to be shared

* sharing can be vital for efficiency


# Future work


## Better Evaluator

* current evaluator is still somewhat ad-hoc and doesn't preserve sharing

* previous evaluator even had correctness bugs
```

```
* switch to using Lambdascope (or similar) as a library?


## Auto Fix

Automatically translating open terms to use fixed point combinators:

* recursive functions can use `fix`
* mutually recursive functions can use `fix` combined with tuples

        many = some `orElse` none
        some = one `andThen` many

    becomes:

        manysome = fix (\p -> pair
            (snd p `orElse` none)
            (one `andThen` fst p) )
        many = fst manysome
        some = snd manysome


## Magic It

* refer to previously evaluated terms
* including the currently evaluating term
* without restarting evaluation

Haskell example (`ghci-8.0.1`):

    > 3
    3
    > it + 5
    8
    > it * 2
    16


## Further Performances / Project Ideas

* "An infinite deal of nothing", a variety of non-terminating loops each with
  their own intrinsic computational rhythm.

* Implement in untyped lambda calculus an interpreter for a known
  Turing-complete tape mutation based language and run some simple programs
  in it.

    Illustrates Turing-completeness of untyped lambda calculus, albeit slowly.


# EOF

## EOF

Thanks!
```

```
Questions?

    https://mathr.co.uk
    mailto:claude@mathr.co.uk

    https://hackage.haskell.org/package/gulcii
    https://code.mathr.co.uk/gulcii
```

2 doc/encoding.tex

```
\documentclass[aspectratio=43]{beamer}
\usepackage{lmodern}
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}
\DeclareUnicodeCharacter{03BB}{\(\lambda\)}
\AtBeginSection[]{\begin{frame}\frametitle{Outline}\tableofcontents[↙
    ↳ currentsection]\end{frame}}
\def\tightlist{}

\definecolor{mathr}{RGB}{128,64,64}
\setbeamercolor{structure}{fg=mathr}
\setbeamercolor{title}{fg=mathr}
\setbeamercolor{titlelike}{fg=mathr}
\beamertemplatenavigationsymbolsempty

\renewcommand{\familydefault}{\sfdefault}

\title[GULCII]{Graphical\\ Untyped Lambda Calculus\\ Interactive Interpreter}
\subtitle{(GULCII)}
\author{Claude Heiland-Allen}
\institute{\url{https://mathr.co.uk} \\ \url{mailto:claude@mathr.co.uk}}
\date[Ed 2017]{Edinburgh, 2017}
\subject{Computer Science}

\begin{document}

\begin{frame} \titlepage \end{frame}
\begin{frame} \frametitle{Outline} \tableofcontents \end{frame}

\input{encoding.md.tex}

\end{document}
```

3 doc/evaluation.tex

```
% note: much of this document is out of date

\documentclass[aspectratio=149]{beamer}
\AtBeginSection[]{\begin{frame}\frametitle{GULCII}\tableofcontents[↙
    ↳ currentsection]\end{frame}}

\usepackage{listings}

\title[GULCII]{Graphical\\ Untyped Lambda Calculus\\ Interactive Interpreter}
\subtitle{(GULCII)}
\author{Claude Heiland-Allen}
```

```
\institute{\url{https://mathr.co.uk} \\ \url{mailto:claude@mathr.co.uk}}
\date[Ed 2017]{Edinburgh, 2017}
\subject{Computer Science}

\begin{document}

\begin{frame} \titlepage
\end{frame}

\section[bruijn]{De Bruijn Indexing}

\begin{frame} \frametitle{Motivation}
\begin{itemize}
\item Naming is hard.
\item \(\alpha\)-conversion for capture-avoiding
substitution is tricky to get right.
\item De Bruijn indices solve this issue neatly.
\end{itemize}
\end{frame}

\begin{frame}[fragile] \frametitle{De Bruijn Terms}
\begin{lstlisting}[language=Haskell]
data Term
  = Bound Integer
  | Lambda Term
  | Apply Term Term
\end{lstlisting}
\begin{itemize}
\item The bound variable indexes outwards to the corresponding lambda.
\end{itemize}
\[ \begin{aligned}
\text{\tt{Lambda (Lambda (Lambda (Bound {\bf 0})))}} =& \lambda x . \lambda y . ↙
    ↪ \lambda \mathbf{z} . \mathbf{z} \\
\text{\tt{Lambda (Lambda (Lambda (Bound {\bf 1})))}} =& \lambda x . \lambda \↙
    ↪ mathbf{y} . \lambda z . \mathbf{y} \\
\text{\tt{Lambda (Lambda (Lambda (Bound {\bf 2})))}} =& \lambda \mathbf{x} . \↙
    ↪ lambda y . \lambda z . \mathbf{x} \\
%\text{\tt{L(A(L(A(B0)(L(B0))))(L(A(B1)(B0))))}} =&\\ \lambda z . (\lambda y . y↙
    ↪  (\lambda x . x)) (\lambda w . z w)
\end{aligned} \]
\end{frame}

\begin{frame}[fragile] \frametitle{Beta Reduction}
\begin{lstlisting}[language=Haskell]
beta :: Integer -> Term -> Term -> Term
beta i s@(Bound j) t = if i == j then t else s
beta i (Lambda s)  t = Lambda (beta (i + 1) s t)
beta i (Apply a b) t = Apply (beta i a t) (beta i b t)
\end{lstlisting}
\end{frame}

\begin{frame}[fragile] \frametitle{De Bruijn Terms in GULCII}
\begin{centering}\begin{lstlisting}[language=Haskell]
data Term
  = Free String
  | Bound Integer
  | Lambda Strategy Term
```

```
    | Apply Term Term
\end{lstlisting}\end{centering}
\begin{itemize}
\item An additional constructor for named free variables.
\item An additional strategy parameter for lambdas.
\end{itemize}
\end{frame}

\section[strategies]{Evaluation Strategies}

\begin{frame}[fragile] \frametitle{Evaluation Strategies}
\begin{lstlisting}[language=Haskell]
data Strategy
  = Lazy
  | Strict
  | Copy
\end{lstlisting}
\end{frame}

\begin{frame}[fragile] \frametitle{Copy Evaluation}
\begin{lstlisting}[language=Haskell]
y = (\ x ? A) B
\end{lstlisting}
\begin{itemize}
\item Evaluation of \(y\) proceeds by substituting \(B\) into all
      occurences of \(x\) in \(A\), before anything else is evaluated.
\item If \(B\) is not needed by the result, it will not be evaluated.
\item If \(B\) is needed by the result more than once, it will be
      evaluated more than once.
\end{itemize}
\end{frame}

\begin{frame}[fragile] \frametitle{Strict Evaluation}
\begin{lstlisting}[language=Haskell]
y = (\ x ! A) B
\end{lstlisting}
\begin{itemize}
\item Evaluation of \(y\) proceeds by first completely evaluating \(B\)
      before substituting it into occurences of \(x\) in \(A\).
\item If \(B\) is not needed by the result, more work than necessary
      will be performed.
\item Syntax in GULCII inspired by Haskell's \tt{-XBangPatterns}.
\end{itemize}
\end{frame}

\begin{frame}[fragile] \frametitle{Lazy Evaluation}
\begin{lstlisting}[language=Haskell]
y = (\ x . A) B
\end{lstlisting}
\begin{itemize}
\item Evaluation of \(y\) shares the evaluation of \(B\) over all
      occurences of \(x\) in \(A\), and \(B\) is evaluated on demand
      as needed.
\item If \(B\) is not needed by the result, it will not be evaluated.
\item If \(B\) is needed by the result more than once, it will be
      evaluated only once, with the result shared.
\end{itemize}
```

```
\end{frame}

\section[graph]{Graph Reduction}

\begin{frame}[fragile] \frametitle{Graph Reduction - Types}
\begin{lstlisting}[language=Haskell]
data Term
  = Free String
  | Bound Integer
  | Lambda Strategy Term
  | Apply Term Term
  | Reference Integer

type References = Map Integer Term
type Definitions = Map String Term

reduce
  :: Definitions
  -> References -> Term
  -> Maybe (References, Term)
\end{lstlisting}
\end{frame}

\begin{frame}[fragile] \frametitle{Graph Reduction - Free}
\begin{lstlisting}[language=Haskell]
reduce defs refs (Free var)
  = (,) refs `fmap` Map.lookup var defs
\end{lstlisting}
\end{frame}

\begin{frame}[fragile] \frametitle{Graph Reduction - Bound}
\begin{lstlisting}[language=Haskell]
reduce defs refs (Bound var)
  = Nothing
\end{lstlisting}
\end{frame}

\begin{frame}[fragile] \frametitle{Graph Reduction - Lambda}
\begin{lstlisting}[language=Haskell]
reduce defs refs (Lambda strat term)
  = fmap (Lambda strat) `fmap` reduce defs refs term
\end{lstlisting}
\end{frame}

\begin{frame}[fragile] \frametitle{Graph Reduction - Reference}
\begin{lstlisting}[language=Haskell]
reduce defs refs term@(Reference ref)
  = case Map.lookup ref refs of
      Just refTerm ->
        case reduce defs refs refTerm of
          Just (refs', term') ->
            Just (Map.insert ref term' refs', term)
          Nothing ->
            Just (refs, refTerm)
      Nothing -> error "reference not found"
\end{lstlisting}
\end{frame}
```

```
\begin{frame}[fragile] \frametitle{Graph Reduction - Apply Lambda Copy}
\begin{lstlisting}[language=Haskell]
reduce defs refs (Apply (Lambda Copy a) b)
  = Just (refs, beta 0 a b)

beta :: Integer -> Term -> Term -> Term
beta i s@(Bound j)  t = if i == j then t else s
beta i (Lambda k s) t = Lambda k (beta (i + 1) s t)
beta i (Apply a b)  t = Apply (beta i a t) (beta i b t)
beta i s            t = s
\end{lstlisting}
\end{frame}

\begin{frame}[fragile] \frametitle{Graph Reduction - Apply Lambda Strict}
\begin{lstlisting}[language=Haskell]
reduce defs refs (Apply l@(Lambda Strict a) b)
  = case reduce defs refs b of
      Just (refs', b') -> Just (refs', Apply l b')
      Nothing -> Just (refs, beta 0 a b)
\end{lstlisting}
\end{frame}

\begin{frame}[fragile] \frametitle{Graph Reduction - Apply Lambda Lazy}
\begin{lstlisting}[language=Haskell]
reduce defs refs (Apply (Lambda Lazy a) b)
  = let r = next refs
    in  Just ( Map.insert r b refs
             , beta 0 a (Reference r) )

next :: Map Integer v -> Integer
next refs = case Map.maxViewWithKey refs of
  Nothing          -> 0
  Just ((k, _), _) -> k + 1
\end{lstlisting}
\end{frame}

\begin{frame}[fragile] \frametitle{Graph Reduction - Apply}
\begin{lstlisting}[language=Haskell]
reduce defs refs (Apply a b)
  = case reduce defs refs a of
      Just (refs', a') ->
        Just (refs', Apply a' b)
      Nothing ->
        fmap (Apply a) `fmap` reduce defs refs b
\end{lstlisting}
\end{frame}

\section[garbage]{Garbage Collection}

\begin{frame} \frametitle{The Problems}
\begin{itemize}
\item
The References map may contain terms that are no longer referenced.

This leaks memory in the interpreter, which may be problematic for
long-running programs.
```

```
235
    \item
    There may end up a long chain of references, adding additional layers
    of indirection, that will only be collapsed when the deepest term is
    irreducible.
240
    \item
    We want to replace references with their terms as soon as they occur
    only once.

