

BOSTON UNIVERSITY
GRADUATE SCHOOL OF ARTS AND SCIENCES

Thesis

PULER

by

BASSEL EL MABSOUT

B.S., American University of Beirut, 2015

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science

2023

© 2023 by
BASSEL EL MABSOUT
All rights reserved

Approved by

First Reader

Renato Mancuso, PhD
Assistant Professor of Computer Science

Second Reader

Marco Gaboardi, PhD
Associate Professor of Computer Science

Puler:

One who pules; one who whines or complains

Webster Dictionary

PULER

BASSEL EL MABSOUT

ABSTRACT

We address the prevalent challenge of Algebraic Data Type duplication in compiler implementations, which results in increased effort, diminished functionality, and complications in synchronizing language constructs across the compiler. To investigate a novel design solution, we present Tree Shaping, a solution to the expression problem. We then implement an experimental compiler using Tree Shaping and examine its potential implications. This compiler processes programs written in **PULER**, an ML-based programming language that boasts distinct features such as unification rules for type mismatches. Contrary to traditional compilers that terminate and generate an error when encountering a type mismatch, **PULER** regards type mismatches as first-class citizens.

Contents

1	Introduction	1
2	The Expression Problem	3
2.1	A Language of Shapes	3
2.1.1	Syntax	3
2.1.2	Example program	4
2.1.3	Adding a behavior	4
2.1.4	Adding a representation	5
2.1.5	Combined	6
2.2	Existing solutions	6
2.3	Tree Shaping	7
3	Compiler Design	11
3.1	Recursion-Schemes	12
3.2	Other compiler stages	13
3.2.1	Renamer	14
3.2.2	Interpreter	15
3.2.3	Autoformatter	15
3.2.4	Emission	16
3.3	REPL	16
4	PULER	18
4.1	Syntax	18
4.1.1	Features	19

4.2	Type system	20
5	Examples	23
5.1	Example Program	23
5.2	Example Commands	23
5.2.1	REPL example	23
5.2.2	Compilation stages example	24
6	Installation instructions	26
7	Conclusion	27
	References	28

List of Abbreviations

AST	Abstract Syntax Tree
ADT	Algebraic Data Type
GHC	Glasgow Haskell Compiler
BNF	Backus-Naur Form
DSL	Domain Specific Languages
REPL	Read Eval Print Loop
ML	Meta-Language

Chapter 1

Introduction

Typical programming language compilers are comprised of various *stages* that are chained together producing a final output from an initial input program. These stages can include parsing, renaming, type inference, optimization, and emission. For example, a C compiler takes as input a C program, which goes through these transformations and finally produces machine code. Similarly the Purescript [8] compiler produces JavaScript. While these transformations share abstract language constructs like **let-in** statements, **lambda** abstractions, and **if-then-else** statements [21], they also require transformation-specific information to be tracked. For instance, during type inference, a data-structure storing the language’s expressions **decorated** with their types and ununified variables must be defined [2].

When compilers are implemented in a programming language supporting **Algebraic Data Types (ADTs)** [10], they often contain multiple ADTs for each transformation, leading to duplication of effort, functionality, and difficulty in keeping the language’s *constructs* in sync. Additionally, extending the language with new constructs becomes difficult as the language designer must add necessary mechanisms to every datatype and function. Without a core datatype, bugs may arise since some intermediary ADTs may be left untouched, and the compiler may not reject the updated code. This issue presents itself in multiple different compiler implementations [15, 1], and is the core focus of a body of existing work [26, 3, 14].

Generalizing the goal of retaining both static type safety and functionality reuse,

the "Expression problem" [] defines the problem as a difficulty in extending both *behavior* and *representation*. We present Tree Shaping, a solution to the expression problem in Chapter ??.

We then introduce an experimental compiler making use of Tree Shaping as a case study of the effectiveness of the method, and study its implications in Chapter 3. This compiler accepts **PULER** programs, a new experimental *ML-based* programming language presented in Chapter 4 of this document. It then invokes different transformation as required by the final output requested, for an example of the stages involved, see Chapter 5.2.2 One of the distinctive features of **PULER** is defining **unification rules** for type mismatches, forming types themselves and being treated as *first-class* citizens in the type system. In contrast, existing compilers mostly exit and produce an error upon a type mismatch. Existing works focusing on this issue of type error debugging [20] generate orthogonal constraints which are separately represented and solved (e.g. via **SMT** solvers), producing errors which help users figure out the root cause. These ideas are discussed in Chapter 4.2.

Chapter 2

The Expression Problem

The Programming language community has identified a general problem in program design which makes designing data abstractions that are extensible both in their representations and behaviors difficult. This is known as the expression problem [27]. Constrained by the programming language’s ability to express such an abstraction while retaining type safety, solving the problem has been the topic of study for many existing techniques of which some are presented in Section 2.2.

