Functional Programming – CS 3FP3

Assignment 4

February 22, 2021

Contents

1 Idea

So proofs, whether in 2DM3 or previously in this course, can get tedious and repetitve. So, let's automate them! (with material borrowed from thisYoutube video).

The idea is to construct a function that takes a list of **laws** and a given expression, and the result is a 'calculation': A sequence of steps where each step is obtained by applying the left-side of one of the given laws to the current expression, moreover the name of the law is printed in curly brackets between the expressions to which it applies.

- The calculation ends when no more laws can be applied.
- We prioritize laws that reduce the complexity of an expression –where 'complexity' can be thought of as the number of symbols. We refer to such laws as *basic laws* and they include, for example, `f . id = f` and `nil . f = nil`.

```
-- The first arguments is (basicLaws, otherLaws) where basic laws are prioritized.
calculate :: ([Law], [Law]) -> Expr -> Calculation
```

At each step of a calculation some subexpression of the current expression is *matched* against the left side of some law. The process results in a *substitution* for the variables occurring in the law.

For example, the expression
`pair (f . fst, g . snd) . pair(h, k)` matches against the left side of the pair fusion law
`pair(f,g) . h = pair(f.h, g.h)` via the substitution `f,g,h -> f . fst, g . snd, pair(h,k)`.

A crucial aspect our calculation algorithm is that laws should be applied only from left to right. This is to prevent calculations from looping. If laws could be applied in both directions, then an automatic calculator could always oscillate by applying a law in one direction and then immediately applying it in the reverse direction. A related argument holds for a law such as `iterate f = cons . pair (id, iterate f . f)`. The appearance of a term on both sides of the law means that it is possible for an automatic calculator to apply the law infinitely often, and so never reach a conclusion.

We may then prove an equation `l = r` by simplifying both sides to the same result. The two calculations can then be pasted together by reversing the steps of the second one. We'll eventually have something like this:

```
proveEqn :: [Law] -> (Expr, Expr) -> Calculation
proveEqn = ...

prove :: [Law] -> String -> String
prove laws = printCalc . proveEqn laws . parseEqn
```

Then, use `parseLaw` we can set up a database of laws. Each law consists of a name and an equation, where the two are seperated by a colon and the equation is two expressions seperated by an equals sign. Recall that expressions are built from constants, like `fst` and `pair`, variables like `f` and `g`, and compositions of expressions.

```
laws, filters, ifs, pairs, others :: [Law]

laws = filters ++ ifs ++ pairs ++ others
filters = map parseLaw
  [ "definition filter: filter p = concat.map(guard p)"
  , "definition guard:  guard p  = if(p, wrap, nil)"
  ]

ifs = map parseLaw
  [ "if over composition: if(p, f, g) . h = if(p.h, f.h, g.h)"
  , "composition over if: h . if(p, f, g) = if(p,   h.f, h.g)"
  ]

pairs = map parseLaw
  [ "definition fst:     fst . pair(f, g) = f"
  , "definition snd:     snd . pair(f, g) = g"
  , "definition cross:   cross(f, g)      = pair(f . fst, g . snd)"
  , "pair fusion:        pair(f,g) . h    = pair(f . h, g . h)"
  ]
```

```
others = map parseLaw
  [ "nil constant:   nil . f        = nil"
  , "nil natural:    map f . nil    = nil"
  , "wrap natural:   map f . wrap   = wrap . f"
  , "concat natural: map f . concat = concat . map (map f)"
  , "map functor:    map f . map g  = map (f . g)"
  ]
```

Many more useful laws could have been included, but with these we can now obtain a nice proof:

```
> prove laws "filter p . map f = map f . filter (p . f)"

  filter p . map f
=  { definition filter }
  concat . map(guard p) . map f
=  { map functor }
  concat . map(guard p . f)
=  { definition guard }
  concat . map(if(p, wrap, nil) . f)
=  { if over composition }
  concat . map(if(p . f, wrap . f, nil . f))
=  { nil constant }
  concat . map(if(p . f, wrap . f, nil))
=  { wrap natural }
  concat . map(if(p . f, map f . wrap, nil))
=  { nil natural }
  concat . map(if(p . f, map f . wrap, map f . nil))
=  { composition over if }
  concat . map(map f . if(p . f, wrap, nil))
=  { definition guard }
  concat . map(map f . guard(p . f))
=  { map functor }
  concat . map(map f) . map(guard(p . f))
=  { concat natural }
  map f . concat . map(guard(p . f))
=  { definition filter }
  map f . filter(p . f)
```

2 What you need to do

Your are provided with source code contains following almost-completed modules:

```
src
|--- Expr.hs
|--- Law.hs
|--- Matching.hs
|--- Parser.hs
|--- Rewriting.hs
|--- Test.hs
```

In each module, there will be some todo(s) for you to fill-in. The suggested order to read and complete the implementation is:

`Parser.hs -> Expr.hs -> Law.hs -> Matching.hs -> Rewriting.hs`

Some functions are documented carefully (especially those you need to fill-in implementation), some of them are not really. However, you are expected to understand them all, as reverse-engineering non-trivial code is an essential part of being a programmer.

