

Mote: Goal-Driven Development and Synthesis for Haskell

by

Izaak Meckler

A thesis submitted in partial satisfaction of the
requirements for the degree of
Bachelor of Science

in

Computer Science

in the

Undergraduate Division

of the

University of Chicago

The thesis of Izaak Meckler, titled Mote: Goal-Driven Development and Synthesis for Haskell, is approved:

_____	Date _____
_____	Date _____
_____	Date _____

University of Chicago

Mote: Goal-Driven Development and Synthesis for Haskell

Copyright 2015
by
Izaak Meckler

Abstract

Mote: Goal-Driven Development and Synthesis for Haskell

by

Izaak Meckler

Bachelor of Science in Computer Science

University of Chicago

,

This thesis describes the design of an editor plugin called Mote. Mote brings several enhancements to Haskell programming, including support for goal-oriented programming, automatic generation of pattern matching expressions, and type directed synthesis of short Haskell expressions. Our synthesis strategy includes a method for eliminating duplicates from synthesized expressions based on the string diagrams of category theory. We describe in detail the theory and algorithms involved in this application of string diagrams.

Contents

Contents	i
List of Figures	ii
Notation	iv
1 Mote	1
1.1 Pattern matching boilerplate	2
1.2 Managing the types of expressions with holes	3
1.3 Synthesis of small expressions	6
2 Searching for polymorphic programs	13
2.1 Preliminaries	14
2.2 A calculus for natural transformations	15
2.3 A semantics in categories	17
2.4 A semantics in string rewriting (a special case)	19
2.5 Completeness	22
2.6 Some proof theoretic observations	23
2.7 String diagrams for efficient search	24
2.8 String diagrams for eliminating redundancy	29
2.9 Remarks on implementation	34
2.10 Turning string digrams to terms	35
2.11 Finding a topmost vertex	36
2.12 Future possibilities	39
Bibliography	40

List of Figures

1.1	Using holes in Agda	2
1.2	An example usage of Mote’s case expansion facilities (1)	4
1.3	An example usage of Mote’s case expansion facilities (2)	4
1.4	Using holes in Mote (1)	5
1.5	Using holes in Mote (2)	5
1.6	Discovered programs of type <code>[Filepath] -> IO [String]</code>	8
1.7	A diagram of <code>findAllTodoLines</code>	9
2.1	A diagram of <code>randomlyNothing</code>	25
2.2	A string diagram of <code>randomSublist</code>	26
2.3	Translations for the remaining rules of \mathcal{N}	28
2.4	Naturality evidently holds for string diagrams	28
2.5	Vertical (cut) and horizontal (juxtapose, or zip) composition evidently commute for string diagrams	29
2.6	The essential rules for our string calculus	30
2.7	The action of $[[A_i]]$ on morphisms	31
2.8	Naturality for n	32
2.9	A trivial diagram	32
2.10	Decomposition of a string diagram	33
2.11	A fusion optimization on string diagrams	35
2.12	A fusion optimization in translating a string diagram into a term	36

Glossary

Σ^* The set of strings over the alphabet Σ .

Σ An alphabet. That is, a finite set of “symbols”..

η, φ Variables for natural transformations.

η_X The component of the natural transformation at the object X .

$\llbracket s \rrbracket$ The interpretation of some syntactic entity s in some model.

\mathcal{C} Variable for a category.

$\text{Hom}_{\mathcal{C}}(X, Y)$ The collection of morphisms from X to Y in the category \mathcal{C} .

$\text{Pre}(X)$ The preorder category on the partially ordered set X .

\underline{A} A sequence A_1, A_2, \dots

inhabited A type or set is inhabited if there is something of that type or in that set (cf. nonempty).

Acknowledgments

First off, I want to thank Ravi Chugh, my thesis advisor. This thesis would not be possible without your guidance and willingness to listen to my ranting. I'd also like to thank all of my friends who discussed with me the ideas described here. Specifically, thank you to Brandon Rayhaun and Kyle Gannon. Finally, I'd like to thank my parents for their perpetual support and love.

Chapter 1

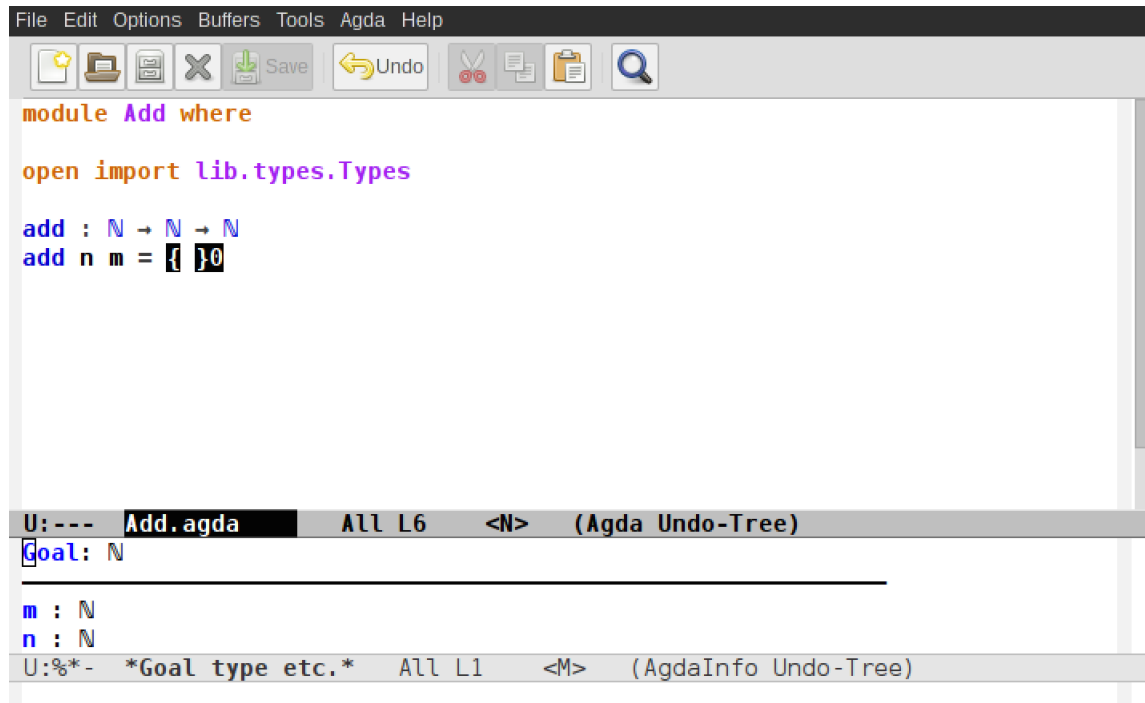
Mote

This thesis documents the design of Mote, an editor plugin for making Haskell programming easier and more enjoyable. On a high level, Mote brings the following enhancements to Haskell development:

1. Writing pattern matching expressions requires knowing the variants of the type of the expression you wish to match on, and is needlessly laborious for types with many variants. To this end, Mote provides a facility for automatic generation of case expressions.
2. The only way to know the types of variables in scope is to know the types of the functions which produced them or the types of the patterns they were bound in, both of which require memorizing APIs or jumping back and forth between the program and documentation (or other modules). Mote’s use of *holes* helps to solve this problem.
3. Often fragments of a program can be automatically generated from values in scope. Mote thus includes a limited facility for synthesizing small expressions.

The Agda Emacs mode [4] does an excellent job of addressing aspects of these problems. Programming using it involves the use of *holes*. A *hole* is a placeholder expression that one types as a subexpression when not immediately sure what should be written. In Agda, holes are introduced by typing `?`. After loading the file in Emacs, the hole is rendered as an underscore surrounded by brackets, as in Figure 1.1. Agda also gives the programmer an info panel showing the types of local variables in the hole, as well as the type of expression which the hole should be filled in with. This type is called the goal type of the hole, or just the type of the hole.

The expression which one fills a hole with often itself has further holes. The idea is to program by constructing terms piece by piece, replacing large goals with “smaller” goals by constructing the eventual program from the outside in. Agda also



```

File Edit Options Buffers Tools Agda Help
[Icons: Save, Undo, Cut, Copy, Paste, Find]

module Add where

open import lib.types.Types

add : ℕ → ℕ → ℕ
add n m = { }

U: --- Add.agda All L6 <N> (Agda Undo-Tree)
Goal: ℕ
-----
m : ℕ
n : ℕ
U:%*- *Goal type etc.* All L1 <M> (AgdaInfo Undo-Tree)

```

Figure 1.1: Using holes in Agda

has facilities for automatic generation of pattern matches and program synthesis, which will be discussed later.

As for Haskell, there is a popular Haskell editor plugin called `ghc-mod`[7], which has a very stable implementation and offers features including in-editor compilation error messages and the ability to query for the types of expressions.

In previous versions, it seems to have supported automatic generation of case expressions and limited hole-based programming. As of the current version, correspondence with the package’s maintainer these facilities indicates that these features are broken or very limited, and we were not able to use them.

1.1 Pattern matching boilerplate

Defining programs by pattern matching is one of the most ubiquitous and useful features of typical functional programming languages. However, writing case expressions can be tedious and requires complete knowledge of the structure of the type of the expression that one is trying to pattern match on. That is, to pattern match on an expression of type T , one must know all the constructors of T and, at the least, the arity of each constructor.

Consider for example the following type which describes expressions in a simple language

```
data Expr
  = Var Var
  | Let [(Pattern, Expr)] Expr
  | Case Expr [(Pattern, RightHandSide)]
  | Lam Var Expr
  | App Expr Expr
```

where `Var`, `Pattern`, and `RightHandSide` are defined elsewhere.

As mentioned, defining a function out of this type by pattern matching requires not only knowing all of the many variants and then laboriously writing a pattern for each of them, but likely the types of the constructors' arguments as well. Mote addresses both of these two problems, though only the first will be discussed in this section.

Drawing inspiration from Agda, to solve the problem of having to know and write all the possible variants of the type, given an expression to case on with a known type, Mote can automatically generate a case expression with one branch for every constructor of that type. The variables bound in the patterns are given names that reflect their type. Specifically, for a base type whose name is t , the name for a variable of that type will consist of all the upper case letters in t . For compound types, names are built up via some inductive rules. In addition, features of Mote described in section 1.2 allow the programmer to easily inspect the types of the variables bound in patterns.

In Haskell, the analogous expression type^[6] has 46 variants whose arguments, taken together, have around 35 distinct types. In such an instance, Mote's case expansion facilities really shine (as the author can attest to after having to write numerous functions by pattern matching on Haskell's expression type for the present work), but this simplified example suffices to illustrate the problem and how Mote can help.

A sample usage of this feature is shown in Figures 1.2 and 1.3. The vim command `:CaseOn EXPR` inserts an exhaustive case expression on the expression `EXPR` in the current hole.

1.2 Managing the types of expressions with holes

Often one wants to make partial progress in the writing of an expression without having to complete it. Moreover, while programming one needs to keep track of the types of expressions in scope. This can be difficult in practice when values have complex types, or when the expressions come from an external library.

```

14 evaluate :: Expr -> Expr
@ 15 evaluate e =           
~
~
~
~
~
~
~
~
~
~
<thesis/code_examples/AST.hs [haskell] [pos=15,14] [len=15 (100%)]
1 Goal: _ :: Expr
2 -----
3 e      :: Expr
4 evaluate :: Expr -> Expr
5
6
HoleInfo [pos=1,1] [len=7 (14%)]

```

Figure 1.2: An example usage of Mote’s case expansion facilities (1)

```

14 evaluate :: Expr -> Expr
15 evaluate e = case e of
16   Var v ->           
17   Let p_and_es e ->           
18   Case e p_and_rhss ->           
19   Lam v e ->           
20   App e e' ->           
~
~
~
~
~
~
~
~
~
~
<thesis/code_examples/AST.hs [haskell] [pos=17,16] [len=20 (85%)]

```

Figure 1.3: An example usage of Mote’s case expansion facilities (2)

```

13
14 evaluate :: Expr -> Expr
15 evaluate e = case e of
16   Var v -> █
@ 17   Let p_and_es e -> █
18   Case e p_and_rhss -> █
19   Lam v e -> █
20   App e e' -> █
~
~
~
<thesis/code_examples/AST.hs [haskell] [pos=17,21] [len=20 (85%)]
1 Goal: _ :: Expr
2 -----
3 e      :: Expr
4 p_and_es :: [(Pattern, Expr)]
5 e      :: Expr
6 evaluate :: Expr -> Expr
HoleInfo [pos=1,1] [len=9 (11%)]

```

Figure 1.4: Using holes in Mote (1)

```

13
14 evaluate :: Expr -> Expr
15 evaluate e = case e of
16   Var v -> █
@ 17   Let p_and_es e ->
18     let eval'd_bindings = map █ p_and_es in
19     █
20   Case e p_and_rhss -> █
21   Lam v e -> █
22   App e e' -> █
~
~
~
<thesis/code_examples/AST.hs [haskell] [pos=18,31] [len=22 (81%)]
1 Goal: _ :: (Pattern, Expr) -> b
2 -----
3 eval'd_bindings :: [b]
4 e               :: Expr
5 p_and_es        :: [(Pattern, Expr)]
6 e               :: Expr
HoleInfo [pos=1,1] [len=10 (10%)]

```

Figure 1.5: Using holes in Mote (2)

Consider again the above example of writing a function `evaluate :: Expr -> Expr`. Suppose we want to write the `Let` case. First, we navigate to the appropriate hole using Mote’s `NextHole` command (Figure 1.4). Then, knowing that the evaluation of the expression `e` will depend on the values bound by the `Let`, we write a sketch of a program that will evaluate and use the bindings (Figure 1.5). The type environment panel informs our writing of the sketch by providing us of the types of the arguments of the `Let` constructor without having to refer to the definition of the `Expr` type. We can then move into the new hole we’ve created and examine the local type environment to plan our next move.