2.1 A Language of Shapes

We showcase the expression problem in a simplistic toy programming language of recursive shapes.

2.1.1 Syntax

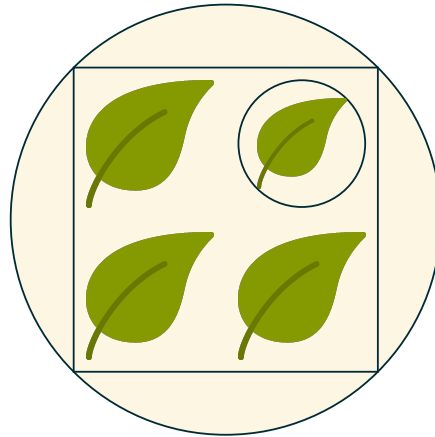
The following defines the BNF of our shapes language:

$$\begin{aligned}
 \langle expr \rangle &::= (\text{Circle } \langle expr \rangle) \\
 &\quad | (\text{Square } \langle square \rangle) \\
 &\quad | \text{Leaf} \\
 \langle square \rangle &::= \langle expr \rangle \langle expr \rangle \\
 &\quad \langle expr \rangle \langle expr \rangle
 \end{aligned}$$

2.1.2 Example program

We would like to compile shapes programs to an image, an example such program and its compiled output are shown below:

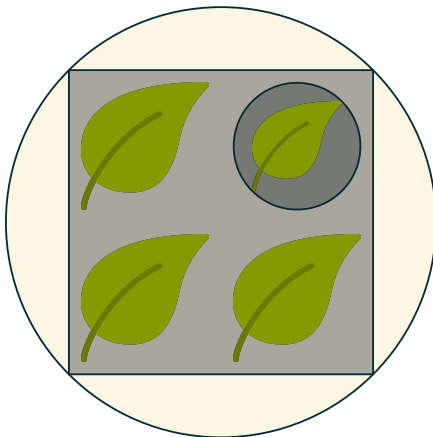
```
(Circle
  (Square
    Leaf (Circle Leaf)
    Leaf Leaf
  )
)
```



The important constraint is that the code which does the compilation may not be edited and now acts as a library. Any subsequent features which must be added can only import the initial library and use it.

2.1.3 Adding a behavior

We now request adding support for depth information, we would like to darken the background of the output image based on depth, as such:



Depending on the abstraction used for defining the initial library code this feature may require duplicating much of the code.

2.1.4 Adding a representation

As another feature, we ask the user to add a polygon representation which allows for adding arbitrary polygons to the language. The new syntax becomes:

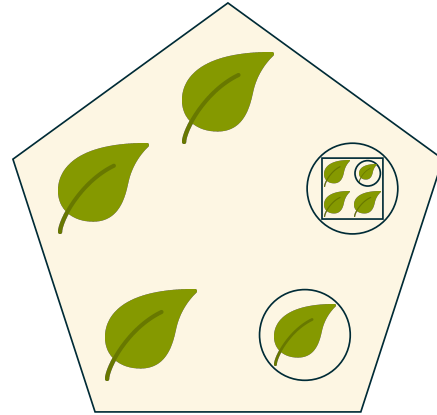
$$\begin{aligned}
 \langle expr \rangle &::= (\text{Circle } \langle expr \rangle) \\
 &\quad | (\text{Square } \langle square \rangle) \\
 &\quad | (\text{Poly } [\langle polygon \rangle]) \\
 &\quad | \text{Leaf} \\
 \langle square \rangle &::= \langle expr \rangle \langle expr \rangle \\
 &\quad \langle expr \rangle \langle expr \rangle \\
 \langle polygon \rangle &::= \langle expr \rangle \\
 &\quad | \langle expr \rangle, \langle polygon \rangle
 \end{aligned}$$

The following is an example program and its new compiled output:

```

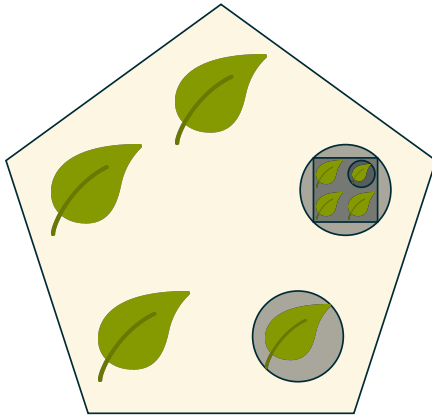
(Poly
  [Leaf, Leaf, Leaf, (Circle Leaf),
   Circle
    (Square
      Leaf (Circle Leaf)
      Leaf Leaf
    )
  ]
)

```



2.1.5 Combined

Finally, we would like to combine both features in a new program while reusing as much of the previous functionality as possible. This would be the output with both depth and polygons:



2.2 Existing solutions

Many works put forth solutions to the expression problem, such as:

- The Tagless Final Representation [5]
- Trees That Grow [19]
- Type Classes
- Open Classes (These don't count, as there is no type safety)
- Polymorphic Variants (Anonymous Discriminated Unions)
- The Visitor Pattern [27]

Examples of implementations of these solutions are available in the following repository: <https://github.com/bmabsout/TreeShaping>

2.3 Tree Shaping

As a solution to the expression problem, we present Tree Shaping. Tree Shaping is a method defined in Haskell for combining ADTs, Typeclasses, deriving mechanisms and Recursion-Schemes [18] (see Chapter 3.1 providing a satisfactory solution to the problem of extensionality while maintaining low functionality duplication. The main idea is summarized in the following steps:

- Define a few parametrized data types
- Auto-derive general typeclass instances for each data type such as a Functor instance
- Create a Typeclass for each F-Algebra[4] representing functionalities
- Collect the data types together into a recursive ADT
- Use recursion-schemes for composing the final recursive function
- Use free-monad based effect libraries with monadic recursion-schemes

For simplicity we ignore effect systems in the shapes example. The following shows a program using the Tree Shaping technique. Here are our core data types:

```
data Square a = Sq {topl :: a, topr :: a, botl :: a, botr :: a}
    deriving (Show, Eq, Functor, Foldable, Traversable)

data Circle a = Circ a
    deriving (Show, Eq, Functor, Foldable, Traversable)
```

Followed by our recursive type (We can automatically generate its corresponding generalized base type

```
data Shapes =
    S (Square Shapes) | C (Circle Shapes) | Leaf
makeBaseFunctor 'Shapes
```

With a few pattern synonyms for having a nice way to construct instances of the shapes language:

```
pattern Square{topl, topr, botl, botr} = S (Sq {topl, topr, botl, botr})
pattern Circle a = C (Circ a)
```

We end up with the ability to write our program as:

```
example =
  Circle $
    Circle $
      Square
        (Circle (Circle Leaf)) (Circle Leaf)
        Leaf                  (Circle Leaf)
```

Now we create our typeclass for drawing the shape as follows:

```
class Drawable f where
  draw :: (Renderable (Path V2 Double) a) => f (Diag a) -> Diag a

instance Drawable Square where
  draw sq = subDiagram # center <> square (maximum $ size subDiagram) # themed
    where
      paddedSq = padSubDiagsAndResize sq
      subDiagram =
        (paddedSq.topl ||| paddedSq.topr)
      == (paddedSq.botl ||| paddedSq.botr)

instance Drawable Circle where
  draw :: Renderable (Path V2 Double) a => Circle (Diag a) -> Diag a
  draw (Circ subDiagram) =
    subDiagram <> circle (norm (size subDiagram) / 2) # themed
```


We can now build our algebra and create the recursive function via a catamorphism:

```
drawShapeF :: _ => Diag a -> ShapesF (Diag a) -> Diag a
drawShapeF leaf = \case
  SF s  -> draw s
  CF s  -> draw s
  LeafF -> leaf

drawShape :: _ => Diag a -> Shapes -> Diag a
drawShape leaf = cata (drawShapeF leaf)
```

All of the previous code acts as our library, now we would like to add depth information, so we use a different data structure which supports decorating our recursive tree with extra information (Cofree):

```
addDepth :: (Recursive t) => t -> Word -> Cofree (Base t) Word
addDepth = cata \s w -> w :< (($ w+1) <$> s)

drawWithDepth :: (Integral n, Functor f) =>
  (Diag a -> f (Diag a) -> Diag a) -> Cofree f n -> Diag a -> Diag a
drawWithDepth diagAlg withDepth leaf =
  cata (\(d F.:< w) -> depthToTheme d (diagAlg leaf w)) withDepth
```

Adding the polygon representation can now be created along with a new ADT reusing ShapesF:

```
newtype Poly a = PolyC [a]
  deriving (Show, Eq, Functor, Foldable, Traversable)

data WithPoly
  = Orig (ShapesF WithPoly)
  | Pol (Poly WithPoly)
```

Now we just need need to add the Drawable instance for the Poly type:

```

instance Drawable Poly where
  draw (PolyC subShapes) = subDiagram2 <> polyDag # themed
    where
      shapes = padSubDiagsAndResize subShapes
      numShapes :: Num b => b
      numShapes = fromIntegral (length subShapes)
      r = norm $ size (head shapes)
      theta = pi / numShapes
      l = r / (2 * sin theta)
      directions = iterate (rotateBy (1 / numShapes)) (V2 0 1)
      subDiagram2 = zipWith translate directions shapes # mconcat
      polyDag = regPoly numShapes (r * tan theta + r)