The `Test.hs` file is for checking your implementation, and the content of the file itself should be self-explanatory. In short, you'll need to:

- Complete all the **todo** in all modules.
- Every time you complete a todo, use `Test.hs` to see if your implementation passes the test.
  - Your code not passing `Test.hs` means your implementation is incorrect, but yours passing does not mean it is correct.
- Submit your code.

The following section walk you through the source code so you have better idea what is going on. But it's fine if you jump in and start with the code right away.

3 Code walkthrough

3.1 Parser

The parser is what gives things a nice concrete syntax. It is helpful to make it an instrance of `Monad`.

```haskell
instance Monad Parser where
  return = success
  (>>=) = andThen
```

There is nothing special about making Parser an instance of Monad other than it allows us to use to syntatic sugar `do` notation. That is, instead of writing:

```haskell
exampleParser =
  digit `andThen` \l ->
  char '+' `andThen` \_ ->
  digit `andThen` \r ->
  success (l + r)
```

We can write it as:

```haskell
exampleParser = do
  l <- digit
  char '+'
  l <- digit
  return (l + r)
```

3.2 Expressions

Expressions is in module `Expr.hs`. In the proposed framework, **every** expression describes a **function**. An expression is built from (function) variables, constants, or composition.

- Variables are **single-letter** names, they represent some unknown functions and take no arguments, for example `f`, `x`, `y`.
- Constant functions are **multi-letter** names (i.e, `f` is not a valid name for a constant function), they can take any number of arguments, which themselves are expressions. Examples are `zip`, `plus`, `unzip`.

```
-- For simplicity, single-letter names denote variables and multi-letter names denote constants.
type VarName = Char

type ConName = String

data Expr
  = Var VarName
  | Con ConName [Expr]
  | Compose [Expr]
  deriving (Eq)
```

It is the intention of the design that associativity of functional composition is built-in. Thus, both `(f . g) . h` and `f . (g . h)` will be represented by the same expression, `Compose [Var 'f', Var 'g', Var 'h']`. As a result, `Expr` has two datatype invariants: The expression `Compose es` is valid only if the length of `xs` is at least two, and **no** expression of the form `Compose es` contains an element of `es` that is itself of the form `Compose fs`.

To maintain these invariants, compositions of expressions are constructed by the smart function `compose`:

```
compose :: [Expr] -> Expr
compose xs = todo "compose"
```

The rest is to implement a parser for `Expr`, which would give us an expression from valid input string.

```
expr :: Parser Expr
expr = do
  xs <- someWith (token (char '.')) term
  return (compose xs)
```

All the code for parsing is provided, but you are **highly encouraged** to read and understand it.

3.3 Laws & Calculations

Laws and calculations are defined in module `Law.hs`. An equational law consists of a name and two expressions –one for each side of the law.

```
type LawName = String

data Law
  = Law
      LawName -- Name
      Expr -- LHS
      Expr -- RHS
  deriving (Eq)
```

By definition, a law is basic if the complexity of its left side is greater than its right.

```
basicLaw :: Law -> Bool
basicLaw = todo "basicLaw"
```

A calculation consists of a starting expression together with a sequence of steps. Each step consists of the name of the law being applied and the resulting expression.

```
type Calculation = (Expr, [Step])
type Step        = (LawName, Expr)
```

Similarly, we'll have code to parse Law from given string. This is rather trivial once we have Parser for expressions.

```
eqn :: Parser (Expr, Expr)
eqn = do
  space
  x <- expr
  token (char '=')
  y <- expr
  return (x, y)

law :: Parser Law
law = do
  space
  name <- many (sat (/= ':'))
  token (char ':')
  (lhs, rhs) <- eqn
  return (Law name lhs rhs)
```

3.4 Matchings and Substitutions

Matching and substitutions are defined in `Matching.hs`.

To apply a law to a given expression we need to find a subexpression that **matches** the left hand side of the rule. We will now specify what does it mean to say a pattern matches an expression, and we'll deal with subexpression in the next section.

