Program Analysis and Transformation

Jacob Herbst (mwr148)

Higher order Deforestation

Tool for playing with deforestation of higher order terms

Repository: `https://github.com/Spatenheinz/LeSpeciale`

2023-06-19

Abstract

Contents

1 Introduction

Many optimization techniques exists for making compiled programs more efficient. Especially purely functional programming languages lend themselves naturally to many optimization techniques as there exists no side conditions hence allowing for easy sound optimization techniques. One such example is fusion, also called deforestation. Deforestation is in essence nothing more than term rewriting to remove intermediate datastructures. An example of a very simple fusion rule is *map f (map g xs) = map (f . g) xs*. In this fusion rule the inner map will create an intermediate list and require two iterations over lists, while the fused version will only create the result list and iterate xs once. More aggressive fusions rules exists, and the early 90's have presented a varyity of deforestation algorithms starting with Wadler's deforestation algorithms [7]. Wadler considers deforestation with respect to a first order functional language. He extends his notion of deforestation to higher order languages by considering macros. This leaves of with an unsatisfactory result, where higher order types are first class citizens. The language is furthermore restrictive in that nested cases are not allowed. Many more deforestation algorithms exist such as [1], but are specific to lists. Hamilton then presented an algorithm to deforest higher order languages in [2]. This algorithm differs from algorithms such as the one presented in [5] since it is guranteed to terminate.

In this report, we present a tool for toying with deforestation in a similar manner to other tools presented in the Program Analysis and Transformation course. Specifically we present a language that adheres to the descriptions of [2]. In this we provide a repl, where it is possible to define custom datatypes and functions. It is possible to load files, int othe repl and at definition time the functions are type-checked using Hindley-Milner style type inference. The repl further provides commands for evaluation, pretty-printing and showing types of expressions. Lastly, as a key focus of the tool it is possible to deforest an expression. This feature is currently broken, as there is some discrepancy between the algorithm we re-represent in **??** and the implementation. The confusion is presented in **??**. We further present an example in which case the deforestation works correctly. We present the language in **??** and give a brief introduction into the tool in **??**.

2 Language

For the tool we present in this report, we consider a small higher-order language, which resembles the Haskell core[3], while adhering to the language in [2].

2.1 Definition

The abstract syntax of the language is presented in **??**. A program is a list of declarations. Each declaration may either be a Type- or a function definition.

Type definitions consist of a type constructor *t* followed by a list of variables, which makes the type potentially polymorphic, followed by a |-separated of Type constructors *c*, which may take several variables as arguments.

A function definition consists of a function name followed by any potential parameters and then the expression the symbols should describe.

expressions may be either, a variable, a constructor, or a literal, where literals are integers and characters and the unit () symbol. Expressions can then also be a lambda abstraction or an application of expression *M* onto the expression *N*. Lastly, the language contains case expression, allowing for the deconstruction of types (and semantically strict let bindings). We do not consider any let bindings as no rules for these are presented in [2].

$$
\begin{array}{rll}
Prog & ::= D_s & \text{(list of declarations)} \\
\\
D & ::= t\ v_s\ =\ (c\ v_s)* & \text{(type declaration)} \\
 & \mid f\ v_s\ =\ M & \text{(function definition)} \\
\\
M, N & ::= v & \text{(variable)} \\
 & \mid c & \text{(constructor)} \\
 & \mid l & \text{(literal)} \\
 & \mid M\ N & \text{(application)} \\
 & \mid \lambda v.M & \text{(abstraction)} \\
 & \mid \textbf{case}\ M\ \textbf{of} P_1 \longrightarrow M_1 \mid \cdots \mid P_n \longrightarrow M_n & \text{(case expression)} \\
P & ::= literal & \text{(literal pattern)} \\
 & \mid v & \text{(variable pattern)} \\
 & \mid & \text{(variable pattern)} \\
 & \mid c\ P_1 \ldots P_n & \text{(constructor pattern)}
\end{array}
$$

Figure 1: Syntax of object language

For the concrete syntax the definitions in **??** should suffice. We can construct types such as List and Bool in other functional languages. Functions may be recursive, but we do not consider any form of mutual recursion. This would require a notion of a binding group which we may extend the language with in the future. This language is more expressible than the one presented by Wadler in [7], as we require no macro notion of higher-order functions, recursive definitions and custom datatypes.

In the internal workings of the tool, we use a different representation for function definition, where it will simply be a pair (f, e). We reduce the top-level expressions to (f,e) by abstracting each variable. For instance, we get:

```
(repeat, \f -> \x -> Cons x (repeat f (f x)))
```

by transforming the repeat function. In this process, we do some trivial sanity checks on declarations, such as ensuring that no two variables have clashing names and the general well-formedness of declarations. These functions can be seen in Appendix **??**.