1.3 Synthesis of small expressions

The third major enhancement Mote brings to Haskell programming is the automatic synthesis of small expressions. Program synthesis[3] is the automatic generation of programs from a statement of intent from a user. Often, the statement of intent is a constraint that a generated program should satisfy, for example a test case the program should pass or a type it should have. In the present work, this constraint will be the latter. That is, we describe aspects of the following problem: given a type, generate a program that has that type.

Agda has the ability to synthesize terms as well, though the search returns at most one term and makes no attempt to find a term the programmer is likely to want. This is because often in Agda, either the types are sufficiently specific that only one term will fit, or the programmer is just trying to finish a proof and doesn’t care much about its computational behavior. Thus, in Haskell the approach must be quite different since the programmer certainly cares about the actual behavior of a synthesized term.

Mote’s search finds as many terms as possible fitting a given type. Typically, this is quite a large number of terms. In order to make the results more useful for the user, we make some attempts reduce the size of this list. Specifically, we eliminate duplicates: terms which are syntactically different but semantically equivalent. To do so, we make use of objects called string diagrams. String diagrams are a representation of programs which identify certain semantically equivalent programs.

Mote is capable of synthesizing programs with types of the form

$$\forall \alpha. F\alpha \rightarrow G\alpha$$

where F and G are functors (possibly constant or identity). That is, $F\alpha$ is a Haskell expression of kind `*` of the form

$$F_1(F_2(\dots A \dots))$$

where A is either the type variable α or a closed type and each F_i is a Haskell expression of kind `* -> *` with a lawful functor instance. Here are a few examples of types of this form.

```
forall a. [IO (Maybe a)] -> IO [a]
forall a. FilePath -> IO [String]
forall a. [a] -> Int
```

Types with longer chains of applications of functors often appear in “effects-munging” code: code which executes and reorders effects (encoded as monads). While this was the original inspiration for the approach we take, the current search facility is more general.

Mote’s high level strategy is extremely simple. Suppose Mote is searching for a program of type $\forall\alpha.F\alpha \rightarrow G\alpha$. Mote will search for a sequence of functions which when composed have this type by doing a depth-limited depth first search. The depth (which corresponds to the length of the sequence of functions) is limited both for performance (since there is an expectation that in-editor tools be fast) and since the user is unlikely to want large terms. Once terms have been discovered, they are converted to string diagrams, duplicates are removed, and converted back to terms.

Let’s look at an example. Imagine that we have a list of paths to source files

```
sourceFiles :: [Filepath]
```

and also a function for reading the contents of a given file

```
readFileMay :: FilePath -> IO (Maybe String)
```

which returns `Nothing` if the file given as an argument does not exist. Suppose our goal is to find all lines in all of these files that contain a “TODO”. To that end, we have written a function

```
findTodos :: String -> [String]
findTodos = filter ("TODO" `isInfixOf`) . lines
```

where `isInfixOf` returns true if its first argument is a contiguous substring of the second and `lines :: String -> [String]` turns a file into the list of lines that comprise it.

Now, we write

```
allTodoLines :: IO [String]
allTodoLines = _ sourceFiles
```

meaning that our goal is to write a program of type `[Filepath] -> IO [String]` to turn the list of source filepaths into the list of “todo”s which they contain.

We can query Mote for a program of this type by executing the command `:MoteSearch [Filepath] -> IO [String]`. The info panel then displays a list of discovered programs of this type (shown in Figure 1.6), ordered by a heuristic ranking function discussed in Chapter 2. All the way as the 17th result is the program we are looking for:

```
fmap (concat . fmap findTodos . catMaybes) . sequenceA . fmap readFileMay
```


which concatenates all “todo”s in all of the files. Though it is the 17th result, notably, it is the first that uses `findTodos`. Let us call this function `findAllTodoLines`.

In future work, we hope to add a facility for the user to add hints to the search, such as a constraint that discovered programs must involve a particular function, so that desired programs appear higher in the list of results. For now, the user would have to do a textual search in the results panel to find this term.

```

1 fmap catMaybes . sequenceA . fmap readFileMay
2 fmap (concat . fmap maybeToList) . sequenceA . fmap readFileMay
3 (fmap . fmap) (unlines . maybeToList) . sequenceA . fmap readFileMay
4 (fmap . fmap) (unwords . maybeToList) . sequenceA . fmap readFileMay
5 fmap (concat . sequenceA . fmap maybeToList) . sequenceA . fmap readFileMay
6 fmap (fmap (unlines . lines) . catMaybes) . sequenceA . fmap readFileMay
7 fmap (catMaybes . fmap listToMaybe . fmap maybeToList) . sequenceA . fmap readFileMay
8 fmap (catMaybes . maybeToList . listToMaybe) . sequenceA . fmap readFileMay
9 fmap (maybeToList . join . listToMaybe) . sequenceA . fmap readFileMay
10 fmap (fmap (unwords . lines) . catMaybes) . sequenceA . fmap readFileMay
11 fmap (fmap (unlines . words) . catMaybes) . sequenceA . fmap readFileMay
12 fmap (concat . fmap lines . catMaybes) . sequenceA . fmap readFileMay
13 fmap (fmap unwords . sequenceA . fmap maybeToList) . sequenceA . fmap readFileMay
14 fmap catMaybes . sequenceA . fmap readFileMay . maybeToList . listToMaybe
15 fmap (concat . fmap words . catMaybes) . sequenceA . fmap readFileMay
16 fmap (concat . maybeToList . sequenceA) . sequenceA . fmap readFileMay
17 fmap (concat . fmap findTodos . catMaybes) . sequenceA . fmap readFileMay
18 fmap (fmap (unwords . findTodos) . catMaybes) . sequenceA . fmap readFileMay
19 fmap (maybeToList . join) . sequenceA . fmap readFileMay . listToMaybe
20 fmap (fmap unlines . maybeToList . sequenceA) . sequenceA . fmap readFileMay
21 fmap (catMaybes . sequenceA . sequenceA) . sequenceA . fmap readFileMay
22 fmap (fmap unwords . maybeToList . sequenceA) . sequenceA . fmap readFileMay
23 fmap (fmap unlines . sequenceA . fmap maybeToList) . sequenceA . fmap readFileMay
24 fmap (fmap (unwords . words) . catMaybes) . sequenceA . fmap readFileMay
25 fmap (catMaybes . maybeToList) . sequenceA . fmap readFileMay . listToMaybe
26 fmap (fmap (unlines . findTodos) . catMaybes) . sequenceA . fmap readFileMay

```

Figure 1.6: Discovered programs of type `[Filepath] -> IO [String]`

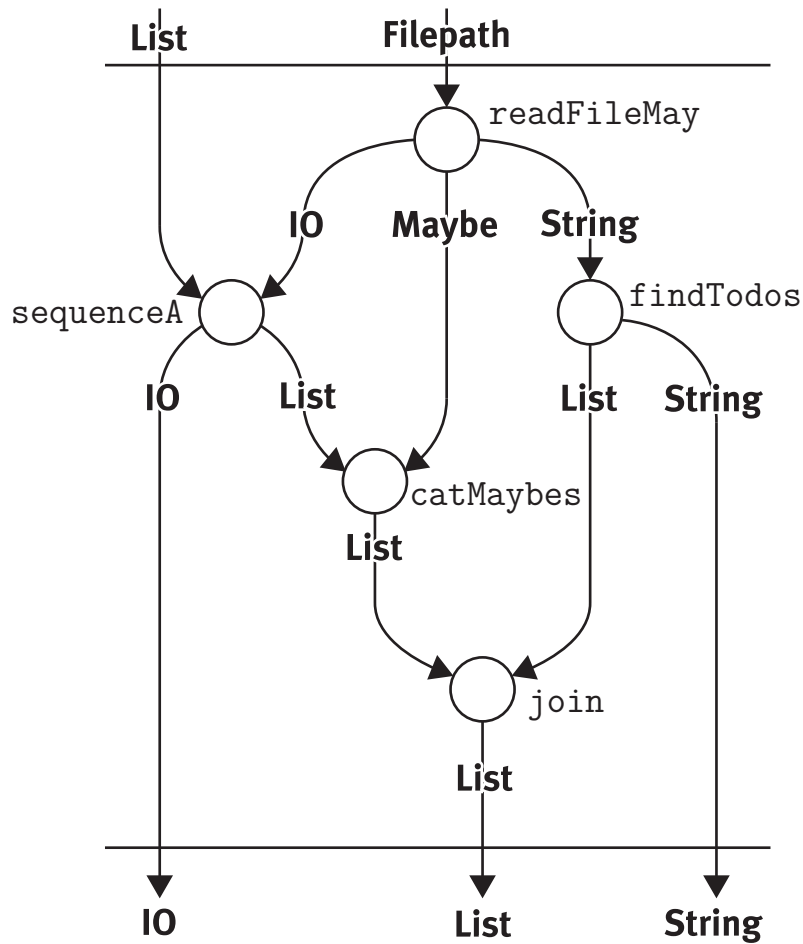
Figure 1.7 shows a *string diagram* of `findAllTodoLines`. A string diagram is a graphical notation from category theory for representing natural transformations, and will be discussed at length in the next chapter. Specialized to Haskell, string diagrams provide a way of representing functions that identifies some semantically equivalent expressions. For our purposes, this means we can eliminate duplicate search results before displaying them to the user.

As an example, `findAllTodoLines` could have been written in (at least) the following three ways (spaced out for readability):

```

fmap (join . catMaybes)
  . sequenceA

```

Figure 1.7: A diagram of `findAllTodoLines`

```

. (fmap . fmap . fmap) findTodos
. fmap readFileMay

fmap (join . catMaybes)
. (fmap . fmap . fmap) findTodos
. sequenceA
. fmap readFileMay

fmap (join . fmap findTodos . catMaybes)
. sequenceA
. fmap readFileMay

```

Though these terms are all syntactically different, they are semantically equivalent¹, as they all correspond to the same string diagram.

¹In an idealized version of Haskell. The proper qualifications are made in Chapter 2

By performing search on string diagrams rather than terms (or alternatively, by converting to string diagrams and then eliminating duplicates after performing search) we are able to make the list of terms displayed to the user more usable. In this particular example, the number of terms discovered before eliminating duplicates is 76, compared to the 26 afterwards.

Related work

There are many existing systems which integrate synthesis tools into a development environment. Agda, for example, has an “auto” command that triggers an attempt to search for a term of the type of the current hole. The search has a timeout of about five seconds. Agda uses a search strategy based on the idea of narrowing[16], which could easily be a complete proof strategy for a full MLTT-style dependently language. In practice, for performance reasons, the implementers of Agda limit the capabilities of the search (for example by disallowing induction) so that the strategy as implemented is not complete.

There are several related tools for Haskell specifically. One such tool is Lennart Augustsson’s Djinn[1], integrated in the `ghc-mod`[7] plugin. Djinn is a theorem prover for intuitionistic propositional logic capable of synthesizing simple programs, but whose utility is severely limited by the fact that it is only capable of reasoning about non-recursive polynomial types. For example, Djinn can infer a program of type `Either a b -> Bool` but cannot even be queried about a type like `[IO a] -> IO [a]`, since neither `IO` nor `[]` are non-recursive polynomial types.

A second is MagicHaskeller[13], which attempts to synthesize functions satisfying a boolean predicate. Mote’s synthesis capabilities differ in that it has no facility for constraining synthesized programs to satisfy behavioral predicates instead constraining them only on types. Notably, MagicHaskeller makes efforts to eliminate duplicate synthesized terms, as Mote does, inputs relying on equations like the η -rule and testing on random inputs to do so. Mote’s method for eliminating duplicate terms relies on naturality and functoriality equations, to be discussed later.

A third is the recently released Exference[23] which synthesizes Haskell terms which have a given type. Exference supports some features not currently in Mote, such as synthesis of terms which pattern match on single constructor types, and more generally, terms of a less restricted form than those generated by mote. At present, Exference searches for terms built out of a curated set of base terms whereas Mote’s Search uses whatever is in scope in the file in which it is invoked.