The result of a match is the list of bindings:

```
type Subst = [(VarName, Expr)]
```

A match may fail, yielding the empty set of substitutions, or it may succeed in one or more ways.

```
-- match a pattern against an expressions, return a list of possible substitutions
match :: Expr -> Expr -> [Subst]
```

For example, matching `foo(f . g, f . g)` against `foo(a . b . c, a . b . c)` succeeds in two different ways, either by binding `f` to `a . b` and `g` to `c`, or by binding `f` to `a` and `g` to `b . c`.

In general, later matchings reduce the set of possible substitutions from an earlier matching. For example, in matching `foo(f . g, bar g)` against `foo(a . b . c, bar c)`, the subexpression `f . g` is matched against `a . b . c` with two possible substitutions, yet only when the subexpression `bar g` is matched with `bar c` is one of the substitutions rejected. As such, a premature commitment to a single substitution for the first pair of subexpressions may result in a successful substitution being missed. You'll see in the code we handle this by having an auxiliary function `xmatch` that takes an extra `Subst`, which can be thought as a *base* or *earliere* matchings.

Note that an empty Substitution does not represent the undefined substitution, but rather the substitution in which each variable `v` is associated with `Var v`. Thus an empty list represents the *identity substitution.*

Matching a variable against an arbitrary expression always succeeds. However, a variable can only correspond to one expression. For example:

```
> match (parseExpr "zip(x,x)") (parseExpr "zip(a . b, a . b)")
[[('x',a . b)]] -- succeed with one possible substituion
> match (parseExpr "zip(x,x)") (parseExpr "zip(a . b, a . c)")
[] -- failed, x can't coresspond to both `a . b` and `a . c`
```

A constant function pattern matches constant expression if and only if:

- Constant function names are identical.
- Coressponding arguments match.

```
xmatch base (Con f patterns) (Con g exprs) =
  if f == g
    then xmatchlist base (zip patterns exprs)
    else []
```

Match a composition pattern agains a composition expression is more tricky. For example, if we want to match `f . g . zip(x, y)` against `pair(a, b) . m . n . p . q`, we first need to break down the composition `pair(a, b) . m . n . p . q` into composition of length 3, e.g, `(pair(a, b) . m) . n . (p . q)`, then try to match `f` against `pair(a, b) . m`, `g` against `n`, and `zip(x, y)` against `(p . q)`. You'll find instruction on this in the source code.

3.5 Rewriting

The final part is rewriting, i.e, apply a rule to an expression turning it to another expression, and it involes 2 steps:

- Find a subexpression that matchs the left hand side of the law.
- Replace that subexpression with the corresponding right hand side of the law.

What are subexpressions of a given expression `e`?
First, `e` is a subexpression of itself.
Second, if we look at the type `Expr`:

```
data Expr
  = Var VarName
  | Con ConName [Expr]
  | Compose [Expr]
  deriving (Eq)
```

Then naturally, those `[Expr]` and their subexpressions also are subexpressions of `e`.
Finally, for compositions `Compose [Expr]`, because composition is associativity and the way we encode them, any composition of `l >= 2` contiguous `Exprs` in the list is also a subexpression. For example, for `e = f . g . h . zip . add`, then `f . g`, `g . h . zip` are examples of subexpresions of `e`.
From that definition, can you write

```
subs :: Expr -> [Expr]
```

That list all subexpressions of a given expression?
So we know how to find subexpressions, and know how to subexpressions against left hand side of a law. However, that much is not enough to do the second step, i.e, replacing since we know nothing about the matched subexpression's whereabout (if you want to replace something in your house, you need to know where it is first).
That's why we need to accompany location information along with subexpressions.

```
-- | Location of a subexpression w.r.t it's parent
data Location
  = -- `All` refers to the whole expression
    All
  | -- `Seg i l` is the composition segment of length l beginning at position i (0-index)
    -- E.g: expr = Compose [f, zip, g, h], then
    -- `Seg 2 2` refers to `Compose [g, h]`
    -- `Seg 0 3` refers to `Compose [f, zip, g]`
    Seg Int Int
```

```
  | -- `Arg i loc` goes to i-th argument (0-index), then keeps going to
    --  location `loc` of that argument.
    -- E.g: expr = Con "mul" [f, g, h, Compose [x, y, z], m] then
    --  `Arg 3 (Seg 0 2)` refers to `Compose [x, y]`
    --  `Arg 0 All` refers to `f`
    Arg Int Location
  deriving (Show)

-- Subexpression in their parent expressions.
data SubExpr
  = SubExpr
      Expr -- The subexpression
      Location -- Location of the subexpression w.r.t its parent
```

And instead of `subs` above, you'll need to write:

```
subExprs :: Expr -> [SubExpr]
subExprs = todo "subExprs"
```