```

Finally we get our final algebra via reusing the previous algebra and using the instance defined above.

```

drawWithPolyF leaf (OrigF shapeF) = drawShapeF leaf shapeF
drawWithPolyF _ (PolF subShapes) = draw subShapes

drawWithPoly :: _ => Diag a -> WithPoly -> Diag a
drawWithPoly leaf = cata (drawWithPolyF leaf)

```

The final test is composing both behavior and the new structure, and the following ends up just being a function call:

```

drawWithDepth drawWithPolyF (addDepth example 0)

```

Chapter 3

Compiler Design

On the implementation of compilers written in Haskell specifically, the work by Najd and Peyton Jones [19] presents and implements "Trees that grow", a technique for "decorating" ADTs by using type families. They then rewrite the internals of the Glasgow Haskell Compiler (GHC) [15] to make use of the method. In GHC, the data types for these transformations are large (dozens of types, hundreds of constructors) making them difficult to maintain. Their solution involves defining a single extensible ADT with Haskell's possible expressions. This allows different "decorations" of Haskell's Abstract Syntax Tree (AST) to be defined in separate parts of the compiler. Plugin writers then benefit from reusing the compiler as a library when extending the language.

In the PULER compiler, following the Tree Shaping method, each language construct is given a separate core parameterized type which is then reused across transformations. We then make use of deriving mechanisms to create reusable functionality across trees composed of these types. For example the following is the definition for the datatype of a Let expression:

```
data Let expr = Let (Decs expr) expr
    deriving (Eq, Functor, Foldable, Traversable)
```

This technique allows us to define dependencies between language constructs (here let expressions use declarations) without constraining the final tree structure and decorations added. The language designer can then work with each structure on its

own, and then define an ADT combining them. This allows the separation of the logic handling each type of expression in the language from the logic handling their composition. As an example, we show how the parser for let expressions is defined:

```
instance Parsable a => Parsable (Let a) where
  parser = Let
    <$> (begKeyword "let" *> parser)
    <*> (midKeyword "in" *> parser)
```

Typeclasses allow us to define how each piece of the language can be parsed while remaining generic with respect to how subexpressions are composed. Contrasting this with the "Trees that grow" method, the structure of the language is solidified in a single ADT, and while they allow open extensions to some parts of the language, they cannot introduce new language constructs easily. More information about the syntax of PULER is provided in Section 4.1.

3.1 Recursion-Schemes

In order to recursively connect subexpressions together, we form a core ADT:

```
data Expr
  = Evar Var
  | Eapp (App Expr)
  | Elet (Let Expr)
  | Eif (If Expr)
  | Edecs (Decs Expr)
  | Elambda (Lambda (N.NonEmpty Var) Expr)
  | Efix (Fixer Expr)
  | Elit Lit
  | Eannotation (Annotated Expr NamedTypes)
  deriving (Eq, Generic)
makeBaseFunctor 'Expr
```

Recursion-schemes [17] allow recursive algorithms to decouple local computations from the patterns used in the recursive step. This allows us to treat our **Expr**

datatype as an *F-algebra* [4] and write our transformations as combinations of algebras and co-algebras over similar types. Specifically, we make use of Haskell’s recursion-schemes [12] library. Combined with the use of effect systems (Polysemy [23]), this allowed breaking down the transformations into many smaller subproblems, and then composing them together into the final computed value. The ”makeBaseFunctor” template function creates an auxillary datatype, namely ”ExprF” which now acts as our interface with recursion schemes. This general method of structuring the compiler, allowed us to define different ADTs specific to each transformation, yet shared information is retained via the different datatypes for each construct. For example, here’s the definition of the ADT representing the possible values that the interpreter can produce:

```
data Value = Vlit Lit
           | Vlambda (Lambda (N.NonEmpty Var) LambdaBody)
           | VDecs (Decs Value)
           | Empty
```

Notice how, even though we define a new ADT that must be changed when adding new language constructs, the actual datatypes are just wrappers connecting our general datatypes (such as Decs). This trades off some duplication for flexibility, as such this occupies a point in the design space between full duplication and ”Trees that grow”.

3.2 Other compiler stages

This section contains the details of other compilation stages and of their implementation. They all make use of the aforementioned ideas for making compiler structuring decisions.

3.2.1 Renamer

The renamer keeps track of the scope of each variable and renames overlapping variables to an incremented version so that we don't worry about scope in later compiler passes:

```

-----INPUT-----
x = 3;
x = 4;
z = "test";
y = \x -> let f = 4 in f + x;
main = x;
-----RENAMED-----
x = 3;
x#1 = 4;
z = "test";
y = \x#2 -> let f = 4;
           in ((Add f) x#2);
main = x#1;

```

This is the main renaming logic which just allows us to push and pop a variable stack:

```

data Scope = Scope {numShadowing :: Shadowing, stack :: [Shadowing]}

type Renamer = M.Map Var Scope

insertToScope :: Var -> Renamer -> Renamer
insertToScope =
  M.alter $ Just . \case
    Just (Scope g l) -> Scope (g + 1) (g + 1 : l)
    Nothing -> Scope 0 $ [0]

removeFromScope :: Var -> Renamer -> Renamer
removeFromScope = M.update \(Scope g l) -> Just (Scope g (tail l))

```

A catamorphism is then used to recursively update the variables associated with individual scope levels, producing the final transformed AST.