245 \end{itemize}
    \end{frame}

    \begin{frame}[fragile] \frametitle{Example Evaluation With GC Disabled}
    \begin{lstlisting}[language=Haskell,basicstyle=\small,mathescape]
250   zero = \s z . z
      succ = \n s z . s (n s z)
      double = \m . m succ m

      (double (succ zero))
255 $\to$       $\cdots$
      (\.(\.(1 ((#6 1) 0))))
       where
        #0 = (\.(\.(1 0)))
        #1 = (\.(\.0))
260     #2 = 1
        #3 = 0
        #4 = (\.(\.(\.(1 ((2 1) 0)))))
        #5 = #0
        #6 = #5
265 $\to$       $\cdots$
    \end{lstlisting}

    Term $\to \#6 \to \#5 \to \#0 \to$ is a chain of indirection.

270 References $\#1, \#2, \#3, \#4$ are all unreachable.
    \end{frame}

    \begin{frame} \frametitle{Example Evaluation With GC Disabled}
    \only<1>{\includegraphics[width=\linewidth]{gulcii-15}}%
275 \end{frame}

    \begin{frame}[fragile] \frametitle{Garbage Collection}
    \begin{lstlisting}[language=Haskell]
    gc
280   :: References -> Term
      -> (References, Term)

    gc refs term
      = let counts = refCount refs term Map.empty
285     in ( Map.fromList
               [ (r, compact counts refs (refs Map.! r))
               | (r, n) <- Map.toList counts
               , n > 1 ]
           , compact counts refs term )
290 \end{lstlisting}
    \end{frame}
```

```
\begin{frame}[fragile] \frametitle{GC Helpers}
\begin{lstlisting}[language=Haskell]
type Count = Integer

refCount
  :: References -> Term
  -> Map Integer Count -> Map Integer Count

compact
  :: Map Integer Count -> References
  -> Term -> Term
\end{lstlisting}
\end{frame}

\begin{frame}[fragile] \frametitle{Reference Counting}
\begin{lstlisting}[language=Haskell]
refCount refs (Lambda strat a) counts
  = refCount refs a counts

refCount refs (Apply a b) counts
  = refCount refs a (refCount refs b counts)

refCount refs (Reference r) counts
  = case r `Map.lookup` counts of
      Just n ->
        Map.insert r (n + 1) counts
      Nothing ->
        refCount refs (refs Map.! r)
                 (Map.insert r 1 counts)

refCount refs term m = m
\end{lstlisting}
\end{frame}

\begin{frame}[fragile] \frametitle{Compaction}
\begin{lstlisting}[language=Haskell]
compact counts refs term@(Reference r)
  = case r `Map.lookup` counts of
      Just 1 -> compact counts refs (refs Map.! r)
      _ -> term

compact counts refs (Lambda strat term)
  = Lambda strat (compact counts refs term)

compact counts refs (Apply a b)
  = Apply (compact counts refs a)
          (compact counts refs b)

compact counts refs term = term
\end{lstlisting}
\end{frame}

\section[examples]{Examples}

\begin{frame} \frametitle{succ zero}
\only<1>{\includegraphics[width=\linewidth]{succ-zero/gulcii-0}}%
```

```
\only<2>{\includegraphics[width=\linewidth]{succ-zero/gulcii-1}}%
\only<3>{\includegraphics[width=\linewidth]{succ-zero/gulcii-2}}%
\only<4>{\includegraphics[width=\linewidth]{succ-zero/gulcii-3}}%
\only<5>{\includegraphics[width=\linewidth]{succ-zero/gulcii-4}}%
\only<6>{\includegraphics[width=\linewidth]{succ-zero/gulcii-5}}%

apply is orange, lambda is magenta, variable is purple, free variable is cyan
\end{frame}

\begin{frame}[fragile] \frametitle{double (succ zero) - Copy}
\begin{lstlisting}[language=Haskell]
  succ = \n s z ? s (n s z)
  zero = \s z ? z
  double = \m ? m succ m
\end{lstlisting}
\end{frame}

\begin{frame} \frametitle{double (succ zero) - Copy}
\only<1>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii-0}}%
\only<2>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii-1}}%
\only<3>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii-2}}%
\only<4>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii-3}}%
\only<5>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii-4}}%
\only<6>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii-5}}%
\only<7>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii-6}}%
\only<8>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii-7}}%
\only<9>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii-8}}%
\only<10>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii-9}}%
\only<11>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii
    ↪ -10}}%
\only<12>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii
    ↪ -11}}%
\only<13>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii
    ↪ -12}}%
\only<14>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii
    ↪ -13}}%
\only<15>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii
    ↪ -14}}%
\only<16>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii
    ↪ -15}}%
\only<17>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii
    ↪ -16}}%
\only<18>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii
    ↪ -17}}%
\only<19>{\includegraphics[width=\linewidth]{double-(succ-zero)-copy/gulcii
    ↪ -18}}%

copy-lambda is pink
\end{frame}

\begin{frame}[fragile] \frametitle{double (succ zero) - Strict}
\begin{lstlisting}[language=Haskell]
  succ = \n s z ! s (n s z)
  zero = \s z ! z
  double = \m ! m succ m
\end{lstlisting}
\end{frame}
```

```
\begin{frame} \frametitle{double (succ zero) - Strict}
\only<1>{\includegraphics[width=\linewidth]{double-(succ-zero)-strict/gulcii-0}}%
\only<2>{\includegraphics[width=\linewidth]{double-(succ-zero)-strict/gulcii-1}}%
\only<3>{\includegraphics[width=\linewidth]{double-(succ-zero)-strict/gulcii-2}}%
\only<4>{\includegraphics[width=\linewidth]{double-(succ-zero)-strict/gulcii-3}}%
\only<5>{\includegraphics[width=\linewidth]{double-(succ-zero)-strict/gulcii-4}}%
\only<6>{\includegraphics[width=\linewidth]{double-(succ-zero)-strict/gulcii-5}}%
\only<7>{\includegraphics[width=\linewidth]{double-(succ-zero)-strict/gulcii-6}}%
\only<8>{\includegraphics[width=\linewidth]{double-(succ-zero)-strict/gulcii-7}}%
\only<9>{\includegraphics[width=\linewidth]{double-(succ-zero)-strict/gulcii-8}}%
\only<10>{\includegraphics[width=\linewidth]{double-(succ-zero)-strict/gulcii-9}}%
\only<11>{\includegraphics[width=\linewidth]{double-(succ-zero)-strict/gulcii-10}}%
\only<12>{\includegraphics[width=\linewidth]{double-(succ-zero)-strict/gulcii-11}}%
\only<13>{\includegraphics[width=\linewidth]{double-(succ-zero)-strict/gulcii-12}}%
\only<14>{\includegraphics[width=\linewidth]{double-(succ-zero)-strict/gulcii-13}}%

strict-lambda is red
\end{frame}


\begin{frame}[fragile] \frametitle{double (succ zero) - Lazy}
\begin{lstlisting}[language=Haskell]
  succ = \n s z . s (n s z)
  zero = \s z . z
  double = \m . m succ m
\end{lstlisting}
\end{frame}

\begin{frame} \frametitle{double (succ zero) - Lazy}
\only<1>{\includegraphics[width=\linewidth]{double-(succ-zero)-lazy/gulcii-0}}%
\only<2>{\includegraphics[width=\linewidth]{double-(succ-zero)-lazy/gulcii-1}}%
\only<3>{\includegraphics[width=\linewidth]{double-(succ-zero)-lazy/gulcii-2}}%
\only<4>{\includegraphics[width=\linewidth]{double-(succ-zero)-lazy/gulcii-3}}%
\only<5>{\includegraphics[width=\linewidth]{double-(succ-zero)-lazy/gulcii-4}}%
\only<6>{\includegraphics[width=\linewidth]{double-(succ-zero)-lazy/gulcii-5}}%
\only<7>{\includegraphics[width=\linewidth]{double-(succ-zero)-lazy/gulcii-6}}%
\only<8>{\includegraphics[width=\linewidth]{double-(succ-zero)-lazy/gulcii-7}}%
\only<9>{\includegraphics[width=\linewidth]{double-(succ-zero)-lazy/gulcii-8}}%
\only<10>{\includegraphics[width=\linewidth]{double-(succ-zero)-lazy/gulcii-9}}%
\only<11>{\includegraphics[width=\linewidth]{double-(succ-zero)-lazy/gulcii-10}}%
\only<12>{\includegraphics[width=\linewidth]{double-(succ-zero)-lazy/gulcii
```

```
    ↪ -11}}%
\only<13>{\includegraphics[width=\linewidth]{double-(succ-zero)-lazy/gulcii ↙
    ↪ -12}}%
\only<14>{\includegraphics[width=\linewidth]{double-(succ-zero)-lazy/gulcii ↙
    ↪ -13}}%
\only<15>{\includegraphics[width=\linewidth]{double-(succ-zero)-lazy/gulcii ↙
    ↪ -14}}%

references are green (first occurence) and turquoise (later occurences)
\end{frame}

\begin{frame}[fragile] \frametitle{Evaluation Strategy Comparison 1/3}
\begin{lstlisting}[language=Haskell]
zero = \s z @ z
succ = \n s z @ s (n s z)
double = \n @ n succ n
const = \a b @ a
join = \f a @ f a a

join const (double (succ zero))
\end{lstlisting}
\begin{itemize}
\item @ = ? Copy : 16 beta reductions, 8 free variable instantiations
\item @ = ! Strict : 13 beta reductions, 6 free variable instantiations
\item @ = . Lazy : 13 beta reductions, 6 free variable instantiations
\end{itemize}
\end{frame}

\begin{frame}[fragile] \frametitle{Evaluation Strategy Comparison 2/3}
\begin{lstlisting}[language=Haskell]
id = \x @ x
loop = (\x @ x x) (\x @ x x )
const = \a b @ a

const id loop
\end{lstlisting}
\begin{itemize}
\item @ = ? Copy : 2 beta reductions, 2 free variable instantiations
\item @ = ! Strict : does not terminate
\item @ = . Lazy : 2 beta reductions, 2 free variable instantiations
\end{itemize}
\end{frame}

\begin{frame}[fragile] \frametitle{Evaluation Strategy Comparison 3/3}
\begin{itemize}
\item Copy evaluation is super-easy to implement, but inefficient.

\item Strict evaluation can reduce space usage.

\item
Lazy evaluation terminates more often than strict evaluation, and is
just as efficient in terms of number of reductions.