Another category of tool in this space, widely used in practice, is code completion systems for object-oriented languages like the partial completions of [18] or Intellisense[24]. Such systems, designed for object-oriented languages, rely on the fact that for any given value, it is immediately clear what operations may be performed on it: namely, the set of methods which that value has.

Things are somewhat less clear in functional languages. In a sense, the situation is similar in that the “only” operation which may be applied to a value of a given type is the eliminator for that type (or put another way, the only operation that can be performed is pattern-matching) and the elimination rule for objects is essentially access to their methods. In practice, this is not exactly the case as many definitions are built out of existing functions and not given by explicit pattern matching, and a search strategy based on this is not likely to be useful. When all things are possible, it is unlikely that the expression the user is looking for will be happened upon. For this reason, we suppose a closed set of operations that can be performed on a value of a type T which will essentially be the functions in scope taking an argument of type at least as general as T . This approach is analogous to that taken in the autocompletion methods mentioned above.

Some of the capabilities of Mote’s search seem to be similar to the Scala synthesis tool InSynth[9]. Namely, the ability to synthesize expressions involving polymorphic subexpressions. It should be stressed however that the primary technical innovation of Mote’s search is not quite in the searching itself, but in eliminating duplicates from the set of synthesized expressions.

Limitations and future work

As of now, the implementation cannot handle polymorphism in the functors themselves. For example, the system is unable to use the function `snd :: (x, a) -> a` to provide a term of type `(Int, a) -> a` since the functor applied to the source, `(,)` `x` is polymorphic in `x` and we currently make no attempt to unify it with `Int`. It should be fairly straightforward to extend the search with this ability, and we have already begun work on this problem.

Also, since the search strategy is a simple exhaustive search, searching for terms of size greater than about 6 takes more time than is acceptable for interactive usage. Notably, the current implementation does not terminate search after some number of solution programs have been found, but only after all have been found. It seems plausible that some heuristics to inform a “best-first” search, or terminating search after some number of programs judged to be good have been found could improve this limitation considerably. Exference[23] uses a best-first search strategy to achieve better performance.

Another limitation is that the search currently cannot use multi-argument elimination and introduction functions like `maybe :: a -> (a -> b) -> Maybe a -> b`, `either :: (a -> c) -> (b -> c) -> Either a b -> c`, `(,)` `:: a -> b -> (a, b)`. It would be a major achievement in our eyes to augment string diagrams to naturally represent sum and product types in such a way that many equations involving them (such as `(a, b) = (b, a)` or `(a, Either b c) = Either (a, b) (a, c)`) hold in the string diagram representation. The work in [21] seems relevant in approaching this problem.

Somewhat relatedly, our search cannot easily handle type constructors functorial in more than one argument. A satisfactory account of product and sum types would likely have the side effect of partially addressing this issue since in Haskell a typical type is constructed as a sum of products.

Chapter 2

Searching for polymorphic programs

Suppose you have a list of directories

```
dirs :: [DirPath]
```

and also a function for obtaining the paths to the files in a given directory

```
listDirectory :: DirPath -> IO (Maybe [FilePath])
```

which returns `Nothing` if the `DirPath` given as an argument does not exist. Now suppose you are trying to list all files in the directories `dirs`

```
allFiles :: IO [FilePath]
```

```
allFiles = _
```

We can fill this in with the predictable definition

```
allFiles :: IO [FilePath]
```

```
allFiles = (fmap (concat . catMaybes) . sequence) (map listDirectory dirs)
```

where

```
catMaybes :: [Maybe a] -> [a]
```

```
catMaybes = foldr (maybe id (:)) []
```

```
concat :: [[a]] -> [a]
```

```
concat = foldr (++) []
```

are library functions. The function

```
fmap (concat . catMaybes) . sequence
```

is a general purpose “effects-munging” function and is likely one of the few such functions of type

```
forall a. [IO (Maybe [a])] -> IO [a]
```

which is both short and useful.

In this chapter, we develop a theory to guide the development of Mote’s search functionality. I.e., for discovering programs with types of the form

forall a. F1 (F2 (... (Fn a) ...)) -> G1 (G2 (... (Gm a) ...))

2.1 Preliminaries

In what follows, we use the notions of categories, functors, and natural transformations extensively. For an introduction to these ideas, please see for example [2].

Important tools that we will make frequent use of are Wadler’s free theorems [25], which are essentially consequences of Reynold’s abstraction (or parametricity) theorem for System F [20]. In particular, we use the following fact. For any System F terms

$$\begin{aligned} \text{map}_F &: \forall \alpha \beta. (\alpha \rightarrow \beta) \rightarrow (F\alpha \rightarrow F\beta) \\ \text{map}_G &: \forall \alpha \beta. (\alpha \rightarrow \beta) \rightarrow (G\alpha \rightarrow G\beta) \\ \eta &: \forall \alpha. F\alpha \rightarrow G\alpha \\ f &: A \rightarrow B \end{aligned}$$

such that

$$\begin{aligned} \text{map}_F (\lambda x. x) &= \lambda x. x \\ \text{map}_G (\lambda x. x) &= \lambda x. x \end{aligned}$$

we have

$$\text{map}_G f \circ \eta = \eta \circ \text{map}_F f$$

where equality of functions is extensional equality. In reality, this fact of parametricity is very delicate, and adding the wrong features to System F will invalidate it. For example, if we add laziness and non-termination, as in Haskell, this fails to hold as the following example illustrates¹ [22]

Let

```
g :: [a] -> Maybe a
g _ = Just (let x = x in x)
```

If the free “theorem” were true, we would have

```
g . map (\_ -> ()) = fmap (\_ -> ()) . g
```

But

¹This example was generated by the wonderful tool “Automatically Generating Counterexamples to Naive Free Theorems” at <http://www-ps.iai.uni-bonn.de/cgi-bin/exfind.cgi>.

```
(g . map (\_ -> ())) [] = Just (let x = x in x)
(fmap (\_ -> ()) . g) [] = Just ()
```

In this work, we ignore this subtlety (and indeed, all subtleties related to \perp) and imagine we are working in an ideal subset of Haskell without non-termination. We also ignore the subtleties associated with making a category of Haskell types and functions. Explicitly, our (false, but not too badly false) assumptions are as follows.

- There is a category `Hask` whose objects are Haskell types and where $\text{Hom}_{\text{Hask}}(A, B)$ consists of the terms of type $A \rightarrow B$ identified by extensional equality.
- Every type constructor $F :: * \rightarrow *$ paired with a function $\text{mapF} :: \text{forall } a \ b. (a \rightarrow b) \rightarrow (F \ a \rightarrow F \ b)$ with $\text{mapF } (\backslash x \rightarrow x) = \backslash x \rightarrow x$ and $\text{mapF } f \ . \ \text{mapF } g = \text{mapF } (f \ . \ g)$ gives rise to an endofunctor on `Hask`.
- For any $F, G :: * \rightarrow *$ with corresponding maps on arrows mapF and mapG satisfying the functor laws, any function $f :: \text{forall } a. F \ a \rightarrow G \ a$ gives rise to a natural transformation from the functor arising from F to the functor arising from G .

In particular we have the following equation for any $g : A \rightarrow B$.

$$f \ . \ \text{mapF } g = \text{mapG } g \ . \ f$$

To make such assumptions is mainly just a linguistic convenience. For further information on the acceptability of reasoning under such assumptions, see [5].

2.2 A calculus for natural transformations

Our goal is a useful algorithm for the synthesis of Haskell programs of types of the form

```
forall a. F a -> G a
```

which, as per the above discussion, correspond to natural transformations from the functor corresponding to F to the functor corresponding G .

With this in mind, in this section, we present \mathcal{N} , a proof calculus for deriving natural transformations between functors, demonstrate a connection with string rewriting systems, and prove the completeness of a particular search strategy. Beginning in section 2.7, we present a modification of the strategy resulting in significant efficiency gains.

The basic ideas underlying this calculus and its relationship to string rewriting have been known since at least the the publication of [19]. It seems the basic idea of connecting natural transformations to string rewriting has been essentially rediscovered several times in different contexts, including [19], [17] in the context of concurrency, and [15] in the context of computer verification of purported commutative

diagrams involving monads, and in the development of the present work, with an eye toward proof search.

One of the contributions of this portion of the present work is to give a clean presentation of a logic of natural transformations, similar systems also termed rewriting logic, as well as an account of the models of this logic. Amazingly, the definitions given here (and the obvious theorems which follow) coincide almost exactly with those given in [17], although the definitions in the present work are simpler, due to ours being a more impoverished logic.

The more novel contribution is in the modification to the logic use string diagrams as its proof terms, which results in an efficiently computable sufficient condition for deciding the equality of two polymorphic programs by giving a sort of normal form. This in turn aids in speeding up program search since the search space is made quite a bit smaller. That is, we demonstrate the usefulness of string diagrams for reasoning about the equivalence of (and potentially optimizing) polymorphic programs, and their utility in program synthesis.

We first give a presentation of the calculus \mathcal{N} which is easily motivated by the goal at hand (constructing natural transformations) and then modify its rules to obtain a calculus more suitable for proof search.

Each rule is justified by operations that can be performed to obtain natural transformations of endofunctors.

Let A, B, C, \dots be symbols intended to range over endofunctors on some category \mathcal{C} . Juxtaposition of these symbols should be thought of as composition of functors. I.e, AB should be interpreted as the composite $A \circ B$. The sequents for this calculus are of the form

$$A_1 \cdots A_n \rightarrow B_1 \cdots B_m$$

which will be interpreted as the type of natural transformations from $A_1 \cdots A_n$ to $B_1 \cdots B_m$. A proof of such a sequent can be thought of as a (description of a) natural transformation from $A_1 \cdots A_n$ to $B_1 \cdots B_m$.

The rules of the calculus \mathcal{N} are

$$\frac{}{A \rightarrow A} \text{ID}$$

$$\frac{B \rightarrow B'}{AB \rightarrow AB'} \text{FUNCTOR}$$

$$\frac{A \rightarrow A'}{AB \rightarrow A'B} \text{COMPONENT}$$

$$\frac{A \rightarrow B \quad B \rightarrow C}{A \rightarrow C} \text{CUT}$$

Note that in the pure system \mathcal{N} as presented, the only derivable sequents are of the form $X \rightarrow X$. So, to the pure system we add a collection of axioms \mathcal{A} , each of the form

$$\frac{}{S \rightarrow T} \ell$$

for some fixed S and T and for some label ℓ . Concretely, we'll imagine that each axiom is a tuple (ℓ, S, T) . We denote the resulting proof calculus $\mathcal{N}[\mathcal{A}]$.

The CUT rule seems to hurt the possibility of proof search in $\mathcal{N}[\mathcal{A}]$ since B cannot be inferred from the conclusion sequent and could in principle be any of infinitely many strings. To remedy this, we add the following rules for each $(\ell, S, T) \in \mathcal{A}$.

$$\frac{X \rightarrow ASB \quad \frac{}{S \rightarrow T} \ell}{X \rightarrow ATB} \text{REWRITE-}\ell\text{-R}$$

$$\frac{\frac{}{S \rightarrow T} \ell \quad ATB \rightarrow Y}{ASB \rightarrow Y} \text{REWRITE-}\ell\text{-L}$$

We will see later on that only one of these rules is necessary for each (ℓ, S, T) .

Note that these rules are essentially special cases of CUT:

$$\frac{X \rightarrow ASB \quad \frac{\frac{\frac{}{S \rightarrow T} \ell}{SB \rightarrow TB} \text{COMPONENT}}{ASB \rightarrow ATB} \text{FUNCTOR}}{X \rightarrow ATB} \text{CUT}$$

$$\frac{\frac{\frac{\frac{}{S \rightarrow T} \ell}{SB \rightarrow TB} \text{COMPONENT}}{ASB \rightarrow ATB} \text{FUNCTOR} \quad ATB \rightarrow Y}{ASB \rightarrow Y} \text{CUT}}$$

and so their addition does not alter the set of provable sequents, although it does of course alter the collection of proofs.

2.3 A semantics in categories

The presentation of $\mathcal{N}[\mathcal{A}]$ with the rules REWRITE- ℓ -R and REWRITE- ℓ -L and without cut is suitable for search assuming \mathcal{A} is finite. This is because given a goal sequent $\mathcal{X} \rightarrow \mathcal{Y}$ there are only finitely many rules from which it could be derived. This gives us a naïve strategy: at each stage, simply try each of the finitely many rules in a depth first manner.

However, it is not immediately clear a priori that removing cut and replacing it with the REWRITE- ℓ -L and REWRITE- ℓ -R rules preserves the set of derivable

sequents. Furthermore, since we care about the actual natural transformations corresponding to proofs of sequents, it is not clear that any natural transformation constructible using CUT is derivable using the other rules.

To that end, we develop a bit of theory connecting the proof system to both natural transformations and string rewriting systems.