2.2 Typing

We should not spend too much time on the typing semantics of the language as the main focus should be on the deforestation algorithm. However, to ensure well-formedness of expression

```
List a = Cons a (List a) | Nil;
Bool = True | False;

fold f a xs = case xs of
               Nil -> a
               | Cons y xs -> fold f (f a y) xs;

map f x = case x of
            Nil -> Nil
            | Cons x xs -> Cons (f x) (map f xs);

until p xs = case xs of
               Nil -> Nil
               | Cons x xs -> case p x of
                              True -> Nil
                              | False -> Cons x (until p xs);

repeat h x = Cons x (repeat h (h x));
```

Figure 2: Simple declarations

we must briefly consider a type-checking phase.

We consider type checking or rather type inference of programs using an algorithm J approach[6] but with the typing information from Typing Haskell in Haskell[4].

Specifically, we consider the inference as presented in Figure **??**. Literals are trivially typed under the constructor for that type. For variables, we look up the type scheme in the environment and then instantiate the type. Lambda expressions generate a new type variable tv and then extend the typing context $\Gamma, x : tv$ when inferring the type of the body. Applications will infer the type of each subpart m and n and then generate a new type variable which is returned. One thing to note however is that we write a constraint using the writer monad, stating that, the type of m must be unifiable with the arrow type $nt \rightarrow tv$, where nt is the type of n and tv is the type of $(m\ n)$. Constructors are handled in the same manner as variables, and primitive operators are essentially just functions and thus handled as an application on two arguments, thus omitted from the presented code. They are presented in Abstract **??**.

For case expressions, we infer the type of the selector m and generate a new type variable and then we infer the alternatives. This is done by inferring each alternative and writing the constraint that each branch in a case should have the same type t, where the result of the alternatives is the type t, which will be a type-variable.

Figure **??** show inference of alternatives. A single alternative is inferred by checking the type of the pattern and then checking the body of the branch. We here further constraint that the pattern must have the same type as the selector of the case expression. Inferring patterns will return a pair. Both the type of the pattern and the typing context should extend the typing context with new types for each variable in the inferred pattern. Here we again omit the trivial patterns.

By running $infer$ on an expression we get its type, which in many cases is just a type-variable, as well as the list of constraints.

The list of constraints is solved, for each constraint $t1$ and $t2$ apply the current substitution, which is initially empty, and then finding the most general unifier of the results as in Figure **??**.

```haskell
infer :: Expr -> Infer Ty
infer = \case
  Lit (LInt _) -> return $ TCon $ TC "Int" Star
  Lit (LChar _) -> return $ TCon $ TC "Char" Star
  Lit LUnit -> return $ TCon $ TC "()" Star

  Var x -> do
    env <- ask
    case TyEnv.lookup x env of
      Nothing -> throwError $ UnboundVariable x
      Just t -> inst t

  Lam x m -> do
    tv <- fresh Star
    -- for now we only allow variables in the lambda
    x' <- case x of
      (Var x) -> return x
      _ -> throwError $ UnboundVariable "lambda"
    t <- local (extend x' $ toScheme tv) $ infer m
    return $ tv `fn` t

  App m n -> do
    t1 <- infer m
    t2 <- infer n
    tv <- fresh Star
    tell [t1 :~: (t2 `fn` tv)]
    return tv

  Case m alts -> do
    tv <- fresh Star
    t <- infer m
    inferAlts alts tv t
```

Figure 3: Implementation of type inference

If the program is well-typed, we get a substitution as a result which we can apply to the result of inferring m and we can then quantify the variables that occur in it. This gives us the following types for the definitions in Figure **??**.

```haskell
fold ::   a   b c. (a -> b -> c) -> a -> List b -> a
map ::   a   b. (a -> b) -> List a -> List b
until ::   a  . (a -> Bool) -> List a -> List a
repeat ::   a   b. (a -> b) -> a -> List a
```