\item In GULCII we can mix and match within a single term.
\end{itemize}

\end{frame}
```

```
\end{document}
```

4 doc/.gitignore

```
*.aux
*.log
*.nav
*.out
*.pdf
*.snm
*.toc
*.vrb
*.md.tex
```

5 doc/Makefile

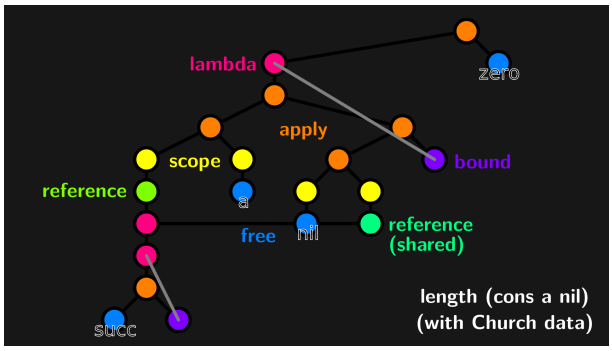
```
all: encoding.pdf

clean:
	-rm -f encoding.aux encoding.log encoding.md.tex encoding.nav encoding.↙
	    ↳ out encoding.pdf encoding.snm encoding.toc encoding.vrb

encoding.pdf: encoding.tex encoding.md.tex
	pdflatex encoding.tex
	pdflatex encoding.tex
	pdflatex encoding.tex

encoding.md.tex: encoding.md
	pandoc -t beamer --slide-level 2 encoding.md -o encoding.md.tex
```

6 doc/visualisation.png



7 extra/gulcii_m.pd

```
#N canvas 3 58 450 300 10;
#X obj 21 22 receive~ \$1-p;
#X obj 20 218 throw~ \$1-l;
#X obj 97 218 throw~ \$1-r;
#X obj 21 134 expr~ sin($v1*6.283185307179586);
#X obj 97 157 expr~ sin($v1*6.283185307179586);
#X obj 147 19 inlet;
```

```
#X obj 147 40 float;
#X obj 147 61 vline~;
#X obj 21 43 *~ \$2;
#X obj 97 189 *~;
#X obj 20 192 *~;
#X obj 96 106 +~ 0.25;
#X obj 147 82 hip~ 1;
#X obj 147 103 lop~ 15;
#X connect 0 0 8 0;
#X connect 3 0 10 0;
#X connect 4 0 9 0;
#X connect 5 0 6 0;
#X connect 6 0 7 0;
#X connect 7 0 12 0;
#X connect 8 0 3 0;
#X connect 8 0 11 0;
#X connect 9 0 2 0;
#X connect 10 0 1 0;
#X connect 11 0 4 0;
#X connect 12 0 13 0;
#X connect 13 0 9 1;
#X connect 13 0 10 1;
```

8 extra/gulcii.pd

```
#N canvas 3 58 335 369 10;
#X obj 18 10 netreceive 8765;
#X obj 21 101 loadbang;
#X obj 21 122 delay 1000;
#X msg 22 148 \; pd dsp 1;
#X obj 121 40 print;
#X obj 141 252 catch~ \$0-r;
#X obj 70 209 gulcii_v \$0 0;
#X obj 27 251 catch~ \$0-l;
#X obj 19 40 s \$0-control;
#X obj 168 126 list prepend set;
#X obj 168 148 list trim;
#X msg 168 170 rebound succ;
#X obj 59 65 route statistics;
#X obj 90 310 dac~;
#X obj 27 273 expr~ tanh($v1);
#X obj 142 275 expr~ tanh($v1);
#X connect 0 0 8 0;
#X connect 0 0 12 0;
#X connect 0 1 4 0;
#X connect 1 0 2 0;
#X connect 2 0 3 0;
#X connect 5 0 15 0;
#X connect 7 0 14 0;
#X connect 9 0 10 0;
#X connect 10 0 11 0;
#X connect 12 1 9 0;
#X connect 14 0 13 0;
#X connect 15 0 13 1;
```

9 extra/gulcii_v.pd

```
#N canvas 3 63 472 453 10;
#X obj 46 214 vline~;
#X obj 37 392 *~;
#X obj 125 392 *~;
#X obj 182 246 send~ \$0-p;
#X obj 127 312 +~;
#X obj 143 267 catch~ \$0-l;
#X obj 223 267 catch~ \$0-r;
#X obj 209 312 +~;
#X obj 127 345 expr~ sin($v1*6.283185307179586);
#X obj 207 365 expr~ sin($v1*6.283185307179586);
#X obj 18 43 route start stop statistics;
#X obj 37 413 throw~ \$1-l;
#X obj 125 413 throw~ \$1-r;
#X obj 18 18 r \$1-control;
#X obj 358 107 gulcii_m \$0 6;
#X obj 358 128 gulcii_m \$0 5;
#X obj 358 149 gulcii_m \$0 4;
#X obj 358 170 gulcii_m \$0 3;
#X obj 358 191 gulcii_m \$0 2;
#X obj 358 212 gulcii_m \$0 1;
#X obj 143 288 *~ 0.125;
#X obj 224 288 *~ 0.125;
#X obj 133 109 +;
#X obj 163 109 +;
#X obj 193 109 +;
#X obj 223 109 +;
#X obj 253 109 +;
#X obj 132 136 vline~;
#X obj 132 179 lop~ 15;
#X obj 130 228 phasor~;
#X obj 132 158 hip~ 1;
#X obj 131 203 +~ 50;
#X msg 18 166 1 1000;
#X msg 80 168 0 1000;
#X obj 143 68 unpack f f f f f f;
#X connect 0 0 1 0;
#X connect 0 0 2 0;
#X connect 1 0 11 0;
#X connect 2 0 12 0;
#X connect 4 0 8 0;
#X connect 5 0 20 0;
#X connect 6 0 21 0;
#X connect 7 0 9 0;
#X connect 8 0 1 1;
#X connect 9 0 2 1;
#X connect 10 0 32 0;
#X connect 10 1 33 0;
#X connect 10 2 34 0;
#X connect 13 0 10 0;
#X connect 20 0 4 1;
#X connect 21 0 7 1;
#X connect 22 0 27 0;
#X connect 23 0 22 1;
#X connect 24 0 23 1;
#X connect 25 0 24 1;
#X connect 26 0 25 1;
```

```
#X connect 27 0 30 0;
#X connect 28 0 31 0;
#X connect 29 0 3 0;
#X connect 29 0 4 0;
#X connect 29 0 7 0;
#X connect 30 0 28 0;
#X connect 31 0 29 0;
#X connect 32 0 0 0;
#X connect 33 0 0 0;
#X connect 34 0 19 0;
#X connect 34 0 22 0;
#X connect 34 1 18 0;
#X connect 34 1 23 0;
#X connect 34 2 17 0;
#X connect 34 2 24 0;
#X connect 34 3 16 0;
#X connect 34 3 25 0;
#X connect 34 4 15 0;
#X connect 34 4 26 0;
#X connect 34 5 14 0;
#X connect 34 5 26 1;
```

10 extra/start.sh

```
#!/bin/sh
sudo cpufreq-set -c 0 -g performance
sudo cpufreq-set -c 1 -g performance
pd extra/gulcii.pd &
pdpid=$!
sleep 5
./.cabal-sandbox/bin/gulcii 1024 768 fullscreen | pdsend 8765
kill "${pdpid}"
```

11 .gitignore

```
/dist/
/.cabal-sandbox/
/cabal.sandbox.config
*.hi
*.o
.stack-work/
stack.yaml
```

12 gulcii.cabal

```
Name:                gulcii
Version:             0.3
Synopsis:            graphical untyped lambda calculus interactive interpreter
Description:
  GULCII is an untyped lambda calculus interpreter supporting interactive
  modification of a running program with graphical display of graph reduction.
  .
  See README.md for the user manual.

Homepage:            https://code.mathr.co.uk/gulcii
License:             GPL-2
```

```
License-file:        LICENSE
Author:              Claude Heiland-Allen
Maintainer:          claude@mathr.co.uk
Category:            Compilers/Interpreters
Build-type:          Simple
Cabal-version:       >=1.6
Tested-With:         GHC==8.2.1

extra-source-files:  doc/Makefile
                     doc/encoding.md
                     doc/encoding.tex
                     doc/visualisation.png
                     README.md

Data-files:          lib/bits.gu,
                     lib/bool.gu,
                     lib/church.gu,
                     lib/either.gu,
                     lib/function.gu,
                     lib/list.gu,
                     lib/maybe.gu,
                     lib/natural.gu,
                     lib/pair.gu,
                     lib/prelude.gu,
                     lib/edinburgh.gu,
                     extra/gulcii.pd,
                     extra/gulcii_m.pd,
                     extra/gulcii_v.pd

Executable gulcii
  HS-source-dirs:    src
  Main-is:           Main.hs
  Build-depends:     base >= 3 && < 6,
                     containers >= 0.3 && < 0.7,
                     filepath >= 1.1 && < 1.5,
                     gtk >= 0.11 && < 0.16,
                     cairo >= 0.11 && < 0.14
  Other-modules:     Bruijn
                     Command
                     Draw
                     Evaluation
                     GC
                     Graph
                     Lambda
                     Layout
                     Meta
                     Parse
                     Paths_gulcii
                     Reduce
                     Setting
                     Sugar
  ghc-options:       -Wall -threaded -rtsopts
--  ghc-prof-options: -prof -auto-all -caf-all

Source-Repository head
  Type: git
  Location: https://code.mathr.co.uk/gulcii.git
```

```
Source-Repository this
  Type: git
  Tag: v0.3
  Location: https://code.mathr.co.uk/gulcii.git
```

13 lib/bits.gu

```
# Data.Bits(Bits(
#   (.&.), (.|.), xor, complement,
#   bit, setBit, clearBit, complementBit, testBit,
#   shift, shiftL, shiftR, rotate, rotateL, rotateR,
#   bitSize, isSigned ))
# bits
bits@and = zipWith and
bits@or = zipWith or
bits@xor = zipWith xor
bits@not = map not
```

14 lib/bool.gu

```
# boolean
false = \ t f . f
true = \ t f . t
and = \ a b . a b a
or = \ a b . a a b
not = \ a . a false true
```

15 lib/church.gu

```
id = \x . x
const = \x y . x
true = \t f . t
false = \t f . f
and = \a b . a b a
or = \a b . a a b
not = \a . a false true
zero = \s z . z
succ = \n s z . s (n s z)
iszero = \n . n (const false) true
add = \m n . m succ n
mul = \m n . m (add n) zero
exp = \m n . n m
pred = \n s z . n (\g h . h (g s)) (\u . z) (\u . u)
sub = \m n . n pred m
equal = \m n . and (iszero (sub m n)) (iszero (sub n m))
nil = \c n . n
cons = \x xs c n . c x (xs c n)
isnil = \l . l (\x xs . false) true
head = \l . l (\x xs . x) id
tail = \l c n . l (\x xs g . g x (xs c)) (\xs . n) (\x xs . xs)
map = \f l . l (\x xs . cons (f x) xs) nil
sum = \l . l (\x xs . add x xs) zero
length = \l . l (\x xs . succ xs) zero
pair = \a b p . p a b
fst = \p . p (\a b . a)
```

```
snd = \p . p (\a b . b)
nothing = \n j . n
just = \a n j . j a
maybe = \n j m . m n j
left = \a l r . l a
right = \b l r . r b
either = \l r e . e l r
```

16 lib/edinburgh.gu

```
# boolean
false = \ t f . f
true = \ t f . t
and = \ a b . a b a
or = \ a b . a a b
not = \ a . a false true
# function
id = \ a . a
const = \ a b . a
compose = \ bc ab a . bc (ab a)
flip = \ abc b a . abc a b
fix = \ f . (\x . f (x x)) (\x . f (x x))
error = \e . error e
seq = \ a ! \b . b
# list
nil = \ c n . n
cons = \ x xs c n . c x xs
null = \ l . l (\x xs . false) true
head = \ l . l (\x xs . x) (error head)
tail = \ l . l (\x xs . xs) (error tail)
map = \ f l . l (\x xs . cons (f x) (map f xs)) nil
append = \ us vs . us (\x xs . cons x (append xs vs)) vs
filter = \ p l . l (\x xs . p x (cons x) id (filter p xs)) nil
index = \ l n . n (\p . l (\x xs . index xs p) (error index)) (l (\x xs . x) (↙
    ↪ error index))
reverse = \ l . l (\x xs . append (reverse xs) (cons x nil)) nil
foldr = \ f e l . l (\x xs . f x (foldr f e xs)) e
length = foldr (const succ) zero
ands = foldr and true
ors = foldr or false
concat = foldr append nil
all = \f . compose ands (map f)
any = \f . compose ors (map f)
concatMap = \f . compose concat (map f)
composes = foldr compose id
sum = foldr add zero
product = foldr mul (succ zero)
repeat = \x . cons x (repeat x)
cycle = compose concat repeat
take = \n l . n (\p . l (\x xs . cons x (take p xs)) nil) nil
drop = \n l . n (\p . l (\x xs . drop p xs) nil) l
transpose = \l . l (\xs xss . xs (\y ys . cons (cons y (concatMap (take 1) xss))↙
    ↪  (transpose (cons ys (map (drop 1) xss)))) (transpose xss)) nil
iterate = \f x . cons x (iterate f (f x))
last = \l . l (\x xs . xs (\y ys . last xs) x) (error last)
replicate = \n x . take n (repeat x)
# natural
```