Definition 1. Fix an alphabet Σ and let \mathcal{A} be a collection of axioms (ℓ, S, T) with $S, T \in \Sigma^*$.

A model of \mathcal{A} is

1. A category \mathcal{C}

2. For each $A_i \in \Sigma$ a functor $\llbracket A_i \rrbracket : \mathcal{C} \rightarrow \mathcal{C}$.

As a notational convenience, for a string $A_1 \cdots A_n \in \Sigma^*$ we define $\llbracket A_1 \cdots A_n \rrbracket := \llbracket A_1 \rrbracket \circ \cdots \circ \llbracket A_n \rrbracket$.

3. For each $(\ell, S, T) \in \mathcal{A}$ a natural transformation $\llbracket \ell \rrbracket : \llbracket S \rrbracket \rightarrow \llbracket T \rrbracket$.

We use the notation

$$\mathcal{C} \models \mathcal{A}$$

to mean that there merely exist endofunctors on \mathcal{C} and natural transformations making \mathcal{C} a model of \mathcal{A} .

If we were after generality, we could have defined a model as an arbitrary 2-category on one object, but we specialize the definition for simplicity.

For our applications, the model to keep in mind is **Hask**, the category of Haskell types and functions. Here, for $(\ell, S, T) \in \mathcal{A}$, the natural transformation $\llbracket \ell \rrbracket$ will typically be represented a value of type $\forall \alpha. \llbracket S \rrbracket \alpha \rightarrow \llbracket T \rrbracket \alpha$.

Proposition 2. If p is a proof $\mathcal{N}[\mathcal{A}] \vdash S \rightarrow S'$ and $\mathcal{C} \models \mathcal{A}$, then there is a natural transformation $\llbracket p \rrbracket : \llbracket S \rrbracket \rightarrow \llbracket S' \rrbracket$.

Proof. By induction on the structure of p .

- ID

$$\frac{}{A \rightarrow A} \text{ID}$$

Take $\llbracket p \rrbracket = \text{id}_{\llbracket A \rrbracket}$.

- FUNCTOR

$$\frac{B \rightarrow B'}{AB \rightarrow AB'} \text{FUNCTOR}$$

Let q be the proof $B \rightarrow B'$. By induction we have $\llbracket q \rrbracket : \llbracket B \rrbracket \rightarrow \llbracket B' \rrbracket$. Now take $\llbracket p \rrbracket = \llbracket A \rrbracket \llbracket q \rrbracket$.

- COMPONENT

$$\frac{A \rightarrow A'}{AB \rightarrow A'B} \text{ COMPONENT}$$

Let q be the proof that $A \rightarrow A'$. By induction we have $\llbracket q \rrbracket : A \rightarrow A'$. Now take $\llbracket p \rrbracket = \llbracket q \rrbracket \llbracket B \rrbracket$.

- CUT

$$\frac{A \rightarrow B \quad B \rightarrow C}{A \rightarrow C} \text{ CUT}$$

Let q be the proof of $A \rightarrow B$ and r the proof of $B \rightarrow C$. Now take $\llbracket p \rrbracket = \llbracket r \rrbracket \circ \llbracket q \rrbracket$.

- Axiom ℓ

$$\frac{}{S \rightarrow T} \ell$$

Take $\llbracket p \rrbracket = \llbracket \ell \rrbracket$.

□

2.4 A semantics in string rewriting (a special case)

Let \mathcal{A} be a set of axioms over the alphabet Σ . Consider the semi-Thue system $\rightarrow_{\mathcal{A}}$ generated by the rewrite rules

$$S \rightarrow_{\mathcal{A}} T$$

for each $(-, S, T) \in \mathcal{A}$. That is, $\rightarrow_{\mathcal{A}}$ is the least relation on Σ^* satisfying

- $S \rightarrow_{\mathcal{A}} T$ for each $(-, S, T) \in \mathcal{A}$.
- If $A \rightarrow_{\mathcal{A}} B$, then for any $X, Y \in \Sigma^*$ $XAY \rightarrow_{\mathcal{A}} XBY$.
- For any $X \in \Sigma^*$, $X \rightarrow_{\mathcal{A}} X$.
- If $A \rightarrow_{\mathcal{A}} B$ and $B \rightarrow_{\mathcal{A}} C$, then $A \rightarrow_{\mathcal{A}} C$.

The idea is we start with some string A , and then we find a substring S of A such that S is the right hand side of some axiom rule $S \rightarrow_{\mathcal{A}} T$, and we then replace the substring S with T in A to obtain a new string A' .

There is a obviously a strong resemblance to $\mathcal{N}[\mathcal{A}]$. What we will show is that the set of sequents $X \rightarrow Y$ deducible from a set of axioms \mathcal{A} is exactly the set of rewrite rules $X \rightarrow_{\mathcal{A}} Y$.

We can build a very syntactic model structure for \mathcal{A} by considering the preorder category on $(\mathcal{A}, \rightarrow_{\mathcal{A}})$, denoted $\text{Pre}(\mathcal{A})$. Explicitly, this is the category whose objects are the strings in Σ^* and where

$$\text{Hom}(X, Y) = \begin{cases} \{ * \} & X \rightarrow_{\mathcal{A}} Y \\ \emptyset & \text{otherwise} \end{cases}$$

This just encodes the partial order $\rightarrow_{\mathcal{A}}$ as a category.

$\text{Pre}(\mathcal{A})$ can be made a model for \mathcal{A} as follows.

- For $A_i \in \Sigma$ define

$$\llbracket A_i \rrbracket(X) := A_i X$$

i.e., the string whose head is A_i and whose tail is X . This gives a functor iff $\text{Hom}(\llbracket A_i \rrbracket(X), \llbracket A_i \rrbracket(Y))$ is inhabited whenever $\text{Hom}(X, Y)$ is. But this holds since

$$\begin{aligned} \text{Hom}(X, Y) \text{ inhabited} &\implies X \rightarrow_{\mathcal{A}} Y \\ &\implies A_i X \rightarrow_{\mathcal{A}} A_i Y \\ &\implies \text{Hom}(A_i X, A_i Y) \text{ inhabited} \end{aligned}$$

- For $(\ell, S, T) \in \mathcal{A}$, we must provide

$$\llbracket \ell \rrbracket : \llbracket S \rrbracket \rightarrow \llbracket T \rrbracket$$

Given the definition of $\llbracket A_i \rrbracket$, it is clear that $\llbracket S \rrbracket$ is the map prepending S to its argument and similarly for $\llbracket T \rrbracket$. So, to construct $\llbracket \ell \rrbracket$, for each $X \in \Sigma^*$, we must exhibit

$$\llbracket \ell \rrbracket_X : \text{Hom}(SX, TX)$$

with naturality following from uniqueness of morphisms in $\text{Pre}(\mathcal{A})$. $\text{Hom}(SX, TX)$ is inhabited because

$$\begin{aligned} (\ell, S, T) \in \mathcal{A} &\implies S \rightarrow_{\mathcal{A}} T \\ &\implies SX \rightarrow_{\mathcal{A}} TX \\ &\implies \text{Hom}(SX, TX) \text{ inhabited} \end{aligned}$$

so we are done.

In fact, what we have proved here is half of the claim that

Lemma 3. $S \longrightarrow_{\mathcal{A}} S'$ iff there is a natural transformation $\llbracket S \rrbracket \rightarrow \llbracket S' \rrbracket$.

Proof. We proved the forward direction above. The other direction of the implication follows from taking the component of the given natural transformation at the empty string. \square

Corollary 4. If $\mathcal{N}[\mathcal{A}] \vdash S \rightarrow S'$, then $S \longrightarrow_{\mathcal{A}} S'$.

Proof. Suppose $\mathcal{N}[\mathcal{A}] \vdash S \rightarrow S'$. By Proposition 2, there is a natural transformation $\llbracket S \rrbracket \rightarrow \llbracket S' \rrbracket$. By Lemma 3, $S \longrightarrow_{\mathcal{A}} S'$. \square

Proposition 5. If $S \longrightarrow_{\mathcal{A}} S'$, then $\mathcal{N}[\mathcal{A}] \vdash S \rightarrow S'$.

Proof. There are two proofs of this fact that we would like to describe with tedious details ommitted. That is, I will just give a sketch of the two proofs. They differ in whether one inducts backwards or forwards. The first builds a proof using only the REWRITE- ℓ -L rules and ID to get a “right-leaning” proof tree and the second only the REWRITE- ℓ -R rules and ID to get a “left-leaning” proof tree. They essentially correspond to different ways of “parenthesizing”.

We can imagine $S \longrightarrow_{\mathcal{A}} S'$ was derived by starting with the string S and then applying some sequence of axiom rewrites to S to get a sequence

$$S = S_0 \longrightarrow_{\mathcal{A}} S_1 \longrightarrow_{\mathcal{A}} \cdots \longrightarrow_{\mathcal{A}} S_n = S'$$

where each S_{i+1} is obtained from S_i by rewriting a substring using a single axiom (ℓ_i, A_i, B_i) in \mathcal{A} by the rule

$$S_i = X_i A_i Y_i \longrightarrow_{\mathcal{A}} X_i B_i Y_i = S_{i+1}$$

Now the two proofs go as follows.

(1) If $S = S'$, we take our proof to be

$$\frac{}{A \rightarrow A} \text{ID}$$

Otherwise, recalling that $S = S_0 = X_0 A_0 Y_0$ and $S_1 = X_0 B_0 Y_0$, we take our proof to be

$$\frac{\frac{}{A_0 \rightarrow B_0} \ell \quad A_0 B_0 Y_0 \rightarrow Y}{X_0 A_0 Y_0 \rightarrow S'} \text{REWRITE-}\ell\text{-L}$$

where the proof of $A_0 B_0 Y_0 \rightarrow Y$ is obtained in this way by induction.

(2) If $S = S'$, again we use the ID rule.

Otherwise, recalling $S_{n-1} = X_{n-1}A_{n-1}Y_{n-1}$ and $S_n = S' = X_{n-1}B_{n-1}Y_{n-1}$, we take our proof to be

$$\frac{S \rightarrow X_{n-1}A_{n-1}Y_{n-1} \quad \frac{A_{n-1} \rightarrow B_{n-1}}{\ell}}{\text{REWRITE-}\ell\text{-R}}}{S \rightarrow X_{n-1}B_{n-1}Y_{n-1}}$$

where the proof of $S \rightarrow X_{n-1}A_{n-1}Y_{n-1}$ is obtained in this way by induction. □

Corollary 6 (Cut elimination). *If $\mathcal{N}[\mathcal{A}] \vdash S \rightarrow S'$ with a proof p , then there is a proof q of $S \rightarrow S'$ only using the rules REWRITE- ℓ -R such that $\llbracket p \rrbracket = \llbracket q \rrbracket$ in any model. There is also such a proof of $S \rightarrow S'$ only using the rules REWRITE- ℓ -L.*

Proof. By Proposition 4, we obtain a sequence of rewrites $S \rightarrow_{\mathcal{A}} S'$. The two constructions going the other way then provide the proofs of $S \rightarrow S'$ desired. Using the definitions of REWRITE- ℓ -L and REWRITE- ℓ -R in terms of cut in section 2.2 and the fact that functors preserve composition, we obtain the desired equalities. □

2.5 Completeness

We can now state and prove a completeness property for the system $\mathcal{N}[\mathcal{A}]$ expressing the following informal claim: “any natural transformation constructible starting from \mathcal{A} in a generic category is derivable in $\mathcal{N}[\mathcal{A}]$ ”.

Theorem 7 (Soundness and completeness). *$\mathcal{N}[\mathcal{A}] \vdash S \rightarrow S'$ iff every model of \mathcal{A} is a model of $S \rightarrow S'$.*

Proof.

1. \implies (Soundness)

Suppose $\mathcal{N}[\mathcal{A}] \vdash S \rightarrow S'$ and $\mathcal{C} \models \mathcal{A}$. Then by Proposition 2, there is a natural transformation $\llbracket S \rrbracket \rightarrow \llbracket S' \rrbracket$. Thus, \mathcal{C} models $S \rightarrow S'$.

2. \impliedby (Completeness)

Suppose every model of \mathcal{A} is a model of $S \rightarrow S'$. Then in particular, $\text{Pre}(\mathcal{A})$ is a model of $S \rightarrow S'$. This means we have a natural transformation $\llbracket S \rrbracket \rightarrow \llbracket S' \rrbracket$, which by Lemma 3 implies $S \rightarrow_{\mathcal{A}} S'$. Thus by Proposition 5, $\mathcal{N}[\mathcal{A}] \vdash S \rightarrow S'$. □

Essentially this fact can be found in [17].

This completeness theorem is frankly inadequate from the perspective of program search, where possibly most sequents $F \rightarrow G$ are inhabited and what we really care about is the structure of the collection of all proofs of the sequent. That is, we would want a completeness theorem which tells us that all natural transformations definable over a base set in a *generic* category are definable in $\mathcal{N}[\mathcal{A}]$, and two “generic natural transformations” are equal exactly when their proofs are equal in $\mathcal{N}[\mathcal{A}]$.