2.3 Operational Semantics

We have not considered an operational semantics for the language, however for the deforestation algorithm to be sound, meaning that it preserves the semantics of an expression, the evaluation strategy must be non-strict. If we consider the expression

fold (+) 0 (map square (until (> n) (repeat (+1) 1)))

then it will never terminate in non-strict semantics, as *repeat (+1) 1* will generate an infinite list, on the other hand, the result of deforestation, presented in **??**, will terminate even with a strict semantic.

```haskell
inferPat :: Pattern -> Infer (Ty, TyEnv)
inferPat = \case
  VarAlt v -> do
    tv <- fresh Star
    return (tv, extend v (toScheme tv) mempty)

  ConAlt c ps -> do
   (ts, envs) <- inferPats ps
   t' <- fresh Star
   t <- tryFind c
   tell [t :~: foldr fn t' ts]
   return (t', envs)

inferAlt :: Ty -> AST.Alt -> Infer Ty
inferAlt t0 (p, m) = do
  (ts, env) <- inferPat p
  tell [t0 :~: ts]
  t' <- local (merge env) $ infer m
  return $ t'

inferAlts :: [AST.Alt] -> Ty -> Ty -> Infer Ty
inferAlts alts t t0 = do
  ts <- mapM (inferAlt t0) alts
  tell $ map (t :~:) ts
  return t

inferPats :: [Pattern] -> Infer ([Ty], TyEnv)
inferPats ps = do
  x <- mapM inferPat ps
  let ts = map fst x
      envs = map snd x
  return (ts, foldr merge mempty envs)
```

Figure 4: Implementation of type inference for patterns

3 Treeless Form

The act of deforestation is to eliminate trees from an expression. This can be done if the term is linear and all functions in the term have a treeless definition. The treeless form for the language is defined in **??**

Specifically, this definition states that a treeless form of an expression requires case selectors and function arguments to be variables, or a blazed term, which we describe later. We further constraint that a treeless form requires the expression to be linear, meaning all variables occurs at most once in the expression. Notice here also that literals are considered variables in the sense of treelessness.

The reason arguments and case selectors must be variables ensures that no intermediate trees are generated, For instance in the term:

until (> n) (repeat (+1) 1)

The *repeat (+1) 1* will generate an intermediate tree. Be aware here that applications where the leftmost function point is a constructor do not impose the same restriction, since the arguments for a constructor are part of the result, whereas they for functions may be destroyed.

```haskell
runSolve :: [Constraint] -> Either InferErr Subst
runSolve cs = runExcept $ solve (mempty, cs)

solve :: Unif -> Solver Subst
solve (sub, []) = return sub
solve (sub, (t1 :~: t2) : cs) = do
  sub' <- unify (apply sub t1) (apply sub t2)
  solve (sub' @@ sub, apply sub' cs)

unify :: Ty -> Ty -> Solver Subst
unify (TVar v) t = bind v t
unify t (TVar v) = bind v t
unify (TAp a b) (TAp a' b') = do
  s <- unify a a'
  s' <- unify (apply s b) (apply s b')
  return (s' @@ s)
unify t1 t2 | t1 == t2 = return mempty
            | otherwise =  throwError $ Unified t1 t2

bind :: TVar -> Ty -> Solver Subst
bind v t | t == TVar v = return mempty
         | v `elem` ftv t = throwError $ Infinite v t -- occurs check
         | kind v /= kind t = throwError $ KindMismatch (TVar v) t
         | otherwise = return $ v +-> t
```

Figure 5: Constraint solving generated from type inference

$$
\begin{aligned}
E ::= {} & v \\
& \mid c\ E_1 \ldots E_n \\
& \mid E\ E' \\
& \mid \lambda v.E \\
& \mid \textbf{case}\ E'\ \textbf{of}\ P_1 \to E_1 \mid \cdots \mid P_n \to E_n
\end{aligned}
$$

$$
\begin{aligned}
E' ::= {} & v \\
& \mid E^{\ominus}
\end{aligned}
$$

Figure 6: Treeless form

The linearity constraint is imposed because the language doesn't define any local storage, which means that functions such as *square* = λ *x* -> *x* * *x* will make a program less efficient by substituting *square e* for *e* * *e* in non-strict semantics in case *e* is expensive to compute.

These restrictions severely limit the programs we can write, and the terms we can remove trees from. Thus we consider the notion of blazing expression, which we denote by $\ominus$. In essence, we blaze variables at their binding level, if they are non-linear, case selectors which are not variables, and function arguments which are not variables. The meaning of this is that these expressions cannot be eliminated during deforestation and must remain in the transformed expression. The reader familiar with the deforestation presented by Wadler[7], will know that his paper uses a different notion of blazing which is type-based[1]This is also the initial intent for the type checker. But as the language transitioned to a higher order, this became less relevant for the project and thus seem superflous.} With these restrictions, we can put all function

[1]{

declarations in treeless form. Allowing their occurrence in expressions to be deforested.

