Boston University, 2019

PULER, An ML-based PL

Bassel El Mabsout

December 18,2020

Contents

1 Introduction

PULER is an ML based programming language that is strictly evaluated, is partially applied, implements a Hindley-Milner type system, and is lexically scoped. It can be interpreted, compiled, and there's a REPL available so that you can interact with the language live. The core language is simple, containing a few data types and language constructs but can be extended very easily through either defining new functions or minimally changing the representation used by the compiler to introduce new features. It is functional first, meaning functions are first class values and does not allow mutation, making it a referentially-transparent language. One interesting feature of PULER is that the type system represents type mismatches, and unification rules are defined for mismatching types, rather than the type checker exiting and producing an error immediately as commonly done.

1.1 Motivations

The main motivation of this project is to experiment on compiler architecture design with the goal of making it easy to experiment on programming language features. In doing so I was able to investigate the following questions [1]:

- Whether structuring a compiler using recursion schemes and free monads (for composing effects) in Haskell would help with separation of concerns in the different compiler stages and whether that would allow for easily extending the language with new constructs.
- Whether sharing a main abstract datatype across transformations would force the handling of new constructs where needed.
- Whether the notion of mismatching types and ununified variables can be part of the type system, delaying when programs are rejected or accepted, so that users can have more information when debugging type errors.
- Whether the algorithm performing the type checking can be separated from the constructs used to define the type system, relying only on the join semi-lattice structure of the datatype representing the type system.

1.2 Related Works

2 Examples

2.1 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
```

2.2 Example Commands

2.2.1 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"
{}
```

2.2.2 Compilation 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
))
```

3 Installation

Install the compiler by following these instructions:

1. Clone the following repository: PULER
2. Install the nix package manager
3. Navigate to the PULER folder
4. Run the `nix-shell` 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`

4 Syntax

PULER's syntax is very similar to languages like SML and Haskell. The syntax can be fully understood by looking at the Parser.hs file but the here's a high level description: At the top level there are *declarations* of the form "$variable = subexpr$;". Subexpressions can be one of the following:

- Lambda *abstraction*: $\backslash variable \rightarrow subexpr$
- Application: $(subexpr\ arg_1\ arg_2 \ldots arg_n)$
- Literal: "*string*" or *integers* or *booleans* or {} (unit)
- Let: let *declerations* in *subexpr*
- If: if *subexpr* then *subexpr* else *subexpr*
- Fix: fix *variable abstraction*
- Annotation: $subexpr : type$

Types have the following syntax: the function type $type \rightarrow type$ or a few base types like Int, String and {} (the unit type)

5 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 typed: 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 (the only time this behavior is not preserved is when you 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.

6 Architecture

The compiler is written (in Haskell) around a few core data types that are reused across transformations. These are defined in the Core.hs file. These pieces are left generic so that the different representations arising from the compiler passes can be written by wiring together these types into a tree. The idea is to connect these pieces through a recursive data structure represented as an algebraic data type. We can then treat this type as an F-algebra and write our transformations as combinations of algebras and co-algebras over these types. This is facilitated by the recursion-schemes library. This combined with the use of effect systems allowed me to break down the transformations into many smaller subproblems and then compose them together. The compiler is divided into the following high-level transformations:

6.1 Parser.hs

This is the parser code, written using parser combinators. It transforms strings into the Expr type representing the abstract syntax tree:

```
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)
```

6.2 Renamer.hs

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:

```haskell
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))
```

6.3 TypeSystem.hs

Type inference 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" 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:

```haskell
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:

```haskell
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 3 seperate 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 to always give a reasonable answer I found to be really nice! Also we can track the trace of a mismatch which I also found useful.

6.4 Evaluator.hs

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. And then by converting that to values of the following type:

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

Which is done with a catamorphism with access to a monad that keeps track of the scope of variables. This interpreter even works without a renaming step.

6.5 PrettyStuff.hs

This is the autoformatter that comes with PULER, it does things like go to an indented new line 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 it decided to insert a new line after → in the second case so that it can fit within 40 characters.

7 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.

8 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. I initially though about outputting to 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 which was very unappealing. There is another way this could be done that I've figured out. Malloc the environment of a function upon creation. Store 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 technically our function would be passed around 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 in a way so that you can tell when the final application is happening. The only ways of doing this that I've thought of end up coupling the language to the backend too much. So I opted to emit to a language with closures.

9 Conclusion

This has certainly been a challenging project to say the least. But I did find it very rewarding as I've rearchitected the language multiple times to account for the new things I've learned and the features I wanted to make and ultimately settled with what is presented here. At its current state, it's actually pretty easy to add any type of construct to the language and fill in the implementation gaps. I do think there are ways to improve things further though. One can use type level lists instead of creating a sum type every time we want a new representation. This would allow us to work on transformations of subsets of the expression tree to while only describing the difference between the types instead of essentially copying the same definition twice.

References

[1] Latex documentation. http://latex-project.org/guides/, jan 2015. Good site for tutorials.