Unfortunately, there can be no such theorem about $\mathcal{N}[\mathcal{A}]$ since $\mathcal{N}[\mathcal{A}]$ does not really represent the collection of natural transformations freely generated over \mathcal{A} . There is too much redundancy. That is, there are distinct proofs in $\mathcal{N}[\mathcal{A}]$ whose interpretations are equal in every model. The exact nature of this redundancy and what can be done about it will be analyzed in the following sections.

2.6 Some proof theoretic observations

As mentioned, the system as presented some redundancy on the level of proofs. That is to say, there are distinct proofs trees which yield the same natural transformation under any interpretation. Suppose we have functors A, B, C, D and $\eta : A \rightarrow C$, $\varphi : B \rightarrow D$ and consider, for example the following four proofs:

$$\frac{\frac{\varphi B \rightarrow D}{A\varphi : AB \rightarrow AD} \text{ FUNCTOR} \quad \frac{\eta : A \rightarrow C}{\eta D : AD \rightarrow CD} \text{ COMPONENT}}{\eta D \circ A\varphi : AB \rightarrow CD} \text{ CUT}$$

$$\frac{\frac{\eta : A \rightarrow C}{\eta B : AB \rightarrow CB} \text{ COMPONENT} \quad \frac{\varphi : B \rightarrow D}{C\varphi : CB \rightarrow CD} \text{ FUNCTOR}}{C\varphi \circ \eta B : AB \rightarrow CD} \text{ CUT}$$

By naturality, $\eta D \circ A\varphi = C\varphi \circ \eta B$. To avoid this redundancy, we could add this derivation of horizontal composition as a primitive inference rule.

$$\frac{\eta : A \rightarrow C \quad \varphi : B \rightarrow D}{C\varphi \circ \eta B : AB \rightarrow CD} \text{ ZIP}$$

The ZIP rule is actually quite nice as it allows us to derive both the FUNCTOR and COMPONENT rules, which could then be removed as primitives.

$$\frac{\overline{A \rightarrow A} \text{ ID} \quad B \rightarrow B'}{AB \rightarrow AB'} \text{ ZIP}$$

$$\frac{A \rightarrow A' \quad \overline{B \rightarrow B} \text{ ID}}{AB \rightarrow A'B} \text{ ZIP}$$

There is also the fact that functors preserve composition, so for $\varphi : B \rightarrow C$ and $\eta : C \rightarrow D$, we should have an equality between the two proofs

$$\frac{\frac{\varphi : B \rightarrow C}{A\varphi : AB \rightarrow AC} \text{FUNCTOR} \quad \frac{\eta : C \rightarrow D}{A\eta : AC \rightarrow AD} \text{FUNCTOR}}{A\eta \circ A\varphi : AB \rightarrow AD} \text{CUT}$$

and

$$\frac{\frac{\varphi : B \rightarrow C \quad \eta : C \rightarrow D}{\eta \circ \varphi : B \rightarrow D} \text{CUT}}{A(\eta \circ \varphi)AB \rightarrow AD} \text{FUNCTOR}$$

This latter equality could be thought of as reduction rule from the first proof to the second, since it can be carried out using only local information and reduces the size of the proof. Interpreting these derivations as programs, in typical cases it also corresponds to an optimization analogous to the list fusion optimization implemented in GHC[10]. That is, $A(\eta \circ \varphi)$ would make one pass over an A data structure rather than the two passes $A\eta \circ A\varphi$ might make.

2.7 String diagrams for efficient search

It would be an interesting future direction of research to investigate the possibility of defining confluent or even normalizing reduction rules on proofs in $\mathcal{N}[\mathcal{A}]$ (or a modified system), but the problem can be essentially side-stepped by translating proofs in $\mathcal{N}[\mathcal{A}]$ into string diagrams. The situation is reminiscent of that of linear logic, where translating linear logic tree-proofs into proof nets [8] eliminates computationally irrelevant distinctions between proofs.

To motivate string diagrams, let us consider a program for randomly selecting a sublist of a list. Suppose we have a monad `Random` with values

```
randomBool :: Random Bool
runRandom :: Random a -> IO a
```

Define

```
randomlyNothing :: a -> Random a
randomlyNothing x =
  fmap (\b -> if b then Just x else Nothing) randomBool
```

which `randomlyNothing` ignores its argument. Now define

```
randomSublist :: [a] -> IO [a]
randomSublist = runRandom . fmap catMaybes . sequence . map randomlyNothing
```

which takes a list to a random sublist by randomly sending each element x in the list to either `Just x` or `Nothing` and then picks out all the elements x which were sent to `Just x`.

In the language of natural transformations, `randomlyNothing :: a -> Random (Maybe a)` can be thought of a natural transformation from the identity functor on `Hask` to the composite `Random o Maybe`. Thinking in this way, we can draw `randomSublist` as a rather uninspiring 2-diagram (in `Cat`, the 2 category of categories) (Figure 2.1).

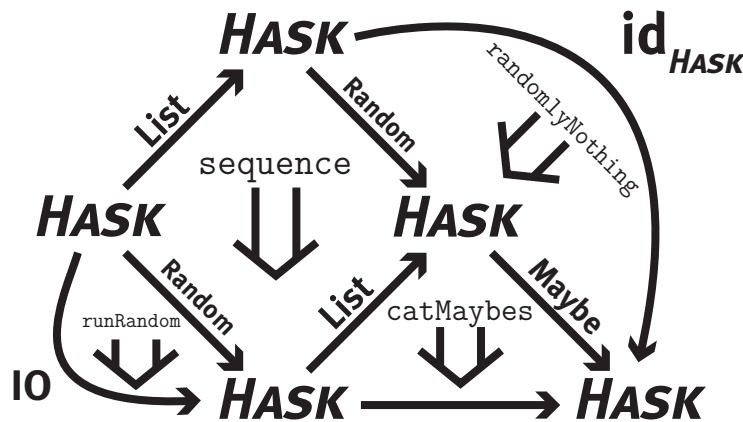
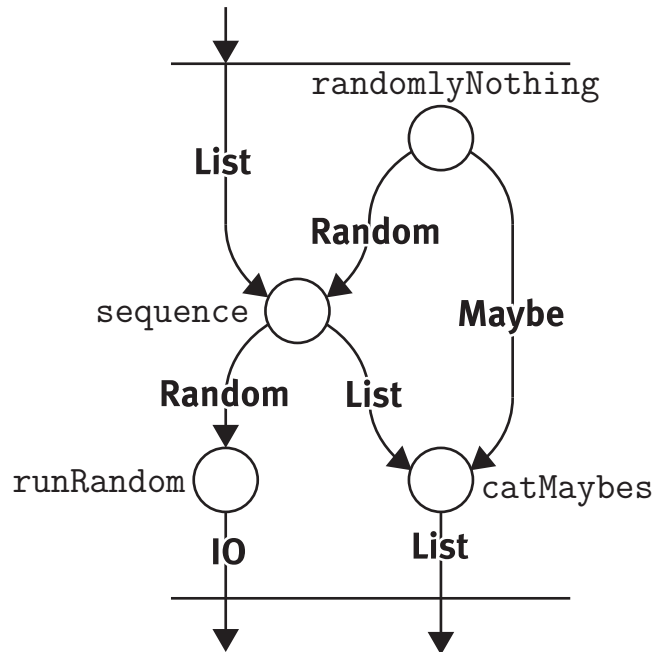


Figure 2.1: A diagram of `randomlyNothing`

This representation makes it hard to see the flow of effects in our program and is also quite cluttered. We can draw the same information in a more manageable way, which lets us more easily imagine effects transforming into other effects, by essentially taking the dual graph to Figure 2.1, as in Figure 2.2. Here, functors are represented by arrows and polymorphic functions by points (drawn as circles). Would-be straight lines corresponding to the identity functor are omitted. Hence, for example, `randomlyNothing` in Figure 2.2 has outgoing arrows but none incoming. Such a diagram is called a string diagram. Here we think of functors as “effects” (e.g., IO, partiality, non-determinism, randomness) and a string diagram shows us how a program translates effects into other effects through polymorphic functions.

A question which arises is “Have we lost any information by translating our program into a string diagram?” That is, is it possible to recover a program from the string diagram which it corresponds to? At first blush the answer seems to be no since there are two distinct programs which have the same string diagram. For example, the two programs

```
randomSublist, randomSublist' :: [a] -> IO [a]
randomSublist =
  runRandom . fmap catMaybes . sequence . map randomlyNothing
```

Figure 2.2: A string diagram of `randomSublist`

```
randomSublist' =
  fmap catMaybes . runRandom . sequence . map randomlyNothing
```

get translated into the same string diagram, namely the one in Figure 2.2. However, the free theorem (naturality) for `forall a. Random a -> IO a` tells us that `runRandom . fmap catMaybes` is indistinguishable from `fmap catMaybes . runRandom` [25]. So although the two programs are not literally the same term, they are observationally indistinguishable². In general, it is true that string diagrams faithfully represent terms identified up to extensional equality (i.e., identifying functions if they agree on all arguments). For details, see [11]. As mentioned in the first chapter, we exploit the fact that string diagrams represent programs up to naturality (and functoriality) equations and not literally to eliminate duplicates from lists of synthesized programs.

In general, string diagrams are a graphical calculus which provide a convenient and easy to reason about notation for describing morphisms in a monoidal category. A monoidal category is a category \mathcal{C} with a functor $\otimes : \mathcal{C} \rightarrow \mathcal{C} \rightarrow \mathcal{C}$, associative up to isomorphism, an object $1 \in \mathcal{C}$ which is a left and right identity for \otimes up to isomorphism, and such that some coherence conditions between these isomorphisms. For more details, see [11].

²As mentioned, in real Haskell, due to the interaction of nontermination with laziness, such naturality equations do not always hold, but in this work as a simplification we assume their validity.

String diagrams were first introduced by Kelly and Laplaza in [14] and put on a solid formal basis by Joyal and Street [11] [12] who proved the correctness of reasoning using string diagrams up to isotopy.

Formally, a string diagram is a directed graph with

- Two distinct vertices called the “top” and “bottom” vertices such that the top vertex has no incoming edges and the bottom vertex has no outgoing edges. These two vertices will be called “special” vertices.
- A downward embedding γ of the graph into the plane.

An downward embedding (hereafter, simply called an “embedding”) γ of a graph is a collection of smooth paths $\gamma_e : [0, 1] \rightarrow \mathbb{R}^2$, one for each edge e of the graph such that

- For any e_1, e_2 , e_1 and e_2 have the same source iff $\gamma_{e_1}(0) = \gamma_{e_2}(0)$.
- For any e_1, e_2 , e_1 and e_2 have the same sink iff $\gamma_{e_1}(1) = \gamma_{e_2}(1)$.
- For any e_1, e_2 , the sink of e_1 is the source of e_2 iff we have $\gamma_{e_1}(1) = \gamma_{e_2}(0)$.
- For each e , the path γ_e travels monotonically down the plane. That is, for $t_1 < t_2$, the y coordinate of $\gamma_e(t_1)$ is greater than the y coordinate of $\gamma_e(t_2)$.
- For $e \neq e'$, γ_e and $\gamma_{e'}$ intersect only at their endpoints if at all.
- The top vertex of the graph is embedded above all other vertices and the bottom vertex of the graph is embedded below all other vertices.

with isotopic embeddings identified. For full details see [11].

As a notational convenience, we will use $\gamma(p)$ to refer to the image of the whole graph as a subspace of the plane, $\gamma(e)$ for the image of an edge e , and $\gamma(v)$ for the image of a vertex v .

Often, we imagine that the top and bottom vertices are pulled off the screen and we do not draw them. The edges leaving from the top vertex may be called the “inputs” or the “incoming edges” of the diagram and the edges entering the bottom vertex may be called the “outputs” or the “outgoing edges” of the diagram.

The embedding of the graph induces an ordering on the incoming edges at any given vertex. Simply order them from left to right. Likewise an ordering on the outgoing edges is induced in the same way. Obviously representing an embedding of the graph in the plane is untenable from the perspective of efficient implementation, so in our implementation we store only the ordering of the edges incident to every vertex. The original embedding is determined up to isotopy by this data.

In our context, the only example of a monoidal category we will consider is the following.

Definition 8. Let $\text{End}(\mathcal{C})$ be the category whose objects are endofunctors on \mathcal{C} , i.e., functors $\mathcal{C} \rightarrow \mathcal{C}$, and whose maps are natural transformations.

Composition of endofunctors $\circ : \text{End}(\mathcal{C}) \rightarrow \text{End}(\mathcal{C}) \rightarrow \text{End}(\mathcal{C})$ is a monoid operation for $\text{End}(\mathcal{C})$ with on-the-nose identity $\text{id}_{\mathcal{C}}$. The action of \circ on arrows is given by horizontal composition of natural transformations. That is, if we have $\eta : A \rightarrow B$ and $\varphi : C \rightarrow D$, then we define $\circ(\eta, \varphi) : A \circ C \rightarrow B \circ D$ by

$$B\varphi \circ \eta C$$

(or alternatively, $\eta D \circ A\varphi$, which is the same map as discussed above). Figure 2.2 shows a string diagram in the monoidal category $\text{End}(\text{Hask})$.