For the functions we defined earlier the treeless form looks as such:

```
fold = \f⊖ -> \a -> \xs -> case xs of
                         Nil ->  a
                         Cons x xs -> fold f (f a x)⊖ xs
map = \f⊖ ->  \x -> case x of
                      Nil -> Nil
                      Cons x xs ->  Cons (f x) (map f xs)
until = \p⊖ -> \xs -> case xs of
                  Nil ->  Nil
                  Cons x⊖ xs -> case (p x)⊖ of
                                  True  ->  Nil
                                  False  ->  Cons z (until p xs)
repeat = \f⊖ ->  \x⊖ ->  Cons x (repeat f (f x)⊖)
square = \x⊖ ->  ((x * x))⊖
```

Figure 7: Treeless form

Making a treeless form representable in the code is simple. We simply add a *Blazed* constructor to the Expr type and similarly for the Pattern type. We also easily blaze variables in lambda expressions as the *x* in λxe is represented as a *Lam (Var "x") e*. The code for blazing is straight forward and shown in Figure **??**. simple expressions such as variables, constructors, literals, and already blazed terms, are not blazed. For abstractions, we check if the binding occurrences are linear and if so we blaze the binder, then we also blaze the body of the abstraction. The case for application shows some of the unfortunate clashes from the curried function application form we have considered for the internal representation. We need some way of knowing if the leftmost function point is a constructor or function. We flatten the entire expression, such that the function point is either a function or a constructor, and the tail of the list are arguments. If we have a constructor we simply traverse arguments to convert to treeless form and reconstruct the application as a tree. In case the function point is not a constructor we check if the argument is a variable and blaze the term if not. Similarly, we blaze case expressions, where patterns act like binders for variables occurring in the pattern.

```
blaze :: Expr -> Expr
blaze = \case
  Lam x e ->
    let x' = getVar x
    in if linear y' e
       then Lam x (blaze e)
       else Lam (Blazed x) $ blaze e

  e@(App e1 e2) ->
    let flat = flatten e
        (hd, tl) = (head flat, tail flat)
    in case hd of
      Con _ -> toTree $ map blaze flat
      _ -> toTree $ (blaze hd) : map (\x -> if compound x then
                                               Blazed $ blaze x
                                            else x) tl
  ... Case and operators can be found in appendix
  x -> x
```

Figure 8: implementations of blazing

When defining a function in the tool it will be blazed, so it can be used in deforestation.

4 Deforestation algorithm

The deforestation algorithm is a transformation on a term of the object language that will attempt to make a higher order treeless version of the input. We denote the transformation as $\mathcal{T}[\![M]\!]$ where M is the term to be transformed. The transformation is syntax directed and defined as a set of equation throughout this section.

Rule 1, simply deforest inside a blazed expression. rule 2-8 deals with applications, and implicitly variables and constants. if a variable and constant is applied to a sequence of arguments we deforest the arguments and blaze them.

$$\mathcal{T}[\![M^{\ominus}]\!] = (\mathcal{T}[\![M]\!])^{\ominus} \tag{1}$$

$$\mathcal{T}[\![v\ M_1 \dots M_n]\!] = v\ (\mathcal{T}[\![M_1]\!])^{\ominus} \dots (\mathcal{T}[\![M_n]\!])^{\ominus} \tag{2}$$

$$\mathcal{T}[\![c\ M_1 \dots M_n]\!] = c\ (\mathcal{T}[\![M_1]\!])^{\ominus} \dots (\mathcal{T}[\![M_n]\!])^{\ominus} \tag{3}$$

Rule 4 are kind of special. To ensure that the deforestation algorithm terminates, we consider function application where the function point is a f. Specifically this is a function defined in the environment. We consider the deforestation w.r.t. a set of newly defined functions generated in the process of deforestation called ϕ. first time we meet a function symbol we make a new function f' and add it to ϕ. We then end deforestation of the current term, but generate a new function which we deforest. If we have seen a function method before, we must identify if it resides in ϕ. Is this the case, then we are done with deforestation. In Section **??** we will give a more practical description of this, as it seems this is also where the implementation fails to meet the correctness of the algorithm.

$$\begin{aligned}
&\mathcal{T}[\![f\ M_1 \dots M_n]\!]\phi \\
&= f'\ v_1 \dots v_j \text{ if } (f' = \lambda v'_1 \dots v'_j.M) \in \phi \text{ and } (f\ M_1 \dots M_n) = [v_1/v'_1, \dots, v_j/v'_j]M \\
&\qquad \text{where } v_1 \dots v_j \text{ are free variables in } (f\ M_1 \dots M_n) \\
&= f''\ v_1 \dots v_j, \text{ otherwise} \\
&\qquad \text{where} \\
&\qquad f = M \\
&\qquad f'' = \lambda v_1 \dots v_j.(\mathcal{T}[\![M\ M_1 \dots M_n]\!]\phi') \\
&\qquad \phi' = \phi \cup \{f'' = \lambda v_1 \dots v_j.f\ M_1 \dots M_n\} \\
&\qquad v_1 \dots v_j \text{ are free variables of } f\ M_1 \dots M_n
\end{aligned} \tag{4}$$

rule 5-8 are applications of a lambda expression onto a sequence of applications. if either the variable is blazed or the argument is blazed then we cannot eliminate the argument and thus we must deforest it seperately and preserve the application. The deforest continues with the body of lambda and the rest of the arguments. for standard lambda expression we simply substitute the argument N_1 with v in M.