```
zero = \ s z . z
succ = \ n s z . s n
infinity = succ infinity
even = \n . n odd true
odd = \n . n even false
add = \ m n . m (\p . succ (add n p)) n
mul = \ m n . m (\p . add n (mul n p)) zero
exp = \ m n . n (\p . mul m (exp m p)) one
sub = \ m n . m (\p . n (sub p) m) zero
equal = \ m n . m (\mm . n (\nn . equal mm nn) false) (n (\nn . false) true)
```

17 lib/either.gu

```
# either
left = \ a l r . l a
right = \ b l r . r b
either = \ ac bc e . e ac bc
```

18 lib/function.gu

```
# function
id = \ a . a
const = \ a b . a
compose = \ bc ab a . bc (ab a)
flip = \ abc b a . abc a b
fix = \ f . f (fix f)
undefined = undefined
error = \e . error e
seq = \ a ! \b . b
```

19 lib/list.gu

```
# module Data.List (
#     (++), head, last, tail, init, null, length, map, reverse,
#     intersperse, intercalate, transpose, subsequences, permutations,
#     foldl, foldl', foldl1, foldl1', foldr, foldr1, concat, concatMap,
#     and, or, any, all, sum, product, maximum, minimum, scanl, scanl1,
#     scanr, scanr1, mapAccumL, mapAccumR, iterate, repeat, replicate,
#     cycle, unfoldr, take, drop, splitAt, takeWhile, dropWhile, span,
#     break, stripPrefix, group, inits, tails, isPrefixOf, isSuffixOf,
#     isInfixOf, elem, notElem, lookup, find, filter, partition, (!!),
#     elemIndex, elemIndices, findIndex, findIndices, zip, zip3, zip4,
#     zip5, zip6, zip7, zipWith, zipWith3, zipWith4, zipWith5, zipWith6,
#     zipWith7, unzip, unzip3, unzip4, unzip5, unzip6, unzip7, lines,
#     words, unlines, unwords, nub, delete, (\\), union, intersect, sort
    ↳ ,
#     insert, nubBy, deleteBy, deleteFirstsBy, unionBy, intersectBy,
#     groupBy, sortBy, insertBy, maximumBy, minimumBy, genericLength,
#     genericTake, genericDrop, genericSplitAt, genericIndex, ↙
    ↳ genericReplicate
#   ) where
# list
nil = \ c n . n
cons = \ x xs c n . c x xs
null = \ l . l (\x xs . false) true
head = \ l . l (\x xs . x) (error head)
```

```
tail = \ l . l (\x xs . xs) (error tail)
map = \ f l . l (\x xs . cons (f x) (map f xs)) nil
append = \ us vs . us (\x xs . cons x (append xs vs)) vs
filter = \ p l . l (\x xs . p x (cons x) id (filter p xs)) nil
index = \ l n . n (\p . l (\x xs . index xs p) (error index)) (l (\x xs . x) (error index))
reverse = \ l . l (\x xs . append (reverse xs) (cons x nil)) nil
foldr = \ f e l . l (\x xs . f x (foldr f e xs)) e
length = foldr (const succ) zero
ands = foldr and true
ors = foldr or false
concat = foldr append nil
all = \f . compose ands (map f)
any = \f . compose ors (map f)
concatMap = \f . compose concat (map f)
composes = foldr compose id
sum = foldr add zero
product = foldr mul (succ zero)
repeat = \x . cons x (repeat x)
cycle = compose concat repeat
zipWith = \f l r . l (\x xs . r (\y ys . cons (f x y) (zipWith f xs ys)) nil) nil
partition = \p l . l (\x xs . p x first second (cons x) (partition p xs)) (pair nil nil)
partitionEithers = \l . l (\e es . e (compose first cons) (compose second cons) (partitionEithers es)) (pair nil nil)
take = \n l . n (\p . l (\x xs . cons x (take p xs)) nil) nil
drop = \n l . n (\p . l (\x xs . drop p xs) nil) l
transpose = \l . l (\xs xss . xs (\y ys . cons (cons y (concatMap (take 1) xss)) (transpose (cons ys (map (drop 1) xss)))) (transpose xss)) nil
catMaybes = \l . l (\x xs . maybe id cons x (catMaybes xs)) nil
iterate = \f . \x ! cons x (iterate f (f x))
last = \l . l (\x xs . xs (\y ys . last xs) x) (error last)
replicate = \n x . take n (repeat x)
rotate = \n bs . splitAt n bs \xs ys . append ys xs
```

20 lib/maybe.gu

```
# module Data.Maybe (
#     Maybe(Nothing, Just), maybe, isJust, isNothing, fromJust, fromMaybe,
#     listToMaybe, maybeToList, catMaybes, mapMaybe
#   ) where instance Monad Functor MonadPlus Eq Ord Read Show
# maybe
nothing = \ j n . n
just = \ a j n . j a
maybe = \ b ab m . m ab b
```

21 lib/natural.gu

```
# natural
zero = \ s z . z
succ = \ n s z . s n
infinity = succ infinity
even = \n . n odd true
odd = \n . n even false
add = \ m n . m (\p . succ (add n p)) n
```

```
mul = \ m n . m (\p . add n (mul n p)) zero
sub = \ m n . m (\p . n (sub p) m) zero
equal = \ m n . m (\mm . n (\nn . equal mm nn) false) (n (\nn . false) true)
lessThan = \x y . x (\xx . y (\yy . lessThan xx yy) false) (y (\yy . true) false↙
    ↳ )
```

22 lib/pair.gu

```
# pair
pair = \ a b p . p a b
fst = \ p . p \ a b . a
snd = \ p . p \ a b . b
curry = \ f a b . f (pair a b)
uncurry = \ f p . f (fst p) (snd p)
swap = \p . pair (snd p) (fst p)
first = \f p . pair (f (fst p)) (snd p)
second = \ f p . pair (fst p) (f (snd p))
```

23 lib/prelude.gu

```
:load bits
:load bool
:load either
:load function
:load list
:load maybe
:load natural
:load pair
```

24 LICENSE

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

```
 Copyright (C) 1989, 1991 Free Software Foundation, Inc.,
 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                        Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.  This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it.  (Some other Free Software Foundation software is covered by
the GNU Lesser General Public License instead.)  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
```

in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the

notices that refer to this License and to the absence of any warranty;
and give any other recipients of the Program a copy of this License
along with the Program.

You may charge a fee for the physical act of transferring a copy, and
you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion
of it, thus forming a work based on the Program, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices
stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in
whole or in part contains or is derived from the Program or any
part thereof, to be licensed as a whole at no charge to all third
parties under the terms of this License.

c) If the modified program normally reads commands interactively
when run, you must cause it, when started running for such
interactive use in the most ordinary way, to print or display an
announcement including an appropriate copyright notice and a
notice that there is no warranty (or else, saying that you provide
a warranty) and that users may redistribute the program under
these conditions, and telling the user how to view a copy of this
License. (Exception: if the Program itself is interactive but
does not normally print such an announcement, your work based on
the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If
identifiable sections of that work are not derived from the Program,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works. But when you
distribute the same sections as part of a whole which is a work based
on the Program, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable
source code, which must be distributed under the terms of Sections

1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes

make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the program's name and a brief idea of what it does.>
> Copyright (C) <year> <name of author>
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

```
If the program is interactive, make it output a short notice like this
when it starts in an interactive mode:

    Gnomovision version 69, Copyright (C) year name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, the commands you use may
be called something other than `show w' and `show c'; they could even be
mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the program, if
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  `Gnomovision' (which makes passes at compilers) written by James Hacker.

  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice

This General Public License does not permit incorporating your program into
proprietary programs.  If your program is a subroutine library, you may
consider it more useful to permit linking proprietary applications with the
library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.
```

25 README.md

```
GULCII
======

GULCII is an untyped lambda calculus interpreter supporting interactive
modification of a running program with graphical display of graph reduction.


Syntax
------

Lambda calculus terms with some sugar coating:

    term ::= variable                       -- free or bound
           | '\' variable+ strategy term    -- lambda abstraction
           | term+                          -- application
           | '(' term ')'
           | integer
           | list
    variable ::= [a-z][A-Za-z0-9]*
    strategy = '.' | '!' | '?'              -- lazy | strict | copy
    integer ::= [0-9]+                      -- uses Scott-encoding
    list ::= '[' (term (, T)*)? ']'         -- uses Scott-encoding

There are three variants of lambda abstraction: lazy, strict, and copy:
```

```
    \x . f x x    -- x is evaluated lazily with sharing
    \x ! f x x    -- x is evaluated strictly and shared
    \x ? f x x    -- x is copied before any evaluation
```

There is additional syntax sugar for natural numbers and lists, using <http://en.wikipedia.org/wiki/Mogensen%E2%80%93Scott_encoding#Scott_encoding>:

```
    [0,1,2,3]
```

There is a small standard library based loosely around the Haskell Prelude:

```
    :load prelude
    :browse
```

If you define a term using free variables, they can be modified while the program is running, but sharing is lost. If you define a term as a fixed point (perhaps with Y-combinator) then sharing works (to some extent), but you can't modify the code while it is running any more.

Meta Commands

To exit type:

```
    :quit
```

Entering a term evaluates it.

Terms can be bound to names, stored in a global dictionary:

```
    foo = bar
```

The global dictionary can be listed or wiped clean:

```
    :browse
    :clear
```

Installed files can be loaded:

```
    :load church
```

Machine-readable node statistics are output to stdout, as well as when free variables are instantiated by looking up their definitions. These are intended to be used for sonification, for example with Pure-data:

```
    pd extra/gulcii.pd &
    sleep 5
    gulcii | pdsend 8765
```

The sonification is controlled by two commands:

```
    :start
    :stop
```

Settings

```
--------

There are some runtime adjustable settings:

    :get NewsOnTop
    :set NewsOnTop
    :unset NewsOnTop
    :toggle NewsOnTop

Where NewsOnTop is a setting.  Settings include:

    TraceEvaluation
    CollectGarbage
    RealTimeDelay
    RealTimeAcceleration
    RetryIrreducible
    EmitStatistics
    EmitRebindings
    EchoToStdOut
    EchoToGUI
    SaveImages
    NewsOnTop
```

26 Setup.hs

```
import Distribution.Simple
main = defaultMain
```

27 src/Bruijn.hs

```
{-
    gulcii -- graphical untyped lambda calculus interpreter
    Copyright (C) 2011, 2013, 2017  Claude Heiland-Allen

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
-}

module Bruijn (Term(..), bruijn, freeVariablesIn) where

import Data.Maybe (listToMaybe)
import Data.Set (Set, empty, singleton, union)

import qualified Lambda as U
import Evaluation (Strategy)
```

```
data Term
  = Free String
  | Bound0
  | Scope Term
  | Lambda Strategy Term
  | Apply Term Term
  deriving (Read, Show, Eq, Ord)

bruijn :: U.Term -> Term
bruijn = bruijn' []

bruijn' :: [String] -> U.Term -> Term
bruijn' m (U.Apply s t) = Apply (bruijn' m s) (bruijn' m t)
bruijn' m (U.Lambda k v t) = Lambda k (bruijn' (v : m) t)
bruijn' m (U.Variable v) =
  case genericElemIndex v m of
    Nothing -> Free v
    Just i -> applyN i Scope Bound0

applyN :: Integer -> (a -> a) -> a -> a
applyN n f
  | n == 0 = id
  | otherwise = f . applyN (n - 1) f

freeVariablesIn :: Term -> Set String
freeVariablesIn (Free v) = singleton v
freeVariablesIn Bound0 = empty
freeVariablesIn (Scope t) = freeVariablesIn t
freeVariablesIn (Lambda _ t) = freeVariablesIn t
freeVariablesIn (Apply s t) = freeVariablesIn s `union` freeVariablesIn t