3.2.2 Interpreter

This is PULER’s interpreter, it works by first “blinding” the original expression tree, meaning it hides the body of functions from recursion schemes so that we never evaluate the values inside of the bodies of functions. It then converts the AST into a Value type (as defined above). This is done with a catamorphism with access to a monad that keeps track of the scope of variables. The interpreter can even correctly evaluate the program without a renaming step.

3.2.3 Autoformatter

This is the autoformatter that comes with PULER, it appropriately indents new lines if there isn’t too much width space. Here’s a short program:

```
x = \test -> if True then 1 else 2
-----PARSED-----
x = \test -> if True
              then 1
              else 2;
```

Here’s a longer one:

```
-----INPUT-----
x = \test -> if True then "this is a very long message" else let y = 3 in ""
-----PARSED-----
x = \test ->
    if True
    then "this is a very long message"
    else let y = 3;
          in "";
```

Notice that the autoformatter decided to insert a new line after \rightarrow in the second case so that it can fit within 40 characters. Similar to how the parser is defined, each datatype has its own pretty printer, which is then stringed together forming the full pretty-printable expression type.

3.2.4 Emission

Finally, we emit the typed language to Python. The output is auto-formatted so it retains some readability even after transformation. The runtime is fairly simple, consisting of a few functions allowing the direct translation of any PULER program into Python. There are some extra tools in the emission code allowing the computation of the free variables computing the environment of every function and such, but for Python since we can create lambdas, then there's no need to create closures ourselves. The compiler initially was targeting C, however the combination of partial applications and C having no ability to create lambdas meant, if done naively, the generated output would create n functions for every n -variable function leading to much duplication. Another possibility is mallocing the environment of a function upon creation; storing it in a struct that also stores the arguments to the function, then when applying a single argument, simply assign the value to the argument in the struct. Now our function would be passed as a struct of arguments and an environment. When the last argument is provided, that's where we actually call the function. The only problem with this approach is that it requires tagging the functions to figure out when the final application is happening. The method we have tried create a tight coupling between the language and the backend. So we opted to emit to a language with closures. Namely, Python.

3.3 REPL

The language also includes a Read Eval Print Loop, which allows users to run valid (or invalid) PULER and interact with the language. This works by using the `conext` associated to every stateful operation in our architecture. Meaning the same code that tracks information like the current scope of variables is reused in the REPL to keep this state live. We can also get the benefit of providing features like autocomplete

since the context is mostly tracked using dictionaries. Another feature is that we can load files into the REPL, if you write a **PULER** program and save it, you can start a REPL that evaluates it and allows you to write code with it as part of the context. I found this to be immensely useful when debugging.

Chapter 4

PULER

PULER is an experimental ML based programming language that is strictly evaluated, partially applied, implements a Hindley-Milner-based [6] type system (Section 4.2), and is lexically scoped (Section 3.2.1). It can be interpreted (Section 3.2.2) or compiled (Section 3). There’s also a Read Eval Print Loop (REPL) (Section ??) available so that one can interact with the language live. The core language is simple, containing a few data types and language constructs. It can be extended through either adding compiler stages or minimally changing the representation used by the compiler to introduce new features. The language is functional first, meaning functions are first class citizens, and it disallows mutation, making it a referentially-transparent language.

One interesting feature of PULER is that the type system represents type mismatches. Unification rules are defined for mismatching types. This contrasts with typical programming languages’ type checking implementations, which exit and produce an error immediately upon mismatch. We motivate

4.1 Syntax

PULER’s syntax is very similar to languages like SML and Haskell. The following is a high-level **BNF**[16] with the details in Parser.hs:

$$\begin{aligned}
\langle abs \rangle &::= \backslash \langle var \rangle \rightarrow \langle subexpr \rangle \\
\langle subexprs \rangle &::= \langle subexpr \rangle | \langle subexpr \rangle \langle subexprs \rangle \\
\langle app \rangle &::= (\langle subexpr \rangle \langle subexprs \rangle) \\
\langle lit \rangle &::= \langle string \rangle | \langle integer \rangle | \langle boolean \rangle | \{\} \\
\langle dec \rangle &::= \langle variable \rangle = \langle subexpr \rangle \\
\langle decs \rangle &::= \langle dec \rangle | \langle dec \rangle ; \langle decs \rangle \\
\langle let \rangle &::= \text{let } \langle declarations \rangle \text{ in } \langle subexpr \rangle \\
\langle fix \rangle &::= \text{fix } \langle var \rangle \langle abs \rangle \\
\langle type \rangle &::= \langle type \rangle \rightarrow \langle type \rangle | \{\} | Int | Bool | Str \\
\langle annotation \rangle &::= \langle subexpr \rangle : \langle type \rangle \\
\langle subexpr \rangle &::= \langle abs \rangle | \langle lit \rangle | \langle let \rangle | \langle fix \rangle | \langle annotation \rangle | \langle app \rangle
\end{aligned}$$

At the top level of a PULER file $\langle decs \rangle$ are expected.