$$\mathcal{T}[\![(\lambda v.M)\ N_1^{\ominus} \dots N_n]\!] = (\lambda v.\mathcal{T}[\![M\ N_2 \dots N_n]\!])\ (\mathcal{T}[\![N_1]\!])^{\ominus} \tag{5}$$

$$\mathcal{T}[\![(\lambda v^{\ominus}.M)\ N_1 \dots N_n]\!] = (\lambda v.\mathcal{T}[\![M\ N_2 \dots N_n]\!])\ (\mathcal{T}[\![N_1]\!])^{\ominus} \tag{6}$$

$$\mathcal{T}[\![(\lambda v.M)\ N_1 \dots N_n]\!] = \mathcal{T}[\![[N_1/v]M\ N_2 \dots N_n]\!]) \tag{7}$$

$$(\lambda v.M) = (\lambda v.\mathcal{T}[\![M]\!]) \tag{8}$$

Rule 9 is much the same as for regular application

$$\mathcal{T}[\![\textbf{case}\ f\ M_1 \ldots M_n\ \textbf{of}\ p_1 \rightarrow N_1 | \ldots | p_n \rightarrow N_k]\!]\phi \tag{9}$$

$$= f'\ v_1 \ldots v_j\ \text{if}\ (f' = \lambda v_1' \ldots v_j'.M) \in \phi$$

$$\text{where } v_1 \ldots v_j \text{ are free variables in } (\textbf{case}\ f\ M_1 \ldots M_n\ \textbf{of}\ p_1 \rightarrow N_1 | \ldots | p_n \rightarrow N_k)$$

$$= f''\ v_1 \ldots v_j,\ \text{otherwise}$$

where

$f = M$

$f'' = \lambda v_1 \ldots v_j.(\mathcal{T}[\![\textbf{case}\ M\ M_1 \ldots M_n\ \textbf{of}\ p_1 \rightarrow N_1 | \ldots | p_n \rightarrow N_k]\!]\phi')$

$\phi' = \phi \cup \{f'' = \lambda v_1 \ldots v_j.(\textbf{case}\ M\ M_1 \ldots M_n\ \textbf{of}\ p_1 \rightarrow N_1 | \ldots | p_n \rightarrow N_k)\}$

$v_1 \ldots v_j$ are free variables of $(\textbf{case}\ M\ M_1 \ldots M_n\ \textbf{of}\ p_1 \rightarrow N_1 | \ldots | p_n \rightarrow N_k)$

If the selector is blazed, then we cannot eliminate the case expression and thus we must deforest all branches as well as the selector, as per rule 10.

$$\mathcal{T}[\![\textbf{case}\ M^{\ominus}\ \textbf{of}\ p_1 \rightarrow N_1 | \ldots | p_n \rightarrow N_k]\!] = \textbf{case}\ \mathcal{T}[\![M]\!]^{\ominus}\ \textbf{of}\ p_1 \rightarrow \mathcal{T}[\![N_1]\!] | \ldots | p_n \rightarrow \mathcal{T}[\![N_k]\!] \tag{10}$$

If the case selector is a variable applied to terms, then we likewise cannot eliminate and we convert to rule 10 and continue.

$$\mathcal{T}[\![\textbf{case}\ v\ M_1 \ldots M_n\ \textbf{of}\ p_1 \rightarrow N_1 | \ldots | p_n \rightarrow N_k]\!] = \mathcal{T}[\![\textbf{case}\ (v\ M_1 \ldots M_n)^{\ominus}\ \textbf{of}\ p_1 \rightarrow N_1 | \ldots | p_n \rightarrow N_k]\!] \tag{11}$$

Given the case selector is a constructor, we find the pattern that matches the case, and then we make a nesting of lambdas of the variables that occur and apply them to the arguments $M_1 \ldots M_n$.

$$\mathcal{T}[\![\textbf{case}\ c\ M_1 \ldots M_n\ \textbf{of}\ p_1 \rightarrow N_1 | \ldots | p_n \rightarrow N_k]\!] = \mathcal{T}[\![\lambda v_1 \ldots j.N_i]\!]\ \text{where}\ p_i = c\ v_1 \ldots v_j \tag{12}$$