{-
Generic list functions.
-}

genericElemIndex :: (Eq a, Enum b, Num b) => a -> [a] -> Maybe b
genericElemIndex x = genericFindIndex (x ==)

genericFindIndex :: (Enum b, Num b) => (a -> Bool) -> [a] -> Maybe b
genericFindIndex p = listToMaybe . genericFindIndices p

genericFindIndices :: (Enum b, Num b) => (a -> Bool) -> [a] -> [b]
genericFindIndices p xs = [ i | (x, i) <- zip xs [0 ..], p x]
```

28 src/Command.hs

```
{-
    gulcii -- graphical untyped lambda calculus interpreter
    Copyright (C) 2011, 2013, 2017  Claude Heiland-Allen

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
```

```
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
-}

module Command (Command(..), parse) where

import Control.Applicative ((<$>), (<*>), (<$), (<*), (<|>))

import qualified Sugar as S
import qualified Meta as M
import Parse

data Command = Define String S.Term | Evaluate S.Term | Meta M.Meta
  deriving (Read, Show, Eq, Ord)

parse :: Parser String Command
parse =  (Define   <$> name <* sym "=" <*> S.parse)
     <|> (Evaluate <$> S.parse)
     <|> (Meta     <$  sym ":" <*> M.parse)
```

29 src/Draw.hs

```
{-
    gulcii -- graphical untyped lambda calculus interpreter
    Copyright (C) 2011, 2013, 2017  Claude Heiland-Allen

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
-}

module Draw (draw) where

import qualified Data.Map.Strict as M
import Data.Map.Strict (Map)

import Graphics.Rendering.Cairo hiding (x, y)

import qualified Layout as L
import Evaluation (Strategy(..))

type RGB = (Double, Double, Double)
```

```
colour :: L.Term -> RGB
colour (L.Free     _ _) = (0, 0.5, 1)
colour (L.Bound0     _) = (0.5, 0, 1)
colour (L.Scope    _ _) = (1, 1, 0)
colour (L.Lambda Strict _ _) = (1, 0, 0)
colour (L.Lambda Lazy   _ _) = (1, 0, 0.5)
colour (L.Lambda Copy   _ _) = (1, 0, 1)
colour (L.Apply   _ _ _) = (1, 0.5, 0)
colour (L.RefInst _ _ _) = (0.5, 1, 0)
colour (L.Reference _ _) = (0, 1, 0.5)

circle :: L.Coords -> RGB -> Render ()
circle (x, y) (r, g, b) = do
  save
  translate (fromIntegral x) (fromIntegral y)
  arc 0 0 0.365 0 (2*pi)
  setSourceRGB r g b
  fillPreserve
  setSourceRGB 0 0 0
  stroke
  restore

line :: L.Coords -> L.Coords -> Render ()
line (x, y) (x', y') = do
  save
  moveTo (fromIntegral x ) (fromIntegral y )
  lineTo (fromIntegral x') (fromIntegral y')
  stroke
  restore

draw :: Double -> Double -> L.Layout -> Render ()
draw ww0 hh0 (L.Layout t w h ps) = do
  save
  translate dx dy
  scale s s
  translate 0.5 0.5
  setLineWidth 0.1
  setSourceRGB 0 0 0
  drawLinks ps t
  drawNodes t
  setLineCap LineCapRound
  setSourceRGB 0.5 0.5 0.5
  drawVLinks [] t
  setFontSize (6 / sqrt s)
  translate 0 0.5
  drawNames t
  setSourceRGB 0 0 0
  fillPreserve
  setLineWidth (0.2 / sqrt s)
  setSourceRGB 1 1 1
  stroke
  restore
  where
    s = if fromIntegral w * hh <= fromIntegral h * ww then hh / fromIntegral h ↙
        ↳ else ww / fromIntegral w
    ww = ww0 - 128
```

```
      hh = hh0 - 64
      dx = (ww0 - s * fromIntegral w) / 2
      dy = (hh0 - s * fromIntegral h) / 2

drawNames :: L.Term -> Render ()
drawNames (L.Free s (x,y)) = do
  e <- textExtents s
  moveTo (fromIntegral x - textExtentsWidth e / 2) (fromIntegral y)
  textPath s
drawNames (L.Bound0 _) = return ()
drawNames (L.Scope t _) = drawNames t
drawNames (L.Lambda _ t _) = drawNames t
drawNames (L.Apply a b _) = drawNames a >> drawNames b
drawNames (L.RefInst _ t _) = drawNames t
drawNames (L.Reference _ _) = return ()

drawLinks :: Map Integer L.Coords -> L.Term -> Render ()
drawLinks _  (L.Free _ _) =
  return ()
drawLinks _  (L.Bound0 _) =
  return ()
drawLinks ps (L.Scope t xy) =
  let x'y' = L.coordinates t
  in  line xy x'y' >> drawLinks ps t
drawLinks ps (L.Lambda _ t xy) =
  let x'y' = L.coordinates t
  in  line xy x'y' >> drawLinks ps t
drawLinks ps (L.Apply a b xy) =
  let axay = L.coordinates a
      bxby = L.coordinates b
  in  line xy axay >> line xy bxby >> drawLinks ps a >> drawLinks ps b
drawLinks ps (L.RefInst _ t xy) =
  let x'y' = L.coordinates t
  in  line xy x'y' >> drawLinks ps t
drawLinks ps (L.Reference p xy) =
  let Just x'y' = M.lookup p ps
  in  line xy x'y'

drawVLinks :: [L.Coords] -> L.Term -> Render ()
drawVLinks ls (L.Bound0 xy) = case ls of
  [] -> return () -- should be error?
  x'y':_ -> line xy x'y'
drawVLinks ls (L.Scope t _) = drawVLinks (drop 1 ls) t
drawVLinks ls (L.Lambda _ t xy) = drawVLinks (xy : ls) t
drawVLinks ls (L.Apply  s t _) = drawVLinks ls s >> drawVLinks ls t
drawVLinks ls (L.RefInst _ t _) = drawVLinks ls t
drawVLinks _ _ = return ()

drawNodes :: L.Term -> Render ()
drawNodes n@(L.Free      _ _) = drawNode n
drawNodes n@(L.Bound0      _) = drawNode n
drawNodes n@(L.Scope     t _) = drawNode n >> drawNodes t
drawNodes n@(L.Lambda  _ t _) = drawNode n >> drawNodes t
drawNodes n@(L.Apply   a b _) = drawNode n >> drawNodes a >> drawNodes b
drawNodes n@(L.RefInst _ t _) = drawNode n >> drawNodes t
drawNodes n@(L.Reference _ _) = drawNode n
```

```
drawNode :: L.Term -> Render ()
drawNode n = circle (L.coordinates n) (colour n)
```

30 src/Evaluation.hs

```
{-
    gulcii -- graphical untyped lambda calculus interpreter
    Copyright (C) 2011, 2013  Claude Heiland-Allen

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
-}

module Evaluation (Strategy(..)) where

data Strategy = Lazy | Strict | Copy
  deriving (Read, Show, Eq, Ord)
```

31 src/GC.hs

```
{-
    gulcii -- graphical untyped lambda calculus interpreter
    Copyright (C) 2011, 2013, 2017  Claude Heiland-Allen

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
-}

module GC (gc) where

import Graph

import Data.Map as Map

```

```
gc
  :: References -> Term
  -> (References, Term)

gc refs term
  = let counts = refCount refs term Map.empty
    in ( Map.fromList
           [ (r, compact counts refs (refs Map.! r))
           | (r, n) <- Map.toList counts
           , n > 1 ]
       , compact counts refs term )


type Count = Integer

refCount
  :: References -> Term
  -> Map Integer Count -> Map Integer Count

compact
  :: Map Integer Count -> References
  -> Term -> Term


refCount refs (Scope a) counts
  = refCount refs a counts

refCount refs (Lambda _ a) counts
  = refCount refs a counts

refCount refs (Apply a b) counts
  = refCount refs a (refCount refs b counts)

refCount refs (Reference r) counts
  = case r `Map.lookup` counts of
      Just n ->
        Map.insert r (n + 1) counts
      Nothing ->
        refCount refs (refs Map.! r)
                (Map.insert r 1 counts)

refCount _ _ counts = counts


compact counts refs (Scope term)
  = Scope (compact counts refs term)

compact counts refs (Lambda strat term)
  = Lambda strat (compact counts refs term)

compact counts refs (Apply a b)
  = Apply (compact counts refs a)
          (compact counts refs b)

compact counts refs term@(Reference r)
  = case r `Map.lookup` counts of
      Just 1 -> compact counts refs (refs Map.! r)
```

```
      _ -> term

compact _ _ term = term
```

32 src/Graph.hs

```
{-
    gulcii -- graphical untyped lambda calculus interpreter
    Copyright (C) 2011, 2013, 2017  Claude Heiland-Allen

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
-}

module Graph (Term(..), Definitions, References, graph, pretty) where

import Data.Map.Strict (Map)

import qualified Bruijn as B
import Evaluation (Strategy(..))

data Term
  = Free !String
  | Bound0
  | Scope !Term
  | Lambda !Strategy !Term
  | Apply !Term !Term
  | Reference !Integer
  deriving (Read, Show, Eq, Ord)

pretty :: Term -> String
pretty = concat . pretty'

pretty' :: Term -> [String]
pretty' (Free s) = [s]
pretty' (Bound0) = ["0"]
pretty' (Scope t) = ["S"] ++ pretty' t
pretty' (Reference i) = ['#':show i]
pretty' (Lambda k t) = ["(", "\\", pretty'' k] ++ pretty' t ++ [")"]
pretty' (Apply s t) = ["("] ++ pretty' s ++ [" "] ++ pretty' t ++ [")"]

pretty'' :: Strategy -> String
pretty'' Strict = "!"
pretty'' Lazy = "."
pretty'' Copy = "?"
```

```
type Definitions = Map String Term
type References = Map Integer Term

graph :: B.Term -> Term
graph (B.Free v) = Free v
graph (B.Bound0) = Bound0
graph (B.Scope t) = Scope (graph t)
graph (B.Lambda k t) = Lambda k (graph t)
graph (B.Apply s t) = Apply (graph s) (graph t)
```

33 src/Lambda.hs

```
{-
    gulcii -- graphical untyped lambda calculus interpreter
    Copyright (C) 2011, 2013, 2017  Claude Heiland-Allen

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
-}

module Lambda (Term(..), pretty, isFreeIn, variablesIn, freeVariablesIn) where

import Data.List (nub)

import Evaluation (Strategy(..))

{-
Untyped lambda calculus terms.
-}

data Term
  = Variable String
  | Lambda Strategy String Term
  | Apply Term Term
  deriving (Read, Show, Eq, Ord)

{-
Pretty-print a term.
-}

pretty :: Term -> String
pretty = unwords . pretty'

pretty' :: Term -> [String]
pretty' (Variable v) = [v]
pretty' (Lambda k v t) = ["(", "\\", v, pretty'' k] ++ pretty' t ++ [")"]
```

```
pretty' (Apply  s t) = ["("] ++ pretty' s ++ pretty' t ++ [")"]

pretty'' :: Strategy -> String
pretty'' Strict = "!"
pretty'' Lazy = "."
pretty'' Copy = "?"