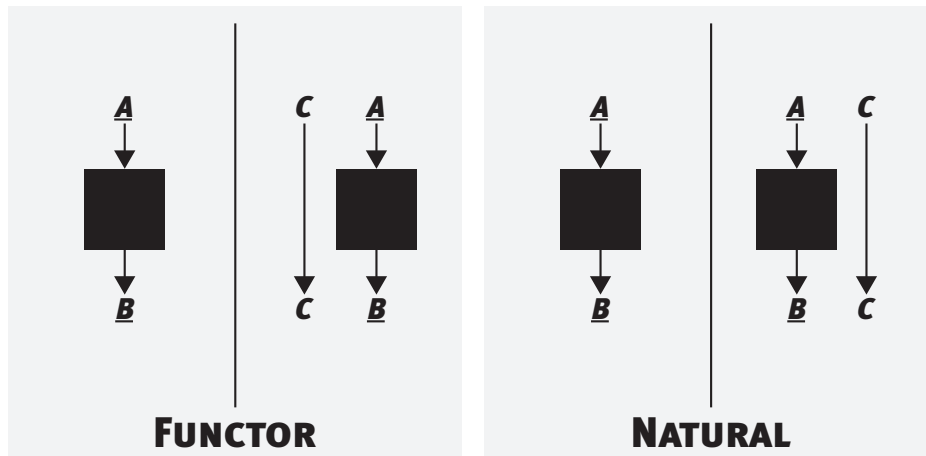


Figure 2.3: Translations for the remaining rules of \mathcal{N}

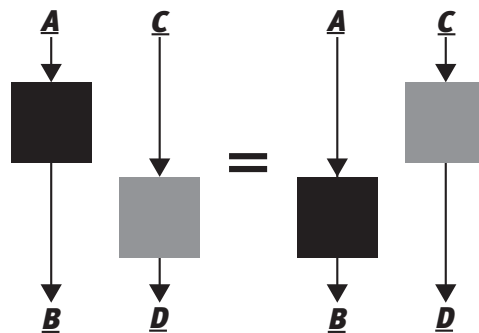


Figure 2.4: Naturality evidently holds for string diagrams

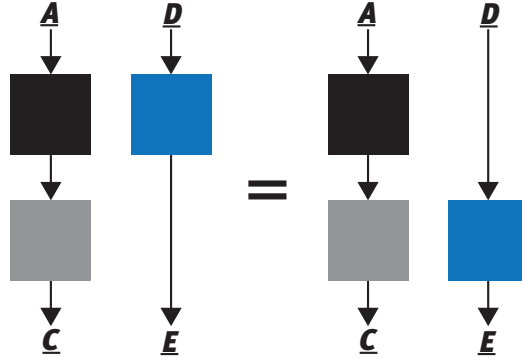


Figure 2.5: Vertical (cut) and horizontal (juxtapose, or zip) composition evidently commute for string diagrams

2.8 String diagrams for eliminating redundancy

In this section, we give a calculus of string diagrams and prove a 2-dimensional or proof relevant completeness theorem.

Fix an alphabet Σ and a set of axioms \mathcal{A} as in the definition of \mathcal{N} . I.e., \mathcal{A} is a set of triples of the form (ℓ, S, T) with $S, T \in \Sigma^*$.

Let us call the calculus that we are defining $\mathcal{G}[\mathcal{A}]$. The theorems of $\mathcal{G}[\mathcal{A}]$ are, just as in $\mathcal{N}[\mathcal{A}]$, statements of the form $A \rightarrow B$ for $A, B \in \Sigma^*$. A proof of $A \rightarrow B$ is a string diagram obtained according to the rules depicted in Figure 2.6. In words, we inductively define the proofs of $\mathcal{G}[\mathcal{A}]$ as follows. In general, a proof in $\mathcal{G}[\mathcal{A}]$ of $A_1 \cdots A_n \rightarrow B_1 \cdots B_m$ (for $A_i, B_j \in \Sigma$) will be a string diagram whose incoming ports are labelled by the A_i and whose outgoing ports are labelled by the B_j .

- ID

For each $A \in \Sigma$, $\mathcal{G}[\mathcal{A}]$ proves $A \rightarrow A$ with proof given by the string diagram with one incoming port labelled by A , one outgoing port labelled by A , and a single edge connecting them. The embedding is the only possible one (up to isotopy) and is shown in Figure 2.6.

- ℓ

For each $(\ell, S, T) \in \mathcal{A}$, $\mathcal{G}[\mathcal{A}]$ proves $S \rightarrow T$ with the proof given by the string diagram in Figure 2.6. In words, we can describe the diagram as follows.

If $S = A_1 \cdots A_n$ and $T = B_1 \cdots B_m$, $A_i, B_j \in \Sigma$, then our diagram has a single vertex labelled by ℓ , incoming ports labelled by A_1, \dots, A_n and ordered in that way, each with an edge into the single vertex. The outgoing ports are labelled B_1, \dots, B_m , ordered in that way and each has an edge from the single vertex.

- CUT

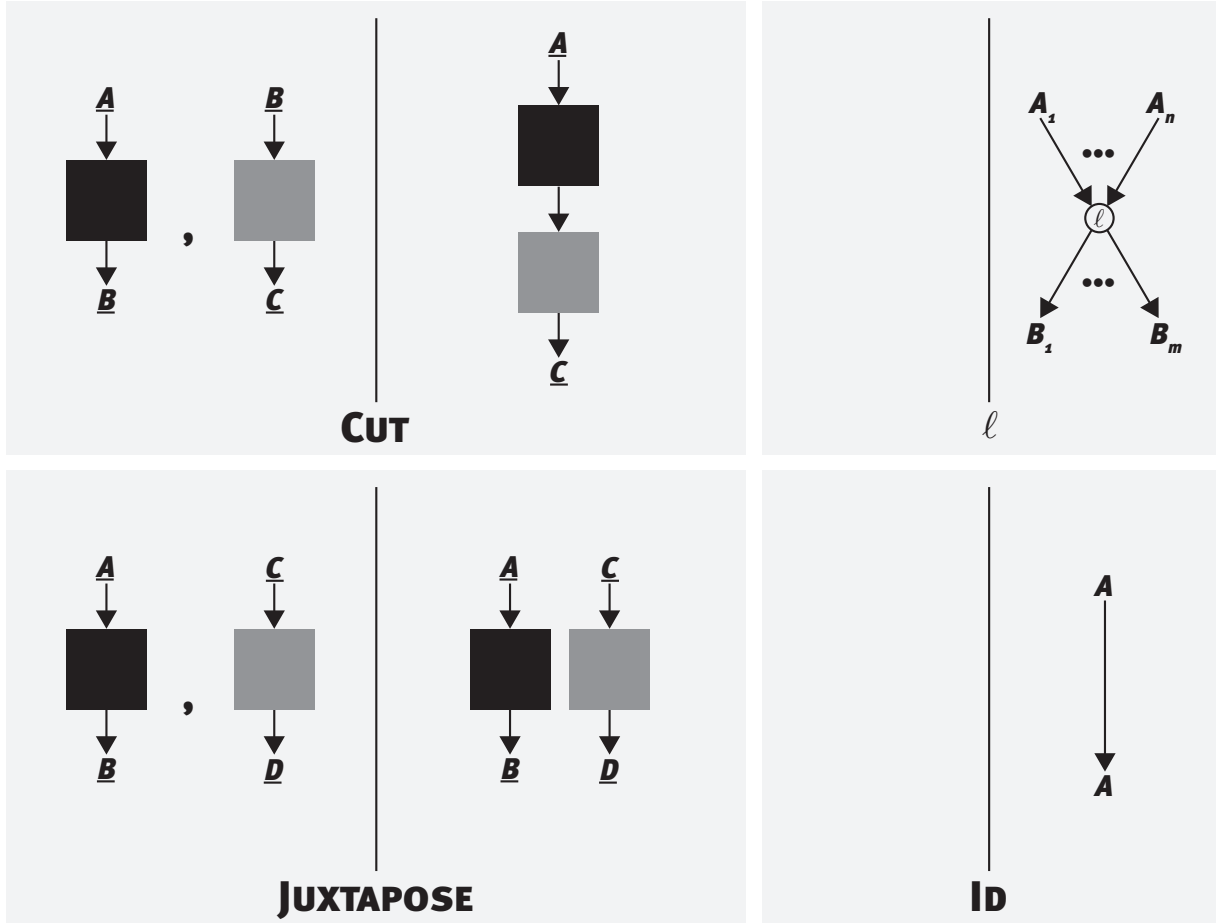


Figure 2.6: The essential rules for our string calculus

If p is a proof of $\underline{A} \rightarrow \underline{B}$ and q is a proof of $\underline{B} \rightarrow \underline{C}$, then we obtain a proof of $\underline{A} \rightarrow \underline{C}$ in the obvious way by joining up the outgoing ports of p to the incoming ports of q .

- **JUXTAPOSE**

If p is a proof of $\underline{A} \rightarrow \underline{B}$ and q is a proof of $\underline{C} \rightarrow \underline{D}$, then we obtain a proof of $\underline{AC} \rightarrow \underline{BD}$ by juxtaposing p and q left-to-right.

For p a proof of $X \rightarrow Y$ in $\mathcal{G}[\mathcal{A}]$ and for \mathcal{C} a model of \mathcal{A} , define $\llbracket p \rrbracket \in \text{Hom}_{\text{End}(\mathcal{C})}(\llbracket X \rrbracket, \llbracket Y \rrbracket)$ (a natural transformation from $\llbracket X \rrbracket$ to $\llbracket Y \rrbracket$) inductively as follows.

If p was obtained using **CUT** by attaching $p_0 : X \rightarrow X'$ to $p_1 : X' \rightarrow Y$, define $\llbracket p \rrbracket = \llbracket p_1 \rrbracket \circ \llbracket p_0 \rrbracket$.

If p was obtained using **JUXTAPOSE** by juxtaposing $p_0 : A \rightarrow B$ with $p_1 : C \rightarrow D$, define $\llbracket p \rrbracket$ to be the horizontal composition of p_0 and p_1 , $\llbracket p_0 \rrbracket_D \circ A \llbracket p_1 \rrbracket$.

If p was obtained using **ID**, then $\llbracket p \rrbracket = \text{id}$.

If p was obtained using the ℓ rule, then $\llbracket p \rrbracket = \llbracket \ell \rrbracket$.

Theorem 9 (2 dimensional completeness). *Suppose that p, q are proofs of $\underline{A} \rightarrow \underline{B}$ in $\mathcal{G}[\mathcal{A}]$ and that for any model \mathcal{C} we have $\llbracket p \rrbracket = \llbracket q \rrbracket$. Then $p = q$.*

Proof. Essentially, this says that our proof system of string diagrams is a free category of some sort. We prove this by constructing a special model. Define the category \mathcal{Z} (\mathcal{Z} for “zebra”) as follows.

- The objects of \mathcal{Z} are the strings over the alphabet Σ .
- $\text{Hom}_{\mathcal{Z}}(X, Y)$ is the set of proofs of $X \rightarrow Y$ in $\mathcal{G}[\mathcal{A}]$. Composition is given by the CUT rule as in Figure 2.6.

Now we define a model structure on \mathcal{Z} .

- For $A_i \in \Sigma$, $\llbracket A_i \rrbracket : \mathcal{Z} \rightarrow \mathcal{Z}$ is given by $\llbracket A_i \rrbracket(X) = A_i X$. I.e., the string whose head is A_i and whose tail is X . The action on morphisms is illustrated in Figure 2.7.

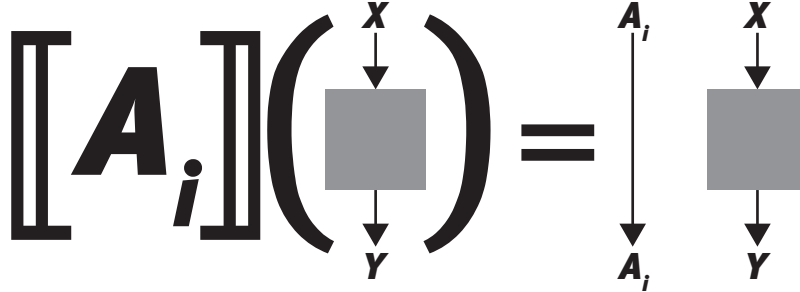


Figure 2.7: The action of $\llbracket A_i \rrbracket$ on morphisms

With this definition, for $\underline{A}, \underline{B} \in \Sigma^*$, a natural transformation $n : \llbracket \underline{A} \rrbracket \rightarrow \llbracket \underline{B} \rrbracket$ satisfies the equation $n_Y \circ \llbracket \underline{A} \rrbracket f = \llbracket \underline{B} \rrbracket f \circ n_X$. Figure 2.8 shows the meaning of this equation on the level of string diagrams.