Rule 13-15 works much the same as the rules for regular application. Rule 16 will flip a nested case expression inside out so to speak, by keeping the original selector but moving all branches of the outer case to be branches of inner cases with $N_1' \ldots N_j'$ as selectors.

$$\mathcal{T}[\![\textbf{case}\ (\lambda v.M)\ N_1^{\ominus} \ldots N_n\ \textbf{of}\ p_1 \rightarrow N_1' | \ldots | p_n \rightarrow N_k']\!] = (\lambda v.\mathcal{T}[\![\textbf{case}\ M\ N_2 \ldots N_n\ \textbf{of}\ p_1 \rightarrow N_1' | \ldots | p_n \rightarrow N_k']\!])\ (\mathcal{T}[\![N_1]\!])^{\ominus} \tag{13}$$

$$\mathcal{T}[\![\textbf{case}\ (\lambda v^{\ominus}.M)\ N_1 \ldots N_n\ \textbf{of}\ p_1 \rightarrow N_1' | \ldots | p_n \rightarrow N_k']\!] = (\lambda v.\mathcal{T}[\![\textbf{case}\ M\ N_2 \ldots N_n\ \textbf{of}\ p_1 \rightarrow N_1' | \ldots | p_n \rightarrow N_k']\!])\ (\mathcal{T}[\![N_1]\!])^{\ominus} \tag{14}$$

$$\mathcal{T}[\![\textbf{case}\ (\lambda v.M)\ N_1 \ldots N_n\ \textbf{of}\ p_1 \rightarrow N_1' | \ldots | p_n \rightarrow N_k']\!] = (\lambda v.\mathcal{T}[\![\textbf{case}\ [N_1/v]M\ N_2 \ldots N_n\ \textbf{of}\ p_1 \rightarrow N_1' | \ldots | p_n \rightarrow N_k']\!] \tag{15}$$

$$\begin{aligned}\mathcal{T}[\![\textbf{case } (\textbf{case } M \textbf{ of } p_1 \to N_1'|\dots|p_n \to N_j') \textbf{ of } p_1 \to N_1|\dots|p_n \to N_k]\!]\\ = \mathcal{T}[\![\textbf{case } M \textbf{ of}\\ p_1 \to \textbf{case } N_1' \textbf{ of } p_1 \to N_1|\dots|p_n \to N_k\\ \vdots\\ p_n \to \textbf{case } N_j' \textbf{ of } p_1 \to N_1|\dots|p_n \to N_k]\!] \quad (16)\end{aligned}$$

Some things to note about this algorithm is that the rules, does not work if there is a clash in name of free and bound variables. The current implementation does not handle this and it is thus up to the user to handle. The reason this is not fixed is that there are more grave problems with the code as we will see.

The code can be seen in Appendix **??**

5 Example

Ideally, I would have presented a deforestation attempt on a run-length encoder and decoder. However since my implementation does not work, we consider the example from [2] to see where it goes wrong. We use the definitions from **??** and want to deforest

fold (+) 0 (map square (until (> n) (repeat (+1) 1)))

As a spoiler, the paper originally defined the algorithm deforest the code to:

```
g 0 1 n
where
g = \a \m \n -> case (m > n)⊖ of
                  True -> a
                  False -> g (a + square m)⊖ (m + 1)⊖  n
```

Figure 9: Deforestation according to [2]

It should not be hard to see that these two definitions should be semantically equivalent but *g 0 1 n* contains no intermediate lists.

We first use rule 4, thus getting *g 0 1 n* as suggested, since we consider literals as variables. We further have that g must be defined by the 3 free variables 0 1 n. For readability, we call the variable representation of #0 and 1 for #1.

after applying rules 6,5 and 5 we have the body of g should be the deforestation of:

```
case map square (until (i > n) (repeat (+ 1) 1)) of
  Nil  -> #0
  Cons x xs ->  fold f (f #0 x)⊖ xs
```

continuing applying the rules we at some point have the following definitions, notice the different variables is renamed to not get name clashes.

```
#repeat = \#1 -> \#0 -> \g -> \f -> \p ->    (generated from free vars (fv))
    (\h⊖ -> (\c⊖ ->  case (p c⊖)⊖ of
                       True  -> #0
                       False  -> ?) #1⊖) (+ #1)⊖
#until = \n -> \#1 -> \#0 -> \g ->  \f -> (generated from fv)
```

```
    (\p⊖ -> #repeat #1 #0 g f p) (i > n)⊖
#map = \n -> \#1 ->  \#0 ->  \g -> (generated from fv)
    (\f⊖ -> #until n #1 #0 g f) square⊖
#fold = \#0 -> \n ->  \#1 -> (generated from free)
   (\g⊖ -> #map n #1 #0 g) (\x -> \y ->  (x + y))⊖
```

if we do some beta reduction on these we get:

```
#fold = \#0 -> \n -> \#1 -> case (#1 > n) of
                              True -> #0
                              False -> ?
```

Which looks very much like our target function in Figure**??**(modulo renaming). But the ? is where things go a little wrong.