4.1.1 Features

- Lexical scoping: The scope of every variable is exactly defined by the location of its definition. Meaning its scope is completely defined at compile time. Defining a variable with the same name as another variable that is in scope means that this new variable shadows the earlier defined one.
- Partially applied: Applying an argument to a 3 argument function returns a function of 2 arguments instead of being an error.
- Hindley-Milner-based type system: If the program is correct then no types will need to be written anywhere, it is all inferred.

- Strict evaluation: The language is strictly evaluated, meaning function arguments are evaluated before the function is.
- Immutable: Any operation or function cannot modify the value of an existing variable. This along with the ability to create pure functions makes the language referentially transparent (this behavior is not preserved when you use the only impure function in PULER which is `Print`). Meaning you can replace a variable by the statement used to create it. The `=` operator is more like the mathematical definition of `=` and less like the assignment we're used to in imperative programming languages.

4.2 Type system

Multiple previous works have focused on the problem of type error localization. Some show relevant portions of failed type inference traces[7, 28], others show a slice of the program involved in the error [9, 25]. Other methods repeatedly call the typechecker, finding several error sources [13]. The most promising approach we've observed in this area comes from the works Pavlinovic et al. [20], Stuckey et al. [24], producing constraints which rank type errors and then are solved by an SMT solver. PULER makes no such choices, instead of producing an error and exiting the computation, the PULER type system keeps track of type mismatches. This is orthogonal to the works by Pavlinovic et al. [20] as then the decision to show some errors can be taken later, or the user can be shown multiple different "views" of the possible errors that happened for flexibility in type error debugging. Note that the PULER compiler cannot currently handle all kinds of type errors this way, as an infinite occurs check (for preventing infinite types) also results in the compiler exiting.

Type inference in PULER converts an `Expr` to a `Cofree ExprF INamedTypes`, this just means that we're annotating each node of the recursive abstract syntax

tree with a type. The type inference algorithm does the following: It initializes each expression with unification variables and relates the unification variables with inference rules. Then we “propagate” [22] the knowledge gained through a pass of applying the inference rules. We repeat this process (while doing occurs checks) until no further knowledge can be gained. When this is done we return the annotated tree.

This is the datatype describing the types used for inference:

```
data Itypes base
  = Iunif Name
  | IbaseType base
  | Iarrow (Itypes base) (Itypes base)
  | Imismatch [Itypes base]
```

In order to precisely describe what it means for knowledge to increase, we allow any 2 types to be joined to create a type considered the least upperbound of both:

```
instance Ord base => Semigroup (Itypes base) where
  -- this is actually a join semilattice
  a <> b | a == b = a
  Imismatch as <> Imismatch bs = Imismatch (nubQuick (as ++ bs))
  Iarrow a1 b1 <> Iarrow a2 b2 = Iarrow (a1 <> a2) (b1 <> b2)
  a@(Iunif _) <> (Iunif _) = a
  (Iunif a) <> b = b
  a <> (Iunif b) = a
  (Imismatch l) <> x = if x `elem` l then x else Imismatch (nubQuick $ x:l)
  x <> (Imismatch l) = if x `elem` l then x else Imismatch (nubQuick $ x:l)
  a <> b = Imismatch [a, b]
```

When we have 2 different types we can now join them to create a “larger” type. We now proceed by unifying types that are supposed to be equal (because of rules on the structure of the expression tree) by representing the types that are equal to each other as a disjoint set. Then we find the least upper bound of each disjoint set. Knowledge is then described as a map between each type and its corresponding “joined” type. Once we have this map we can “gain” knowledge by replacing any unification variables with a corresponding concrete type. This in turn will generate

new types to unify. This process is repeated until this knowledge map is unchanging. What's nice about doing things this way is that we can give users as much information about the types of expressions as we can get even when the types mismatch or are kept ambiguous. Here's an example with mismatching and ununified types:

```
x = 4;
y = x + "string";
main = (x 3);
-----INFERRED-----
x = 4:Int;
y = ((Add x:[ Int
          , String
          , Int->?d ]):[ Int
                        , String ]->Int
     "string":String):[ Int
                        , String
                        , Int->?d ];
main = (x:[Int, String, Int->?d]
        3:Int):?d;:{}