By Figure 2.8, since we can push around the blue and gray boxes so that they’re all at the same height, the string diagram labelled n_X is the same as the string diagram labelled n_Y . Thus n is determined by this single diagram and for any X , $n_X : \text{Hom}_{\mathcal{Z}}(\llbracket \underline{A} \rrbracket X, \llbracket \underline{B} \rrbracket X)$ is just this single diagram juxtaposed with the straight line ID diagram $X \rightarrow X$ on the right. Moreover, a map defined in that way satisfies naturality.

- Now we can finish the definition of the model structure. For $(\ell, S, T) \in \mathcal{A}$. $\llbracket \ell \rrbracket_X$ is the diagram for ℓ in Figure 2.6 juxtaposed with the straight line ID diagram $X \rightarrow X$ on the right. By the preceding discussion, this is natural.

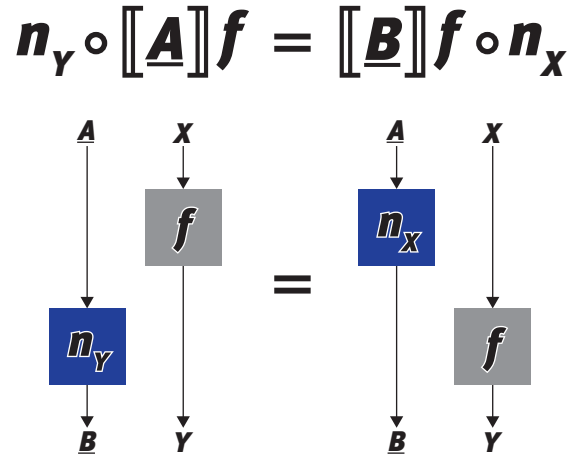


Figure 2.8: Naturality for n

By the inductive definition of $\llbracket - \rrbracket$ on all proofs (string diagrams) in $\mathcal{G}[\mathcal{A}]$, it is clear that for p a proof of $\underline{A} \rightarrow \underline{B}$, $\llbracket p \rrbracket$ is the natural transformation such that $\llbracket p \rrbracket_X : \text{Hom}_{\mathcal{Z}}(\underline{A}X, \underline{B}X)$ is p juxtaposed with the straight line $X \rightarrow X$ on the right. That is, \mathcal{Z} essentially interprets proofs as themselves.

Now we are ready to finish the proof. Let p, q be given proofs (string diagrams) of $\underline{A} \rightarrow \underline{B}$ and for any model of \mathcal{A} we have $\llbracket p \rrbracket = \llbracket q \rrbracket$. Then in particular this is true of \mathcal{Z} . But then clearly $p = q$ by the preceding paragraph. □

Proposition 10. *Any string diagram such that each vertex corresponds to an axiom in \mathcal{A} can be obtained using the rules of $\mathcal{G}[\mathcal{A}]$.*

Proof. The idea of the proof is to slide vertices which are topmost in some embedding off the page one by one. A vertex (here we exclude the top vertex) is topmost in an embedding if its image has the highest y coordinate of any vertex .

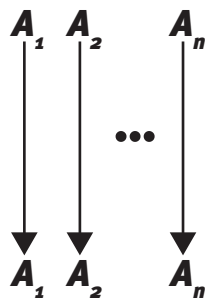


Figure 2.9: A trivial diagram

The proof that any graph can be built using the moves of $\mathcal{G}[\mathcal{A}]$ proceeds by induction on the number of internal vertices in p , sliding vertices up and out of the picture one-by-one.

If p is a straight line graph as in Figure 2.9, then p can be written as a collection of juxtaposed ID graphs and we're done.

Otherwise, p contains vertices. Let v be a vertex labelled by ℓ , such that there is some downward embedding of p with v having the highest y -coordinate. In what follows, the words left and right refer to left and right in this embedding. Say the

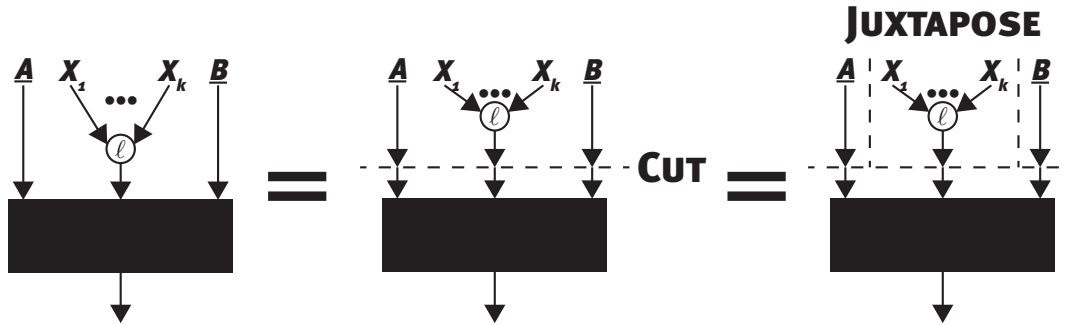


Figure 2.10: Decomposition of a string diagram

edges going into v are from the incoming ports X_1, \dots, X_k , the ports to the left of X_1 are \underline{A} and the ports to the right of X_k are \underline{B} so the graph is as depicted in Figure 2.10. Figure 2.10 then shows how we can apply **CUT** and then **JUXTAPOSE** twice to isolate v , which is now just the axiom given by the rule ℓ . By induction, we can obtain the lower half of the graph (depicted as a black box in Figure 2.10) using the rules of $\mathcal{G}[\mathcal{A}]$, and so we can obtain p using $\mathcal{G}[\mathcal{A}]$. \square

Note that this proof almost corresponds to an algorithm for turning string diagrams (whose vertices are labelled with Haskell functions terms) into actual Haskell programs since **CUT** corresponds to function composition and **JUXTAPOSE** to horizontal composition, which is $g \cdot \text{fmap } f$ if g corresponds to the diagram on the left and f the one on the right. “Almost”, because we have given no algorithm for finding a vertex which in some embedding appears with the highest y -coordinate. We will approach this problem in Section 2.11.

Together, Theorem 9 and Proposition 10 prove that a brute-force search using the rules of $\mathcal{G}[\mathcal{A}]$ will not only find all the natural transformations all the sequents which hold in every model, but will also find all the natural transformations which may be constructed in every model and will find them only “once”. Of course, naturality equations are not the only relations between Haskell programs (and because not all Haskell programs of type $\forall \alpha. F\alpha \rightarrow G\alpha$ are of the form produced by $\mathcal{G}[\mathcal{A}]$) so the search strategy is bound to miss some programs, and to find some programs twice.

Our current implementation includes an extension to string diagrams which reflect equations involving constant functors (as string diagrams reflect naturality and functoriality equations) but due to time constraints, we will not describe it here.

2.9 Remarks on implementation

In this section we describe briefly the application of the ideas of this chapter to a very simple implementation of program search, as well as some heuristics used to rank generated programs.

Suppose we want to find a string diagram of type $\underline{A} \rightarrow \underline{B}$. We proceed by trying to rewrite \underline{A} to \underline{B} by applying our axioms \mathcal{A} to substrings of \underline{A} . I.e., at step n , we will have

$$X_n = \{ p \text{ a string digram starting from } \underline{A} \mid p \text{ has } n \text{ internal vertices} \}$$

We can compute X_{n+1} from X_n by applying all the applicable rules to each element of X_n . To ensure we don't store two graphs that have the same interpretation, we define a hash function on graphs whose value depends only on the isomorphism type of the graph (and not of the particulars of how it is represented). This permits us to quickly eliminate duplicates which helps prevent duplicating efforts in search.

In our implementation, an in scope Haskell function is included in our set of axioms \mathcal{A} if after giving values for all but one of its arguments it has a type of the form

```
forall a. F a -> G a
```

If the function has n arguments, we say that as an axiom it has $n - 1$ holes (since we have to come up with values for $n - 1$ of its arguments for it to be a function of the desired type). Note that a function can be considered as an axiom in many ways, depending on which arguments we imagine putting holes in for.

We use the following heuristic for ranking discovered programs before presenting them to the user. Candidate programs are ordered lexicographically by

- (1) The total number of holes in the program.
- (2) The number of vertices in the string diagram.
- (3) The number of connected components of the string diagram.

In our usage this seems to produce a fairly good ordering of discovered terms, but more extensive testing is needed to see how useful a ranking scheme this is.

2.10 Turning string diagrams to terms

Surprisingly one of the trickiest aspects of this system of program search from an implementation point of view is turning discovered string diagrams (graphs) into Haskell terms. The algorithm we use is essentially described by Proposition 10 and proceeds by sliding vertices up and off the top of the page one by one. Each vertex corresponds to a primitive function of type $\forall\alpha.F\alpha \rightarrow G\alpha$ and the order in which they are slid off corresponds to the order they appear in the term, which is a composite of such primitives (with some number of `fmaps` applied to each primitive). Several improvements related to the fusion optimization can be made to this algorithm. So, in this section we present a modification of the algorithm implicit in Proposition 10 for converting string diagrams to Haskell terms. In what follows, *StringDiagram* will be the type of string diagrams whose vertices are labelled by Haskell terms of the appropriate types.

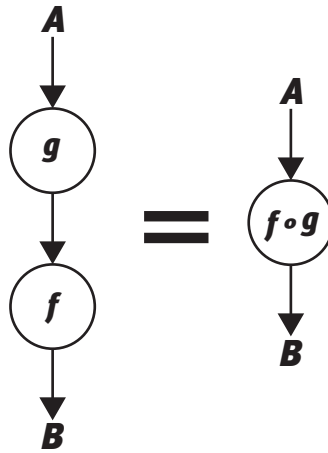


Figure 2.11: A fusion optimization on string diagrams

First, we define a preprocessing step *compressPaths* which implements a fusion optimization. Given a diagram p , let *compressPaths* p be the diagram obtained by applying the edge contraction rule illustrated in Figure 2.11. The correctness of this operation with respect to a translation into Haskell terms relies on the assumption that Haskell functors preserve composition.

We give an informal definition of a function $toTerm' : StringDiagram \rightarrow HaskellTerm$. Let p be the input diagram.

- (1) If the input diagram is a bunch of straight lines, we return `id`.
- (2) Suppose the input diagram p is a straight line on the right of a diagram p' . Then $toTerm' p = toTerm' p'$. This corresponds to turning a program of type

$\forall \alpha. F(G_2\alpha) \rightarrow G_1(G_2\alpha)$ which “doesn’t use” the G_2 into one of type $\forall \alpha. F\alpha \rightarrow G_1\alpha$. The correctness of this operation is implied by the fact that $\text{fmap id} = \text{id}$.

- (3) Suppose the input diagram p is a straight line on the left of a diagram p' . Then $\text{toTerm}' p = \text{fmap} (\text{toTerm}' p')$. See Figure 2.12 for an illustration of this case. This case is an example of a fusion optimization.

$$\text{toTerm}' \left(\begin{array}{c} \text{A} \\ \downarrow \\ \text{A} \end{array} \begin{array}{c} \text{B} \\ \downarrow \\ \square \\ \downarrow \\ \text{C} \end{array} \right) = \text{fmap}_A \left(\text{toTerm}' \left(\begin{array}{c} \text{B} \\ \downarrow \\ \square \\ \downarrow \\ \text{C} \end{array} \right) \right)$$

Figure 2.12: A fusion optimization in translating a string diagram into a term

- (4) Otherwise, choose a vertex which in some embedding is the topmost vertex, labelled by the Haskell term t , which is to the right of k lines. Let p' be the diagram obtained by sliding this vertex up “off the page”. Then define

$$\text{toTerm}' p = \text{toTerm}' p' \circ \underbrace{(\text{fmap} \circ \dots \circ \text{fmap})}_k t$$

We then take $\text{toTerm} = \text{toTerm}' \circ \text{compressPaths}$.

An actual implementation of these functions can be found (as of the time of writing) at github.com/imeckler/mote/blob/master/Search/Graph.hs.

Both compressPaths and case (3) of toTerm' implement a kind of fusion optimization, but a moment’s thought shows that each catches cases that the other misses.

2.11 Finding a topmost vertex

We now describe an approach to the problem of computing when a vertex in a string diagram is the topmost in some embedding from the combinatorial data of a string diagram.

To reiterate, the combinatorial data of a string diagram is an acyclic directed graph with, a distinguished vertex called the top vertex with in-degree 0, a distinguished vertex called the bottom vertex with out-degree 0, a linear ordering on the incoming edges, and one on the outgoing edges at every vertex.