The expression we consider, when filling ? is

```
fold g (g #0 (f c))⊖ (map f (until p (repeat h (h c)⊖)))
```

but this gives us the free variables $\{g, 0, f, c, p, h\}$, remember here the rules state nothing of substitution of variables, and we assume therefore this is an afterthought to get the program on a nice form and should not conflict with the algorithm. if we substitute into the expression we have:

```
fold (+) (#0 + square #1) (map square (until (> n) (repeat (+1) (#1 + 1))))
```

We can see that the second argument (#0 + square #1) is what we want as the first argument to #fold and n and (#1 + 1) also reside in the expression and we should be able to get them. This does not correspond to the #fold defined in ϕ and thus we must generate a new function. So we fill the question mark with:

```
##fold g #0 c f p h
```

We then again get to

```
case map f (until p (repeat h (h c)⊖)) of
  Nil  ->  a
  Cons x xs ->  fold g (g a x)⊖ xs
```

after rules 6,5,5. Again we cannot match and generate new functions. this happens for an entire round more, before a cycle starts to form. Thus I assume something goes wrong around here. But exactly what is the problem I am unfortunately not sure.

We can even further state that when considering

```
fold g (g #0 (f c))⊖ (map f (until p (repeat h (h c)⊖)))
```

for it to be converted appropriately into

```
#fold (g #0 (f c))⊖ (h #1) n
```

which when expanded would give the correct result, we should at some point earlier have bound these exact expressions to a free variable in this expression, but this cannot happen. Thus to be frank I am a bit confused if the rules presented in [2] is even valid.

Hence we leave the implementation broken as is.

5.1 Run length encoding?

Just as a little experiment, we look at runlength encoding and decoding to see what output it will give us, in this broken state. We consider the following definitions:

```
List a = Cons a (List a) | Nil;
Bool = True | False;
Pair a b = P a b;

map f x = case x of
            Nil -> Nil
            | Cons x xs -> Cons (f x) (map f xs);

take i xs = case i of
              0 -> Nil
              | n -> case xs of
                     Nil -> Nil
                     | Cons x xs -> Cons x (take (i-1) xs);

length as = case as of
              Nil -> 0
              | Cons a as -> 1 + length as;

head bs = case bs of
            Cons b bs -> b;

span p cs = case cs of
              Nil -> P Nil Nil
              | Cons c cs' -> case p c of
                               False -> P Nil cs
                               | True -> case span p cs' of
                                           P cs ds -> P (Cons c cs) ds;

groupBy y es = case es of
                 Nil -> Nil
                 | Cons e es -> case span (y e) es of
                                  P es fs -> Cons (Cons e es) (groupBy y fs);

group gs = groupBy (\xx -> \yy -> xx == yy) gs;

encode xs = map (\x -> P (length x)  (head x)) (group xs);

repeat h = Cons h (repeat h);

replicate i j = take i (replicate j);

append ks ls = case ks of
                 Nil -> ls
                 | Cons k ns -> Cons k (append ks ls);

concat ms = case ms of
              Nil -> Nil
              | Cons m ms -> append m (concat ms);


decode ns = concat (map (\o -> case o of P p q -> replicate p q) ns)
```

and deforesting the following

```
\rs -> decode (encode rs)
```

Will give us the output:

```
\rs -> #decode rs
where
  #decode :: \rs -> #concat rs
  #concat :: \rs -> (\ms -> ms⊖) (#map rs)⊖
  #map :: \rs ->  (\f⊖ -> #encode rs f) (\o -> o⊖)⊖
  #encode :: \rs -> \f -> ##map rs f
  ##map :: \rs -> \f -> (\f⊖ -> (\x -> x⊖) (#group rs)⊖) (\x⊖ ->
                                                       P (#length x)⊖ (#head
    x)⊖)⊖
  #head :: \x -> x⊖
  #length :: \x -> x⊖
  #group :: \rs -> #groupBy rs
  #groupBy :: \rs -> (\y⊖ -> rs⊖) ( xx  ->  yy  -> ((xx == yy))⊖)⊖
```

And by some beta-reduction we reach

```
\rs -> (\o -> o) rs
```

which means that even with the broken implementation we can in this case get a deforested representation, that in fact does no intermediate computation.