```

Notice that there are 3 different type errors in here. First when we see the type [a, b] this means that the types a and b don't match. We can see here that the type of x is inferred to be [Int, String, Int->?d]. Second when we see a ? symbol this means it is a unification variable and that this variable remained ununifiable. Then notice that x it's self is actually used in three separate incompatible ways. First x is set to be the number 4 which is an Int, then we're adding a string to x but you can't add strings to numbers so that creates the first mismatch, the second mismatch comes from using x as a function in main. Given that we never use the result of the function this means we don't "know" what the result type is, which is another type error. Even though the output is currently verbose we could simply query the type of specific variables and this ability allows tracking the trace of a mismatch which we also found useful.

Chapter 5

Examples

5.1 Example Program

```

fib = fix fib \x ->
  if x == 0
  then 0
  else if x == 1
    then 1
    else (fib (x-1)) + (fib (x-2));

main = (fib 8);
>>
21

```

5.2 Example Commands

5.2.1 REPL example

Open a repl and run expressions as well as check their types:

```

> PULER repl
Welcome to PULER!

> x = 1
x = 1;
> y = \z -> z
y = \z -> <body>;
> :t (y x)
(y: Int -> Int x: Int): Int

```

```

> (y x) + 1
2
> (Print "test")
"test"
{}

```

5.2.2 Compilation stages example

Compile an example *.pul file implementing the factorial function

```
> PULER example.pul
```

```

-----INPUT-----
fact = fix f \x ->
    if x == 0
    then 1
    else let nextFac = x * (f (x - 1));
          p = (Print (Int2Str nextFac))
          in nextFac;
main = let g = (fact 8) in {};

-----PARSED-----
fact = fix f \x -> if ((EqInt x) 0)
    then 1
    else let nextFac = ((Mul x) (f ((Sub x) 1)));
          p = (Print (Int2Str nextFac));
          in nextFac;
main = let g = (fact 8);
    in {};

-----RENAMED-----
fact = fix f \x -> if ((EqInt x) 0)
    then 1
    else let nextFac = ((Mul x) (f ((Sub x) 1)));
          p = (Print (Int2Str nextFac));
          in nextFac;
main = let g = (fact 8);
    in {};

-----INFERRED-----
fact = fix f \x -> if ((EqInt x:Int):Int->Bool 0:Int):Bool
    then 1:Int
    else let nextFac = ((Mul x:Int):Int->Int
        (f:Int->Int ((Sub x:Int):Int->Int

```



```

1:Int):Int):Int):Int;
    p = (Print (Int2Str nextFac:Int):String):{};
    in nextFac:Int:Int:Int:Int->Int;
main = let g = (fact:Int->Int 8:Int):Int;
      in {}:{}:{}: {};{}
-----CHECKED-----
Typechecks!
-----EVALUATED-----
1
2
6
24
120
720
5040
40320
{}
-----EMITTED-----
fact = fix(lambda f: lambda x:(
  (
    1
  ) if (
    EqInt(x)(0)
  ) else (
    let(
      nextFac = Mul(x)(f(Sub(x)(1))),
      inn = lambda nextFac:(
        let(
          p = Print(Int2Str(nextFac)),
          inn = lambda p:(
            nextFac
          )
        ))
    ))
  )
))
main = let(
  g = fact(8),
  inn = lambda g:(
    unit
  ))

```

Chapter 6

Installation instructions

1. Clone the following repository: [PULER](#)
2. Install the [nix package manager](#) and enable flakes
3. Navigate to the PULER folder
4. Run the `nix develop` command which puts you in a reproducible environment with the exact packages (and packages versions) required to compile the compiler
5. Compile the code by running `ghc -O2 Main.hs`
6. Now you can use the compiler by calling `./Main`

Chapter 7

Conclusion

After multiple iterations of compiler design, **PULER** has converged to the idea of having main datatypes stringed together to form algebras on which the algorithms forming compiler stages are built upon. In exploring this architecture, we've built a full programming language with multiple interesting features. The architecture made it easy to use the compiler as a library and build a REPL for example with very minimal amounts of code. We did notice however, that even with careful design in avoiding duplication, there is still boilerplate code which duplicates parts of the core ADT throughout compiler transformations. The sum types used tag each alternative branch with a name which is unnecessary in our approach since the datatypes are already unique. As such, we believe connecting the types of expressions together via strongly typed heterogeneous lists [11] would lead to an even more terse and composable interface, we leave this as an exploration in future work. At its current state, we found it easier to add any type of construct to the language and fill in the implementation gaps, than otherwise would be the case without this architecture. Another remaining challenge is targetting C as a backend, the limitations of which may strain the architectural decisions made in **PULER**.