{-
Check if a variable occurs free in a term.
-}

isFreeIn :: String -> Term -> Bool
isFreeIn n (Variable v)  = n == v
isFreeIn n (Lambda _ v t)  = if n == v then False else n `isFreeIn` t
isFreeIn n (Apply t t')  = n `isFreeIn` t || n `isFreeIn` t'

{-
Get all variable names defined or referenced by a term.
-}

variablesIn :: Term -> [String]
variablesIn (Variable v)  = [v]
variablesIn (Lambda _ v t)  = nub $ v : variablesIn t
variablesIn (Apply t t')  = nub $ variablesIn t ++ variablesIn t'

{-
Get all free variables referenced by a term.
-}

freeVariablesIn :: Term -> [String]
freeVariablesIn t = filter (`isFreeIn` t) (variablesIn t)
```

34 src/Layout.hs

```
{-
    gulcii -- graphical untyped lambda calculus interpreter
    Copyright (C) 2011, 2013, 2017  Claude Heiland-Allen

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
-}

module Layout (Term(..), Coords, Layout(..), layout, coordinates, Counts(..), ↙
    ↳ counts) where

import qualified Data.Map.Strict as M
```

```
import Data.Map.Strict (Map)

import qualified Graph as G
import Evaluation (Strategy(..))

type Coords = (Integer, Integer)

data Term
  = Free String Coords
  | Bound0 Coords
  | Scope Term Coords
  | Lambda Strategy Term Coords
  | Apply Term Term Coords
  | RefInst Integer Term Coords
  | Reference Integer Coords
  deriving (Read, Show, Eq, Ord)

data Layout = Layout Term Integer Integer (Map Integer Coords)
  deriving (Read, Show, Eq, Ord)

coordinates :: Term -> Coords
coordinates (Free      _ xy) = xy
coordinates (Bound0      xy) = xy
coordinates (Scope     _ xy) = xy
coordinates (Lambda  _ _ xy) = xy
coordinates (Apply   _ _ xy) = xy
coordinates (RefInst _ _ xy) = xy
coordinates (Reference _ xy) = xy

layout :: G.Term -> G.References -> Layout
layout = layout' (0, 0) M.empty

layout' :: Coords -> Map Integer Coords -> G.Term -> G.References -> Layout
layout' xy ps (G.Free v) _ =
  Layout (Free v xy) 1 1 ps
layout' xy ps (G.Bound0) _ =
  Layout (Bound0 xy) 1 1 ps
layout' (x, y) ps (G.Scope t) g =
  let Layout lt w h ps' = layout' (x, y + 1) ps t g
      (px, _) = coordinates lt
  in  Layout (Scope lt (px, y)) w (h + 1) ps'
layout' (x,y) ps (G.Lambda k t) g =
  let Layout lt w h ps' = layout' (x, y + 1) ps t g
      (px, _) = coordinates lt
  in  Layout (Lambda k lt (px, y)) w (h + 1) ps'
layout' (x,y) ps (G.Apply a b) g =
  let Layout la aw ah psa = layout' (x, y + 1) ps a g
      Layout lb bw bh psb = layout' (x + aw + 1, y + 1) psa b g
  in  Layout (Apply la lb (x + aw, y)) (1 + aw + bw) (1 + (ah `max` bh)) psb
layout' xy@(x,y) ps (G.Reference p) g =
  if p `M.member` ps
    then Layout (Reference p xy) 1 1 ps
    else case M.lookup p g of
      Nothing -> error $ "layout': bad pointer: " ++ show p
      Just t  ->
        let Layout lt w h pst = layout' (x, y + 1) ps t g
            (px, py) = coordinates lt
```

```
80          in  Layout (RefInst p lt (px, y)) w (1 + h) (M.insert p (px, py) pst)

data Counts = Counts
  { nFree, nBound0, nScope, nLambda, nApply, nRefInst, nReference :: !Int }

85 instance Monoid Counts where
  mempty = Counts 0 0 0 0 0 0 0
  mappend c d = Counts
                  { nFree = nFree c + nFree d
                  , nBound0 = nBound0 c + nBound0 d
90                , nScope = nScope c + nScope d
                  , nLambda = nLambda c + nLambda d
                  , nApply = nApply c + nApply d
                  , nRefInst = nRefInst c + nRefInst d
                  , nReference = nReference c + nReference d
95                }

counts :: Term -> Counts
counts (Free _ _) = mempty{ nFree = 1 }
counts (Bound0 _) = mempty{ nBound0 = 1 }
100 counts (Scope t _) =  let c = counts t in c{ nScope = nScope c + 1 }
counts (Lambda _ t _) = let c = counts t in c{ nLambda = nLambda c + 1 }
counts (Apply a b _) = let c = counts a ; d = counts b in mappend c d
counts (RefInst _ t _) = let c = counts t in c{ nRefInst = nRefInst c + 1 }
counts (Reference _ _) = mempty{ nReference = 1 }
```

35 src/Main.hs

```
{-
    gulcii -- graphical untyped lambda calculus interpreter
    Copyright (C) 2011, 2013, 2017  Claude Heiland-Allen

5   This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

10  This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

15  You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
-}

20 module Main (main) where

import Control.Applicative ((<$>))
import Control.Concurrent (forkIO, killThread, threadDelay, newChan, readChan, ↙
    ↳ writeChan)
import Control.Monad (forever, when, unless, forM_)
25 import qualified Data.Map.Strict as M
import Data.Maybe (isJust)
import Data.IORef (IORef, newIORef, readIORef, writeIORef, atomicModifyIORef)
import System.IO (hSetBuffering, BufferMode(LineBuffering), stdout)
```

```
import System.IO.Error (catchIOError)
import System.FilePath ((</>), (<.>))
import Graphics.UI.Gtk hiding (Meta, Settings)
import Graphics.Rendering.Cairo hiding (width, height)

import Paths_gulcii (getDataFileName)

import qualified Command as C
import qualified Meta as M
import Setting (Settings)
import qualified Setting
import qualified Sugar as S
import qualified Bruijn as B
import qualified Graph as G
import qualified GC as GC
import qualified Reduce as R
import qualified Layout as L
import qualified Draw as D
import qualified Parse as P

data Interpret = Fail | Skip | Define String G.Term | Pure G.Term | Meta M.Meta
  deriving (Read, Show, Eq, Ord)

interpret :: String -> Interpret
interpret l =
  case P.unP C.parse `fmap` P.tokenize (P.decomment l) of
    Just ((C.Define d sterm, []):_) ->
      case S.desugar sterm of
        Just term -> Define d . G.graph . B.bruijn $ term
        _ -> Fail
    Just ((C.Evaluate sterm, []):_) ->
      case S.desugar sterm of
        Just term -> Pure . G.graph . B.bruijn $ term
        _ -> Fail
    Just ((C.Meta m, []):_) -> Meta m
    Just [] -> Skip
    _ -> Fail

main :: IO ()
main = do
  args <- initGUI
  let (width, height, fullscreen) = case args of
        [w,h,"fullscreen"] -> (read w, read h, True)
        [w,h] -> (read w, read h, False)
        _ -> (1280, 720, False)
  envR <- newIORef M.empty
  lRef <- newIORef Nothing
  evalR <- newIORef Nothing
  settingsR <- newIORef Setting.defaults
  outC <- newChan
  let out = writeChan outC
  win <- windowNew
  _ <- onDestroy win mainQuit
  windowSetDefaultSize win width height
  when fullscreen $ do
    windowSetGeometryHints win (Nothing `asTypeOf` Just win)
      (Just (width, height)) (Just (width, height)) Nothing Nothing Nothing
```

```
    set win [ windowDecorated := False ]
    windowSetKeepAbove win True
    windowMove win 0 0
  vb <- vBoxNew False 0
  hb <- hPanedNew
  tt <- textTagTableNew
  tagInputRem    <- textTagNew Nothing
  tagInputDef    <- textTagNew Nothing
  tagInputPure   <- textTagNew Nothing
  tagInputRun    <- textTagNew Nothing
  tagInputMeta   <- textTagNew Nothing
  tagOutput      <- textTagNew Nothing
  tagOutputMeta <- textTagNew Nothing
  tagError       <- textTagNew Nothing
  set tagInputRem    [ textTagForeground := "cyan"    ]
  set tagInputDef    [ textTagForeground := "green"   ]
  set tagInputPure   [ textTagForeground := "yellow"  ]
  set tagInputRun    [ textTagForeground := "orange"  ]
  set tagInputMeta   [ textTagForeground := "blue"    ]
  set tagOutput      [ textTagForeground := "magenta" ]
  set tagOutputMeta [ textTagForeground := "pink"    ]
  set tagError       [ textTagForeground := "red"     ]
  textTagTableAdd tt tagInputRem
  textTagTableAdd tt tagInputDef
  textTagTableAdd tt tagInputPure
  textTagTableAdd tt tagInputRun
  textTagTableAdd tt tagInputMeta
  textTagTableAdd tt tagOutput
  textTagTableAdd tt tagOutputMeta
  textTagTableAdd tt tagError
  tf <- textBufferNew (Just tt)
  tb <- textBufferNew (Just tt)
  tv <- textViewNewWithBuffer tf
  mk <- textMarkNew Nothing False
  it <- textBufferGetIterAtOffset tf (-1)
  textBufferAddMark tf mk it
  textViewSetEditable tv False
  textViewSetWrapMode tv WrapWord
  da <- drawingAreaNew
  _ <- da `on` exposeEvent $ do
    dw <- eventWindow
    liftIO $ do
      ml <- atomicModifyIORef lRef (\m -> (Nothing, m))
      case ml of
        Nothing -> return ()
        Just l -> do
          (ww, hh) <- drawableGetSize dw
          renderWithDrawable dw $ do
            D.draw (fromIntegral ww) (fromIntegral hh) l
    return True
  en <- entryNew
  entrySetWidthChars en 25
  font <- fontDescriptionFromString "Monospaced 18"
  widgetModifyFont tv (Just font)
  widgetModifyFont en (Just font)
  sw <- scrolledWindowNew Nothing Nothing
  scrolledWindowSetPolicy sw PolicyAutomatic PolicyAlways
```

```
    containerAdd sw tv
    al <- alignmentNew 1 0 1 1
    set al [ containerChild := da ]
    if Setting.newsOnTop Setting.defaults
      then do
        boxPackStart vb en PackNatural 0
        boxPackStart vb sw PackGrow 0
      else do
        boxPackStart vb sw PackGrow 0
        boxPackStart vb en PackNatural 0
    panedPack1 hb vb False True
    panedPack2 hb al True True
    set win [ containerChild := hb ]
    containerSetFocusChain vb [toWidget en]
    let scrollDown = do
          textViewScrollToMark tv mk 0 Nothing
        addText tag txt = do
          newsOnTop <- Setting.newsOnTop <$> readIORef settingsR
          start' <- textBufferGetIterAtOffset tb 0
          end' <- textBufferGetIterAtOffset tb (-1)
          textBufferDelete tb start' end'
          textBufferInsert tb start' (unlines [txt])
          start <- textBufferGetIterAtOffset tb 0
          end <- textBufferGetIterAtOffset tb (-1)
          textBufferApplyTag tb tag start end
          pos <- textBufferGetIterAtOffset tf (if newsOnTop then 0 else -1)
          textBufferInsertRange tf pos start end
          pos' <- textBufferGetIterAtOffset tf (if newsOnTop then 0 else -1)
          textBufferMoveMark tf mk pos'
    _ <- en `onEntryActivate` do
      let exec echo txt =
            case interpret txt of
              Fail -> addText tagError txt
              Skip -> when echo $ do
                addText tagInputRem txt
                entrySetText en ""
              Define def term -> do
                when echo $ do
                  addText tagInputDef txt
                  entrySetText en ""
                atomicModifyIORef envR (\defs -> (M.insert def term defs, ()))
              Pure term -> do
                when echo $ do
                  addText tagInputPure txt
                  entrySetText en ""
                mtid <- readIORef evalR
                case mtid of
                  Nothing -> return ()
                  Just tid -> killThread tid
                tid <- forkIO $ evaluator "gulcii-" 0 settingsR 500000 lRef out ↙
                    ↳ envR M.empty term goPure
                writeIORef evalR (Just tid)
              Meta M.Start -> do
                when echo $ do
                  addText tagInputMeta txt
                  entrySetText en ""
                _ <- forkIO $ do
```