That would give us all subexpressions along with their locations like this:

```
> e = parseExpr "f . g . zip (m . n, p . q)"
> subExprs e
[ subexpression f . g . zip(m . n, p . q) located at All
, subexpression f . g                     located at Seg 0 2
, subexpression g . zip(m . n, p . q)     located at Seg 1 2
, subexpression f                         located at Arg 0 All
, subexpression g                         located at Arg 1 All
, subexpression zip(m . n, p . q)         located at Arg 2 All
, subexpression m . n                     located at Arg 2 (Arg 0 All)
, subexpression m                         located at Arg 2 (Arg 0 (Arg 0 All))
, subexpression n                         located at Arg 2 (Arg 0 (Arg 1 All))
, subexpression p . q                     located at Arg 2 (Arg 1 All)
, subexpression p                         located at Arg 2 (Arg 1 (Arg 0 All))
, subexpression q                         located at Arg 2 (Arg 1 (Arg 1 All))
]
```

The next thing you need to complete is `replace`.

```
-- | Replacing a subexpression of expression ~e~ at a location ~loc~ with a
--   replacement expression ~r~.
-- Note that we assume the location is valid w.r.t the expression, i.e
-- `map location (subExprs e)` contains `loc`
-- Example
-- > e = parseExpr "f . g . h . zip (m . n, mul)"
-- (e has subexpression m . n located at Arg 3 (Arg 0 All))
-- > replace e (Arg 3 (Arg 0 All)) (parseExpr "mul . add")
-- f . g . h . zip(mul . add, mul)
replace :: Expr -> Location -> Expr -> Expr
replace e All replacement = replacement
replace (Con f xs) (Arg j loc) y = todo "replace1"
replace (Compose xs) (Arg j loc) y = todo "replace2"
replace (Compose xs) (Seg j k) y = todo "replace3"
```

Note that we only have to consider so many cases because location `(Seg i j)` can only refer to `Compose [Expr]`.

Now we're ready to rewrite expressions . Remember `applyLaw` returns **all** possible ways to rewrite the expression with given law. As you expected, you'll need to use `match` and `replace`.

```haskell
-- | Given a pair of laws and an expression `e`, returns all possible pair way of rewriting `e`
-- The result is the list of (LawName, Expr), where LawName is the name of the applicable law, and
-- Expr is the result of applying LawName to `e`.
rewrite :: ([Law], [Law]) -> Expr -> [(LawName, Expr)]
rewrite (llaws, rlaws) x = concat $
  [applyLaw law sx x | law <- llaws, sx <- subExprs x] ++
  [applyLaw law sx x | law <- rlaws, sx <- subExprs x]
  where
    applyLaw :: Law -> SubExpr -> Expr -> [(LawName, Expr)]
    applyLaw (Law name lhs rhs) (SubExpr sub loc) exp = todo "appyLaw"
```

The rest of the code provide a naive and simple `calculate` that repeatly chose the first rewrite until no law is applicable.

3.6 Automatic theorem prover

Now we can finally put everything together to build an automatic theorem prover.

Let's have a database of laws as in introduction.

```haskell
laws, filters, ifs, pairs, others :: [Law]

laws = filters ++ ifs ++ pairs ++ others
filters = map parseLaw
  [ "definition filter: filter p = concat.map(guard p)"
  , "definition guard:  guard p  = if(p, wrap, nil)"
  ]

ifs = map parseLaw
  [ "if over composition: if(p, f, g) . h = if(p.h, f.h, g.h)"
  , "composition over if: h . if(p, f, g) = if(p,   h.f, h.g)"
  ]

pairs = map parseLaw
  [ "definition fst:     fst . pair(f, g) = f"
  , "definition snd:     snd . pair(f, g) = g"
  , "definition cross:   cross(f, g)      = pair(f . fst, g . snd)"
  , "pair fusion:        pair(f,g) . h    = pair(f . h, g . h)"
  ]

others = map parseLaw
  [ "nil constant:   nil . f        = nil"
  , "nil natural:    map f . nil    = nil"
  , "wrap natural:   map f . wrap   = wrap . f"
  , "concat natural: map f . concat = concat . map (map f)"
  , "map functor:    map f . map g  = map (f . g)"
  ]
```

Equational proving is just 2 calculation with an additional link:

```haskell
proveEqn :: [Law] -> (Expr, Expr) -> Calculation
proveEqn laws (lhs, rhs) = paste (calculate (basic, others) lhs) (calculate (basic, others) rhs)
  where (basic, others) = partition basicLaw laws
```

Finally, our theorem prover:

```haskell
prove :: [Law] -> String -> String
prove laws = printCalc . proveEqn laws . parseEqn
```

4 Submission Instruction

After completing all **todoes**, you should put all the files in a single folder `src`, then zip it and name the file `A3_{your_mac_id}.zip` and submit to Avenue.