References

- [1] Ocaml. URL <https://v2.ocaml.org/manual/>.
- [2] A. W. Appel and J. Palsberg. Introduction. In *Modern Compiler Implementation in Java*, pages 3–15. Cambridge University Press, Cambridge, Oct. 2002.
- [3] E. Axelsson. A generic abstract syntax model for embedded languages. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, page 323–334, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310543. doi: 10.1145/2364527.2364573. URL <https://doi.org/10.1145/2364527.2364573>.
- [4] M. Bartosz. F-algebras. URL <https://bartoszmilewski.com/2017/02/28/f-algebras/>.
- [5] J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated. In *Programming Languages and Systems*, pages 222–238. Springer Berlin Heidelberg. doi: 10.1007/978-3-540-76637-7_15. URL https://doi.org/10.1007/978-3-540-76637-7_15.
- [6] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '82*. ACM Press, 1982. doi: 10.1145/582153.582176. URL <https://doi.org/10.1145/582153.582176>.
- [7] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996. ISSN 0167-6423. doi: [https://doi.org/10.1016/0167-6423\(95\)00007-0](https://doi.org/10.1016/0167-6423(95)00007-0). URL <https://www.sciencedirect.com/science/article/pii/0167642395000070>.
- [8] P. Freeman. *PureScript by Example*. <https://leanpub.com/purescript>, 2017. <https://leanpub.com/purescript>.
- [9] H. Gast. Explaining ML type errors by data flows. In *Implementation and Application of Functional Languages*, pages 72–89. Springer Berlin Heidelberg, 2005. doi: 10.1007/11431664_5. URL https://doi.org/10.1007/11431664_5.
- [10] J. Guttag and J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1), 1978. doi: 10.1007/bf00260922. URL <https://doi.org/10.1007/bf00260922>.

- [11] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, page 96–107, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138504. doi: 10.1145/1017472.1017488. URL <https://doi.org/10.1145/1017472.1017488>.
- [12] E. Kmett. Recursion-schemes. URL <https://hackage.haskell.org/package/recursion-schemes>.
- [13] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. *SIGPLAN Not.*, 42(6):425–434, jun 2007. ISSN 0362-1340. doi: 10.1145/1273442.1250783. URL <https://doi.org/10.1145/1273442.1250783>.
- [14] A. Löb and R. Hinze. Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, page 133–144, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933883. doi: 10.1145/1140335.1140352. URL <https://doi.org/10.1145/1140335.1140352>.
- [15] S. Marlow and S. L. P. Jones. The glasgow haskell compiler. 2012.
- [16] D. D. McCracken and E. D. Reilly. *Backus-Naur Form (BNF)*, page 129–131. John Wiley and Sons Ltd., GBR, 2003. ISBN 0470864125.
- [17] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer Berlin Heidelberg, 1991. doi: 10.1007/3540543961_7. URL https://doi.org/10.1007/3540543961_7.
- [18] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer Berlin Heidelberg, 1991. doi: 10.1007/3540543961_7. URL https://doi.org/10.1007/3540543961_7.
- [19] S. Najd and S. Peyton Jones. Trees that grow. *Journal of Universal Computer Science (JUCS)*, 23:47–62, January 2017. URL <https://www.microsoft.com/en-us/research/publication/trees-that-grow/>.
- [20] Z. Pavlinovic, T. King, and T. Wies. Finding minimum type error sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, page 525–542, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325851. doi: 10.1145/2660193.2660230. URL <https://doi.org/10.1145/2660193.2660230>.
- [21] B. C. Pierce. *Types and Programming Languages*. The MIT Press. MIT Press, London, England, Jan. 2002.

- [22] A. Radul. *Propagation networks: A flexible and expressive substrate for computation*. PhD thesis, Massachusetts Institute of Technology, 2009.
- [23] M. Sandy. Polysemy. URL <https://hackage.haskell.org/package/polysemy>.
- [24] P. J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, page 80–91, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138504. doi: 10.1145/1017472.1017486. URL <https://doi.org/10.1145/1017472.1017486>.
- [25] F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.*, 10(1):5–55, jan 2001. ISSN 1049-331X. doi: 10.1145/366378.366379. URL <https://doi.org/10.1145/366378.366379>.
- [26] M. Torgersen. The expression problem revisited. In *ECOOP 2004 – Object-Oriented Programming*, pages 123–146. Springer Berlin Heidelberg, 2004. doi: 10.1007/978-3-540-24851-4_6. URL https://doi.org/10.1007/978-3-540-24851-4_6.
- [27] P. Wadler. Expression problem. URL <https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt>.
- [28] M. Wand. Finding the source of type errors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, page 38–43, New York, NY, USA, 1986. Association for Computing Machinery. ISBN 9781450373470. doi: 10.1145/512644.512648. URL <https://doi.org/10.1145/512644.512648>.