```
      out "start;"
    return ()
  Meta M.Stop -> do
    when echo $ do
      addText tagInputMeta txt
      entrySetText en ""
    _ <- forkIO $ do
      out "stop;"
    return ()
  Meta M.Quit -> do
    _ <- forkIO $ do
      out "quit;"
      postGUISync mainQuit
    return ()
  Meta M.Clear -> do
    when echo $ do
      addText tagInputMeta txt
      entrySetText en ""
    atomicModifyIORef envR (\_ -> (M.empty, ()))
  Meta M.Browse -> do
    when echo $ do
      addText tagInputMeta txt
      entrySetText en ""
    defs <- readIORef envR
    addText tagOutputMeta(unwords (M.keys defs))
  Meta (M.Load f) -> do
    when echo $ do
      addText tagInputMeta txt
    f' <- getDataFileName ("lib" </> f <.> "gu")
    s <- (fmap Right (readFile f')) `catchIOError` (return . Left . ↙
      ↳ show)
    case s of
      Right t -> do
        when echo $ do
          entrySetText en ""
        mapM_ (exec False) (lines t)
      Left e ->
        addText tagError e
  Meta (M.Get s) -> do
    when echo $ do
      addText tagInputMeta txt
      entrySetText en ""
    addText tagOutputMeta . ((show s ++ " = ") ++) . show . Setting.↙
      ↳ get s =<< readIORef settingsR
  Meta (M.Set s) -> do
    when echo $ do
      addText tagInputMeta txt
      entrySetText en ""
    atomicModifyIORef settingsR $ (\ss -> (Setting.set s True ss, ()))
    addText tagOutputMeta . ((show s ++ " = ") ++) . show . Setting.↙
      ↳ get s =<< readIORef settingsR
  Meta (M.UnSet s) -> do
    when echo $ do
      addText tagInputMeta txt
      entrySetText en ""
    atomicModifyIORef settingsR $ (\ss -> (Setting.set s False ss, ())↙
      ↳ )
```

```
                addText tagOutputMeta . ((show s ++ " = ") ++) . show . Setting.↙
                    ↳ get s =<< readIORef settingsR
              Meta (M.Toggle s) -> do
                when echo $ do
255               addText tagInputMeta txt
                  entrySetText en ""
                atomicModifyIORef settingsR $ (\ss -> (Setting.set s (not (Setting↙
                    ↳ .get s ss)) ss, ()))
                addText tagOutputMeta . ((show s ++ " = ") ++) . show . Setting.↙
                    ↳ get s =<< readIORef settingsR
      txt <- entryGetText en
260   exec True txt
      scrollDown

    _ <- forkIO $ do
         hSetBuffering stdout LineBuffering
265      forever $ do
           settings <- readIORef settingsR
           s <- readChan outC
           when (Setting.echoToStdOut settings) $ putStrLn s
           when (Setting.echoToGUI    settings) $ postGUIAsync (addText tagOutput↙
               ↳  s >> scrollDown)
270
    _ <- flip timeoutAdd 10 $ do
      ml <- atomicModifyIORef lRef (\m -> (m, m))
      when (isJust ml) $ widgetQueueDraw da
      return True
275 widgetShowAll win
    mainGUI

type Go = G.References -> G.Term -> IO (G.Term, G.References)

280 goPure :: Go
goPure refs term = return (term, refs)

evaluator :: String -> Int -> IORef Settings -> Int -> IORef (Maybe L.Layout) ->↙
    ↳  (String -> IO ()) -> IORef G.Definitions -> G.References -> G.Term -> Go ↙
    ↳ -> IO ()
evaluator pngPrefix frame settingsR tick layout out defsR refs term go = do
285 defs <- readIORef defsR
  settings <- readIORef settingsR
  when (Setting.traceEvaluation settings) $ do
    out $ "  " ++ G.pretty term
    unless (M.null refs) $ do
290   out "    where"
      forM_ (M.toList refs) $ \(r, t) -> do
        out $ "      #" ++ show r ++ " = " ++ G.pretty t
  let (refs1, term1)
        | Setting.collectGarbage settings = GC.gc refs term
295     | otherwise = (refs, term)
      collectedGarbage = term1 /= term || refs1 /= refs
  when (Setting.traceEvaluation settings && collectedGarbage) $ do
    out "$\\to$      {- collect garbage -}"
    out $ "  " ++ G.pretty term1
300 unless (M.null refs1) $ do
      out "    where"
      forM_ (M.toList refs1) $ \(r, t) -> do
```

```
        out $ "    #" ++ show r ++ " = " ++ G.pretty t
    (term0, refs0) <- go refs1 term1
    (l, L.Counts c1 c2 c3 c4 c5 c6 c7) <- atomicModifyIORef layout $ \_ ->
      let l@(L.Layout lt _ _ _) = L.layout term0 refs0
      in  (Just l, (l, L.counts lt))
    when (Setting.saveImages settings) $ do
      withImageSurface FormatRGB24 1920 1080 $ \surface -> do
        renderWith surface $ do
          let g = 23 / 255
          setSourceRGB g g g
          paint
          D.draw 1920 1080 l
        surfaceWriteToPNG surface (pngPrefix ++ show frame ++ ".png")
    when (Setting.emitStatistics settings) $ do
      out $ "statistics " ++ unwords (map show [c1,c2,c3,c4,c5,c6,c7]) ++ ";"
    when (Setting.realTimeDelay settings) $ do
      threadDelay tick
    let tick'
          | Setting.realTimeAcceleration settings = ceiling (fromIntegral tick * ↙
              ↪ 0.97 + 0.03 * 42000 :: Double)
          | otherwise = tick
    case R.reduce defs refs0 term0 of
      Nothing -> do
        when (Setting.traceEvaluation settings) $ do
          out "$\\to$      {- in normal form -}"
          out "  $\\qedsymbol$"
        when (Setting.retryIrreducible settings) $ do
          evaluator pngPrefix (frame + 1) settingsR tick layout out defsR refs0 ↙
              ↪ term0 go
      Just (reason, (refs', term')) -> do
        when (Setting.traceEvaluation settings) $ do
          out $ "$\\to$      {- " ++ (case reason of
              R.Beta -> "beta reduce"
              R.RefInst -> "instantiate reference"
              R.Rebound var -> "definition of \"" ++ var ++ "\""
              R.Extrude -> "scope extrude"
            ) ++ " -}"
        evaluator pngPrefix (frame + 1) settingsR tick' layout out defsR refs' ↙
            ↪ term' go
```

36 src/Meta.hs

```
{-
    gulcii -- graphical untyped lambda calculus interpreter
    Copyright (C) 2011, 2013, 2017  Claude Heiland-Allen

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License along
```

```
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
-}

module Meta (Meta(..), parse) where

import Control.Applicative((<|>), (<$), (<*>))

import Setting (Setting)
import qualified Setting
import Parse

data Meta = Start | Stop | Quit | Clear | Browse | Load String | Get Setting | ↙
    ↳ Set Setting | UnSet Setting | Toggle Setting
  deriving (Read, Show, Eq, Ord)

parse :: Parser String Meta
parse =  (Start  <$ sym "start")
     <|> (Stop   <$ sym "stop")
     <|> (Quit   <$ sym "quit")
     <|> (Clear  <$ sym "clear")
     <|> (Browse <$ sym "browse")
     <|> (Load   <$ sym "load" <*> name)
     <|> (Get    <$ sym "get" <*> Setting.parse)
     <|> (Set    <$ sym "set" <*> Setting.parse)
     <|> (UnSet  <$ sym "unset" <*> Setting.parse)
     <|> (Toggle <$ sym "toggle" <*> Setting.parse)
```

37 src/Parse.hs

```
{-
    gulcii -- graphical untyped lambda calculus interpreter
    Copyright (C) 2011, 2013, 2017  Claude Heiland-Allen

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
-}

module Parse where

import Control.Applicative(Applicative, Alternative, pure, empty, (<|>), (<$>), ↙
    ↳ (<*>), (<*))

{-
Strip comments
--------------
```

```
A comment is everything from '#' to the end of the line.
-}

decomment :: String -> String
decomment = concatMap (fst . break ('#'==)) . lines

{-
Tokenize
--------
-}

digits :: String
digits = "0123456789"

lowers :: String
lowers = "abcdefghijklmnopqrstuvwxyz"

uppers :: String
uppers = "ABCDEFGHIJKLMNOPQRSTUVWXYZ@"

alpha :: String
alpha = lowers ++ uppers

alphanum :: String
alphanum = alpha ++ digits

symbols :: String
symbols = "\\.!?()[],=:"

spaces :: String
spaces = " "

{-
Split a string into tokens (each itself a string), such that each token
consists of either all digits, all letters, or a single symbol.  Use
whitespace to separate tokens.
-}

tokenize :: String -> Maybe [String]
tokenize [] = Just []
tokenize (c:cs)
  | c `elem` digits  = let (t,ts) = span (`elem` digits) cs
                       in ((c:t):) <$> tokenize ts
  | c `elem` uppers
  ||c `elem` lowers  = let (t,ts) = span (`elem` alphanum) cs
                       in ((c:t):) <$> tokenize ts
  | c `elem` symbols = ([c]:) <$> tokenize cs
  | c `elem` spaces  = tokenize cs
  | otherwise = Nothing

{-
Parsing primitives
------------------

A parser takes a list of tokens to a list of possible partial parses.
-}
```

```
85  newtype Parser s t = P{ unP :: [s] -> [(t, [s])] }

    instance Functor (Parser s) where
      fmap f (P p) = P (\q -> [ (f v, s) | (v,s) <- p q ] )

90  instance Applicative (Parser s) where
      pure f = P (\q -> [(f, q)])
      P p1 <*> P p2 = P (\q -> [ (u v, t) | (u, s) <- p1 q, (v, t) <- p2 s ])

    instance Alternative (Parser s) where
95    empty = P (\_ -> [])
      P p1 <|> P p2 = P (\q -> p1 q ++ p2 q)

    {-
    Accept a token that satisfies a predicate.
100 -}

    satisfy :: (s -> Bool) -> Parser s s
    satisfy p = P (\q -> case q of
      (x:xs) | p x -> [(x,xs)]
105   _ -> [])

    {-
    Accept a specific token.
    -}
110
    sym :: Eq s => s -> Parser s s
    sym a = satisfy (== a)

    {-
115 Accept some p's separated by s's.
    -}

    someSep :: Parser s a -> Parser s t -> Parser s [t]
    someSep s p = ((:[]) <$> p) <|> ((:) <$> p <* s <*> someSep s p)
120
    {-
    Accept some p's separated by s's, or nothing.
    -}

125 manySep :: Parser s a -> Parser s t -> Parser s [t]
    manySep s p = pure [] <|> someSep s p

    {-
    Accept a name consisting of letters.
130 -}

    name :: Parser String String
    name = P (\q -> case q of
      (p@(r:_):ps) | r `elem` lowers -> [(p, ps)]
135   _ -> [])

    {-
    Accept an integer consisting of digits.
    -}
140
```

```
integer :: Parser String Integer
integer = P (\q -> case q of
  (p:ps) | all (`elem` digits) p -> [(read p, ps)]
  _ -> [])
```

38 src/Reduce.hs