Claim 11. *Suppose \mathcal{C} is a monoidal category such that there is only one natural transformation from $\text{id}_{\mathcal{C}}$ to itself. Suppose p is a string diagram. Let p' be the subgraph induced by restriction to the undirected connected components of the top and bottom vertices. Then p and p' both have the same interpretation in \mathcal{C} .*

Proof. Let v be any vertex in p but not p' . I.e., a vertex not connected by an undirected path to one of the special vertices. Let X be the undirected connected component of v . The topmost vertices (under the embedding γ) must have no incoming edges and the bottommost vertices must have no outgoing edges.

Thus, X is interpreted as a natural transformation from the identity to the identity. But by assumption, there is only one, namely the identity transformation. The identity can be represented by the empty diagram, so we can replace X by the empty graph. Doing this for all such connected components X amounts to restricting p to vertices connected by an undirected path to one of the special vertices. \square

We will call a string diagram where all vertices are connected by an undirected path to one of the special vertices *reduced*. The assumption of uniqueness of natural transformations $\text{id}_{\mathcal{C}} \rightarrow \text{id}_{\mathcal{C}}$ holds in our System F setting, where the only function of type $\forall \alpha. \alpha \rightarrow \alpha$ is $\lambda x. x$.

Claim 12. *Suppose v_0 is the topmost non-special vertex in some embedding γ of a reduced diagram p with incoming edges e_1, \dots, e_n ordered left to right. Then the following holds:*

- *All e_i have the top vertex as their source.*
- *For any e_i, e_{i+1} , there is no edge e such that e is to the right of e_i and e_{i+1} is to the right of e .*

Proof. Let v_0 be the topmost non-special vertex. Then any edge entering v have the top vertex as their origin. Call the edges e_1, \dots, e_n ordered left to right. Take any e_i, e_{i+1} . Consider the region bounded by e_i and e_{i+1} . Since p is reduced, there can be no vertices or edges contained in this region. In particular it is clear there can be no edge e with e to the right of e_i and e_{i+1} to the right of e since such an edge would be in this region. \square

Claim 13. *There is an algorithm for computing a vertex which in some embedding is the topmost given the combinatorial data of a non-empty, reduced string diagram p .*

Proof. Portions of the proof of correctness of this algorithm are incomplete. We will explicitly mention which in the body of the proof.

Let S be the set of vertices v such that all incoming edges of v have the top vertex as their source. Since p is non-empty, S must be non-empty. This can be seen as follows. Fix an embedding γ of p . Some vertex is topmost in this embedding. By

Claim 12, all incoming edges of v have the top vertex as their source. So, if there is any hope of finding the topmost vertex, it will be in S .

There are three cases.

- (1) There exists a $v \in S$ such that v has exactly one incoming edge.

In this case, we can clearly alter a given embedding γ to an embedding γ' in which the y coordinate of v is arbitrarily close to the y coordinate of the top vertex by dragging it along the incoming edge. Thus, v is topmost in some embedding, and we are done.

- (2) There exists a $v \in S$ such that v has at least 2 incoming edges. Recall that all the edges entering v originate in the top vertex. Now there are two cases.

(a) All the incoming edges of v are contiguous in the left to right ordering on the edges leaving the top vertex. Then the incoming edges of v can be contracted, pulling v arbitrarily close to the top vertex.

(b) The incoming edges of v are not contiguous. That is, there are edges e_i, e_{i+1} with e_i to the left of e_{i+1} from the top vertex to v such that there are some edges x_1, x_2, \dots which come to the right of e_i but left of e_{i+1} .

Let p' be the subdiagram induced by taking x_1, x_2, \dots , their endpoints and all vertices reachable (by an undirected path) from their endpoints. This is the subdiagram contained in the region enclosed by e_i and e_{i+1} . Recursively find a vertex which is topmost in an embedding of p' . Clearly, this vertex can also be made topmost in an embedding of p by squishing the embedding of p' appropriately.

- (3) Every vertex in S has no incoming edges.

Say y_1, \dots, y_k are the outgoing edges of the top vertex. Let p be a graph (with orderings on edges) obtained from p as follows. Let u_1, \dots, u_k be k new vertices not present in p . Change the source of each y_i to be u_i rather than the top vertex. Essentially we explode the top vertex into k different vertices, one for each edge.

Let γ be an embedding of p . We can obtain an embedding γ' of p' respecting all the orderings from γ simply by slightly retracting each of the edges y_i away from $\gamma_{y_i}(0)$, the image of the top vertex.

Consider $X := \mathbb{R}^2 \setminus \gamma'(p')$. This space has exactly one non-compact (indeed, co-compact) connected component, call it X_0 . We want to compute all vertices and edges in S whose images lie on the boundary of X_0 .

This we do as follows. First, we define a sequence $(v_0, e_0), (v_1, e_1), \dots$ inductively such that

- The endpoints of e_i are v_i and v_{i+1}

- The image of each e_i under γ' is in the boundary of X_0

Let v_0 be the bottom vertex of p' and let e_0 be the rightmost edge entering v_0 . It is clear that e_0 must belong to the boundary of X_0 . Suppose we have defined v_i and e_i . Let v_{i+1} be the endpoint of e_i which is not v_i . There is a counterclockwise linear ordering on the edges incident to v_{i+1} induced by the embedding of p . Let e_{i+1} be the next edge counterclockwise from e_i at v_{i+1} . Since e_i is in the boundary, so too must be e_{i+1} since the region to the exterior of e_i is plainly connected to the region exterior to e_{i+1} .

Now carry out analogous processes with $(v_0, e_0) = (u_i, y_i)$ for each i . We have not proved that every edge lying on the boundary of X_0 will be discovered by this method.

The topmost vertex of p under γ belongs to the boundary of X_0 . This can be seen since there is clearly a path up from this vertex to the top vertex which doesn't intersect any edges. Furthermore, if a vertex v in S lies on the boundary of X_0 , then it is topmost in some embedding. Since it lies on the boundary of X_0 and X_0 is co-compact, there is a path starting at $\gamma'(v)$ which goes arbitrarily high in the plane. We have not proved it, but it should be possible to drag v along this path to make it the topmost vertex.

□

We can thus find the topmost vertex of a diagram by first reducing it and then applying the above algorithm.

2.12 Future possibilities

Several directions for future work were mentioned in section 1.3. Here we reiterate that an extension of string diagrams which handle product and sum types well would be a useful and interesting direction to pursue. As mentioned earlier, the current implementation in Mote uses an extension of string diagrams which are a kind of canonical form for natural transformation terms quotiented by equations holding for constant functors. Due to time constraints we have not been able to include a description in this thesis but we hope to write a paper describing this extension soon.

Bibliography

- [1] Lennart Augustsson. *Djinn, a Theorem Prover in Haskell, for Haskell*. Accessed: 2015-06-16. URL: <http://www.augustsson.net/Darcs/Djinn/>.
- [2] Steve Awodey. *Category Theory*. Great Clarendon Street, Oxford: Oxford University Press, 2010.
- [3] James Bornholt. *Program Synthesis, Explained*. Accessed: 2015-05-28. URL: <https://homes.cs.washington.edu/~bornholt/post/synthesis-for-architects.html>.
- [4] Ana Bove, Peter Dybjer, and Ulf Norell. “A Brief Overview of Agda — A Functional Language with Dependent Types”. In: *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*. TPHOLs '09. Munich, Germany: Springer-Verlag, 2009, pp. 73–78. ISBN: 978-3-642-03358-2. DOI: 10.1007/978-3-642-03359-9_6. URL: http://dx.doi.org/10.1007/978-3-642-03359-9_6.
- [5] Nils Anders Danielsson et al. “Fast and Loose Reasoning is Morally Correct”. In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '06. Charleston, South Carolina, USA: ACM, 2006, pp. 206–217. ISBN: 1-59593-027-2. DOI: 10.1145/1111037.1111056. URL: <http://doi.acm.org/10.1145/1111037.1111056>.
- [6] *GHC HsExpr type*. Accessed: 2015-05-28. URL: <https://downloads.haskell.org/~ghc/7.8.3/docs/html/libraries/ghc-7.8.3/src/HsExpr.html#HsExpr>.
- [7] *ghc-mod: Happy Haskell Programming*. Accessed: 2015-06-16. URL: <https://github.com/kazu-yamamoto/ghc-mod>.
- [8] Jean-Yves Girard. “Proof-nets: The parallel syntax for proof-theory”. In: *Logic and Algebra*. Marcel Dekker, 1996, pp. 97–124.
- [9] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. “Interactive Synthesis of Code Snippets”. English. In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 418–423. ISBN: 978-3-642-22109-5. DOI: 10.1007/

- 978-3-642-22110-1_33. URL: http://dx.doi.org/10.1007/978-3-642-22110-1_33.
- [10] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. “Playing by the rules: rewriting as a practical optimisation technique in GHC”. In: *2001 Haskell Workshop*. ACM SIGPLAN, 2001.
- [11] André Joyal and Ross Street. “The Geometry of Tensor Calculus, I”. In: *Advances in Mathematics* 88.1 (1991), pp. 55–112. ISSN: 0001-8708. DOI: [http://dx.doi.org/10.1016/0001-8708\(91\)90003-P](http://dx.doi.org/10.1016/0001-8708(91)90003-P). URL: <http://www.sciencedirect.com/science/article/pii/000187089190003P>.
- [12] André Joyal and Ross Street. *The Geometry of Tensor Calculus, II*. Accessed: 2015-04-09. URL: <http://maths.mq.edu.au/~street/GTCII.pdf>.
- [13] Susumu Katayama. “Recent Improvements of MagicHaskeller”. English. In: *Approaches and Applications of Inductive Programming*. Ed. by Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer. Vol. 5812. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 174–193. ISBN: 978-3-642-11930-9. DOI: 10.1007/978-3-642-11931-6_9. URL: http://dx.doi.org/10.1007/978-3-642-11931-6_9.
- [14] G.M. Kelly and M.L. Laplaza. “Coherence for compact closed categories”. In: *Journal of Pure and Applied Algebra* 19 (1980), pp. 193–213. ISSN: 0022-4049. DOI: [http://dx.doi.org/10.1016/0022-4049\(80\)90101-2](http://dx.doi.org/10.1016/0022-4049(80)90101-2). URL: <http://www.sciencedirect.com/science/article/pii/0022404980901012>.
- [15] Dextex Kozen. *Natural Transformations as Rewrite Rules and Monad Composition, Tech. Rep. TR2004-1942*. 2004.
- [16] Fredrik Lindblad. “Higher-Order Proof Construction Based on First-Order Narrowing”. In: *Electron. Notes Theor. Comput. Sci.* 196 (Jan. 2008), pp. 69–84. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2007.09.018. URL: <http://dx.doi.org/10.1016/j.entcs.2007.09.018>.
- [17] José Meseguer. “Conditional rewriting logic as a unified model of concurrency”. In: *Theoretical Computer Science* 96.1 (1992). Selected Papers of the 2nd Workshop on Concurrency and Compositionality, pp. 73–155. ISSN: 0304-3975. DOI: [http://dx.doi.org/10.1016/0304-3975\(92\)90182-F](http://dx.doi.org/10.1016/0304-3975(92)90182-F). URL: <http://www.sciencedirect.com/science/article/pii/030439759290182F>.
- [18] Daniel Perelman et al. “Type-directed Completion of Partial Expressions”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’12. Beijing, China: ACM, 2012, pp. 275–286. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254098. URL: <http://doi.acm.org/10.1145/2254064.2254098>.

- [19] A.J. Power. “An abstract formulation for rewrite systems”. English. In: *Category Theory and Computer Science*. Ed. by DavidH. Pitt et al. Vol. 389. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1989, pp. 300–312. ISBN: 978-3-540-51662-0. DOI: 10.1007/BFb0018358. URL: <http://dx.doi.org/10.1007/BFb0018358>.
- [20] John C. Reynolds. “Types, Abstraction and Parametric Polymorphism”. In: *Information Processing 83* (1983), pp. 512–523.
- [21] Mikael Rittri. “Retrieving Library Identifiers via Equational Matching of Types”. In: *Proceedings of the 10th International Conference on Automated Deduction, volume 449 of LNAI*. Springer Verlag, 1992, pp. 603–617.
- [22] Daniel Seidel and Janis Voigtländer. “Automatically Generating Counterexamples to Naive Free Theorems”. In: *Proceedings of the 10th International Conference on Functional and Logic Programming. FLOPS’10*. Sendai, Japan: Springer-Verlag, 2010, pp. 175–190. ISBN: 3-642-12250-7, 978-3-642-12250-7. DOI: 10.1007/978-3-642-12251-4_14. URL: http://dx.doi.org/10.1007/978-3-642-12251-4_14.
- [23] Lennart Spitzner. Accessed: 2015-05-28. URL: <https://github.com/lspitzner/exference/raw/master/exference.pdf>.
- [24] *Using IntelliSense*. <https://msdn.microsoft.com/en-us/library/hcw1s69b.aspx>. Accessed: 2015-05-28.
- [25] Philip Wadler. “Theorems for free!” In: *Functional Programming Languages and Computer Architecture*. ACM Press, 1989, pp. 347–359.