```
{-
    gulcii -- graphical untyped lambda calculus interpreter
    Copyright (C) 2011, 2013, 2017  Claude Heiland-Allen

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
-}

module Reduce (Reduce(..), Reduction, reduce) where

import Prelude hiding (replicate)

import qualified Data.Map.Strict as Map

import Evaluation (Strategy(..))
import Graph

{-
Reduce a graph one step, returning Nothing if it is irreducible.
-}

data Reduce = Beta | RefInst | Rebound String | Extrude
  deriving (Read, Show, Eq, Ord)

type Reduction = (Reduce, (References, Term))

mapTerm :: (Term -> Term) -> Reduction -> Reduction
mapTerm f = fmap (fmap f)


reduce :: Definitions -> References -> Term -> Maybe Reduction
reduce defs refs term =
  case reduce' False defs refs term of
    Nothing -> reduce' True defs refs term
    r -> r


reduce' :: Bool -> Definitions -> References -> Term -> Maybe Reduction

```

```
reduce' _ defs refs (Free var)
  = (,) (Rebound var) `fmap` (,) refs `fmap` Map.lookup var defs

reduce' _ _    _    (Bound0)
  = Nothing

reduce' _ _    refs (Scope (Lambda strat term))
  = Just (Extrude, (refs, Lambda strat (lifting refs 1 term)))

reduce' f defs refs (Scope t@(Reference _))
  = case dereference refs t of
      l@(Lambda _ _) ->
        Just (RefInst, (refs, Scope l))
      _ -> mapTerm Scope `fmap` reduce' f defs refs t

reduce' _ _    refs (Scope (Apply a b))
  = Just (Extrude, (refs, Apply (Scope a) (Scope b)))

reduce' f defs refs (Scope term)
  = mapTerm Scope `fmap` reduce' f defs refs term

reduce' f defs refs (Lambda strat term)
  = mapTerm (Lambda strat) `fmap` reduce' f defs refs term

reduce' f defs refs term@(Reference ref)
  = case Map.lookup ref refs of
      Just refTerm ->
        case reduce' f defs refs refTerm of
          Just (reason, (refs', term')) ->
            Just (reason, (Map.insert ref term' refs', term))
          Nothing ->
            Just (RefInst, (refs, refTerm))
      Nothing -> Nothing -- error "reference not found"

reduce' _ _    refs (Apply (Lambda Copy a) b)
  = Just (Beta, (refs, beta refs a b))

reduce' f defs refs (Apply l@(Lambda Strict a) b)
  = case reduce' f defs refs b of
      Just (reason, (refs', b')) -> Just (reason, (refs', Apply l b'))
      Nothing -> Just (Beta, (refs, beta refs a b))

reduce' _ _    refs (Apply (Lambda Lazy a) b)
  = let r = next refs
        refs' = Map.insert r b refs
    in  Just (Beta, ( refs'
                    , beta refs' a (Reference r) ))

reduce' f defs refs (Apply a b)
  = case (a, dereference refs a) of
      (Reference _, l@(Lambda _ _)) -> Just (RefInst, (refs, Apply l b))
      _ ->
       case reduce' f defs refs a of
          Just (reason, (refs', a')) ->
            Just (reason, (refs', Apply a' b))
          Nothing
            | f -> mapTerm (Apply a) `fmap` reduce' f defs refs b
```

```
            | otherwise -> Nothing

beta :: References -> Term -> Term -> Term
beta refs l v = substitute refs l v 0

substitute :: References -> Term -> Term -> Integer -> Term
substitute _    Bound0 s 0 = s
substitute _    Bound0 _ _ = Bound0
substitute _    (Scope t) _ 0 = t
substitute refs (Scope t) s i = Scope (substitute refs t s (i - 1))
substitute refs (Lambda k t) s i = Lambda k (substitute refs t s (i + 1))
substitute refs (Apply a b) s i = Apply (substitute refs a s i) (substitute refs
    ↳ b s i)
substitute _    t@(Free _) _ _ = t
substitute refs (Reference r) s i = substitute refs (refs Map.! r) s i

lifting :: References -> Integer -> Term -> Term
lifting _    0 t = Scope t
lifting _    _ s@(Free _) = s
lifting _    _ Bound0 = Bound0
lifting refs i (Scope t) = Scope (lifting refs (i - 1) t)
lifting refs i (Lambda k t) = Lambda k (lifting refs (i + 1) t)
lifting refs i (Apply a b) = Apply (lifting refs i a) (lifting refs i b)
lifting refs i (Reference r) = lifting refs i (refs Map.! r)


dereference :: References -> Term -> Term
dereference refs (Reference r) = dereference refs (refs Map.! r)
dereference _ t = t


next :: References -> Integer
next refs = case Map.maxViewWithKey refs of
  Nothing        -> 0
  Just ((k,_),_) -> k + 1
```

39 src/Setting.hs

```
{-
    gulcii -- graphical untyped lambda calculus interpreter
    Copyright (C) 2011, 2013, 2017  Claude Heiland-Allen

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
-}
```

```
20  module Setting (Setting(..), parse, Settings(..), get, set, defaults) where

    import Control.Applicative ((<$))
    import Data.Foldable (asum)

25  import Parse

    data Setting
      = TraceEvaluation
      | CollectGarbage
30    | RealTimeDelay
      | RealTimeAcceleration
      | RetryIrreducible
      | EmitStatistics
      | EmitRebindings
35    | EchoToStdOut
      | EchoToGUI
      | SaveImages
      | NewsOnTop
      deriving (Eq, Ord, Enum, Bounded, Read, Show)
40
    parse :: Parser String Setting
    parse = asum [ s <$ sym (show s) | s <- [minBound .. maxBound :: Setting] ]

    data Settings = Settings
45    { traceEvaluation :: Bool
      , collectGarbage :: Bool
      , realTimeDelay :: Bool
      , realTimeAcceleration :: Bool
      , retryIrreducible :: Bool
50    , emitStatistics :: Bool
      , emitRebindings :: Bool
      , echoToStdOut :: Bool
      , echoToGUI :: Bool
      , saveImages :: Bool
55    , newsOnTop :: Bool
      }

    get :: Setting -> Settings -> Bool
    get TraceEvaluation = traceEvaluation
60  get CollectGarbage = collectGarbage
    get RealTimeDelay = realTimeDelay
    get RealTimeAcceleration = realTimeAcceleration
    get RetryIrreducible = retryIrreducible
    get EmitStatistics = emitStatistics
65  get EmitRebindings = emitRebindings
    get EchoToStdOut = echoToStdOut
    get EchoToGUI = echoToGUI
    get SaveImages = saveImages
    get NewsOnTop = newsOnTop
70
    set :: Setting -> Bool -> Settings -> Settings
    set TraceEvaluation b s = s{ traceEvaluation = b }
    set CollectGarbage b s = s{ collectGarbage = b }
    set RealTimeDelay b s = s{ realTimeDelay = b }
75  set RealTimeAcceleration b s = s{ realTimeAcceleration = b }
```

```
    set RetryIrreducible b s = s{ retryIrreducible = b }
    set EmitStatistics b s = s{ emitStatistics = b }
    set EmitRebindings b s = s{ emitRebindings = b }
    set EchoToStdOut b s = s{ echoToStdOut = b }
    set EchoToGUI b s = s{ echoToGUI = b }
    set SaveImages b s = s{ saveImages = b }
    set NewsOnTop b s = s{ newsOnTop = b }

    defaults :: Settings
    defaults = Settings
      { traceEvaluation = False
      , collectGarbage = True
      , realTimeDelay = True
      , realTimeAcceleration = True
      , retryIrreducible = True
      , emitStatistics = True
      , emitRebindings = True
      , echoToStdOut = True
      , echoToGUI = False
      , saveImages = False
      , newsOnTop = True
      }
```

40 src/Sugar.hs

```
    {-
        gulcii -- graphical untyped lambda calculus interpreter
        Copyright (C) 2011, 2013, 2017  Claude Heiland-Allen

        This program is free software; you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation; either version 2 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License along
        with this program; if not, write to the Free Software Foundation, Inc.,
        51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
    -}

    {-

    Sugared term parser
    ===================

    Grammar
    -------

    T ::= v | '\\' v+ ('.' | '!' | '?') T | T+ | '(' T ')' | n | '[' (T (, T)*)? ']'↙
        ↳  | '{' v ':' T '}' T

    -}
```

```
module Sugar(Term(..), parse, desugar) where

import Control.Applicative((<|>), (<$>), (<$), (<*>), (<*), some, liftA2)
import Data.List ((\\))

import Parse
import qualified Lambda as U
import Evaluation (Strategy(..))

data Term
  = Variable String
  | Lambda Strategy [String] Term
  | Apply [Term]
  | Group Term
  | Natural Integer
  | List [Term]
  deriving (Read, Show, Eq, Ord)

parse :: Parser String Term
parse = Apply <$> some parse'

parse' :: Parser String Term
parse' =  (flip Lambda <$ sym "\\" <*> some name <*> strategy <*> parse)
      <|> (Group <$ sym "(" <*> parse <* sym ")")
      <|> (Variable <$> name)
      <|> (Natural <$> integer)
      <|> (List <$ sym "[" <*> manySep (sym ",") parse <* sym "]")

strategy :: Parser String Strategy
strategy =  (Strict <$ sym "!") <|> (Lazy <$ sym ".") <|> (Copy <$ sym "?")

{-
A list of all possible legal variable names, need to generate fresh
variables (ie, names guaranteed to be unused in a given term).
-}

variables :: [String]
variables = [ v:vs | vs <- [] : variables, v <- ['a'..'z'] ]

{-
Desugar according to the following conventions:

  Outermost parentheses are dropped:
    M N means of (M N)

  Applications are left associative:
    M N P means (M N) P

  The body of an abstraction extends as far right as possible
    \x.M N means \x.(M N) and not (\x.M) N

  A sequence of abstractions are contracted:
    \x.\y.\z.N is abbreviated as \x y z.N

Desugar naturals with their Scott-encoding:

  0      -> \s z . z
```

```
  (1+n)  -> \s z . s (desugar n)

Desugar lists with their Scott-encoding:

  []     -> \ c n . n
  (x:xs) -> \ c n . c (desugar x) (desugar xs)
-}

desugar :: Term -> Maybe U.Term
desugar (Variable v) = Just $ U.Variable v
desugar (Lambda _ [] _) = Nothing
desugar (Lambda k [v] t) = fmap (U.Lambda k v) $ desugar t
desugar (Lambda k (v:vs) t) = fmap (U.Lambda k v) $ desugar (Lambda k vs t)
desugar (Apply []) = Nothing
desugar (Apply ts) = foldl1 (liftA2 U.Apply) (map desugar ts)
desugar (Group t) = desugar t
desugar (Natural 0) = Just $ lam "s" (lam "z" (U.Variable "z"))
desugar (Natural n) =
  let Just nn = desugar (Natural (n - 1))
      (s:z:_) = variables \\ U.freeVariablesIn nn
  in  Just $ lam s (lam z (U.Apply (U.Variable s) nn))
desugar (List []) = Just $ lam "c" (lam "n" (U.Variable "n"))
desugar (List (l:ls)) =
  let t = desugar l
      ts = desugar (List ls)
  in case (t,ts) of
    (Just t', Just ts') ->
      let (c:n:_) = (variables \\ U.freeVariablesIn t') \\ U.freeVariablesIn ts'
      in Just $ lam c (lam n (U.Apply (U.Apply (U.Variable c) t') ts'))
    _ -> Nothing

lam :: String -> U.Term -> U.Term
lam = U.Lambda Lazy
```

41 stack-10-5.yaml

```
resolver: lts-10.5

packages:
- .

extra-deps:
- gtk-0.14.7
- gio-0.13.4.1
```