6 User interface

The idea behind doing this project, was to make a tool that other Program analysis and Transformation students could use to toy with and get a better feeling with deforestation as a program transformation, however as the deforestation implemented is broken either do to the algorithm or the implemementation, the tool is not all that useful. We do however still present the general interface, as it could be extended in the future. The tool serve as a repl with a similar interface to ghci. You have a prompt:

```
λ>
```

Here datatypes and functions can be declared as such:

```
λ> Tree a = Leaf a | Node a (Tree a) (Tree a);

λ> flip t = case t of Leaf -> t | Node v l r -> Node v (flip r) (flip l);
```

We then define a set of commands:

- eval: to evaluate and expression. This features is not currently implemented to a working extend.
- type: to get the type of an expression
- load: to load a file into the context.
- quit: to quit the repl.
- print: to pretty-print an expression, using Wadlers pretty printing style.
- deforest: to run deforestation on an expression.

A command is called by prefixing with :. so to deforest an expression one would type:

```
λ> :deforest decode (encode xs)
```

to deforest the decoding of the encoding of a defined list xs.

We have in this report presented a tool that allow users to explore deforestation of a small higher order functional language. Although small the language is expressible and provides similar constructors to that of Haskell core. This goes to show that the deforestation algorithm discussed here may be used in real world scenarios. We have presented a problem with the implementation, but we have also shown that an expression such as *decode (encode xs)* in fact gets deforested and will simply be the identity function. We have also presented the implementation for a type-inference. This was done to both show another form of program analysis and to algorithmically ensure the programs we consider to be well formed. As mentioned the tool is not complete but is given as open source for further development.

References

[1] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. "A Short Cut to Deforestation". In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. FPCA '93. Copenhagen, Denmark: Association for Computing Machinery, 1993, pp. 223–232. ISBN: 089791595X. DOI: 10.1145/165180.165214. URL: https://doi.org/10.1145/165180.165214.

[2] Geoff Hamilton. "Higher Order Deforestation". In: (January 1995).

[3] *Haskell core syn type*. URL: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/core-syn-type (visited on 06/19/2023).

[4] Mark P. Jones. "Typing Haskell in Haskell". In: 1999.

[5] Simon Marlow and Philip Wadler. "Deforestation for Higher-Order Functions". In: *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Berlin, Heidelberg: Springer-Verlag, 1992, pp. 154–165. ISBN: 3540198202.

[6] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(78)90014-4. URL: https://www.sciencedirect.com/science/article/pii/0022000078900144.

[7] Philip Wadler. "Deforestation: transforming programs to eliminate trees". In: *Theoretical Computer Science* 73.2 (1990), pp. 231–248. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(90)90147-A. URL: https://www.sciencedirect.com/science/article/pii/030439759090147A.

A Abstract Syntax

```
module AST where

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

```

```haskell
import Debug.Trace

data Expr = Var Var -- variable
          | Lit Literal
          | Lam Expr Expr
          -- | Let Bind Expr
          | App Expr Expr
          | Case Expr [Alt]
          | Con Name
          | Prim Op Expr Expr
          | Blazed Expr
  deriving(Show, Eq)

data Op = Add | Sub | Mul | Div
      | Lt | Gt | Eq | Neq | Leq | Geq
  deriving(Eq)

instance Show Op where
  show Add = "(+)"
  show Sub = "(-)"
  show Mul = "(*)"
  show Div = "(/)"
  show Lt = "(<)"
  show Gt = "(>)"
  show Eq = "(==)"
  show Neq = "(/=)"
  show Leq = "(<=)"
  show Geq = "(>=)"

data Pattern = ConAlt Name [Pattern]   -- Constructor Pattern
             | VarAlt Var
             | LitAlt Literal    -- literals
             | WildCard              -- wildcard
             | PBlazed Pattern
  deriving(Show, Eq)

type Alt = (Pattern, Expr)

type Var = String

data Literal = LInt Integer
             | LChar Char
             | LUnit
  deriving(Show, Eq)
type Name = String

type Import = (FileName, Prefix, Maybe [Var])
type FileName = String
type Prefix = String

type TyHead = (Name, [Name])

data UncheckedDecl = UTDecl TyHead [Pattern]
                   | UFDecl TyHead Expr
  deriving(Show)

data Decl = TDecl TyHead [Pattern]
          | FDecl Name Expr
  deriving(Show, Eq)

type Prog = [UncheckedDecl]

```

```

freeVarsOcc :: Expr -> Map Var Int
freeVarsOcc (Var v) = M.singleton v 1
freeVarsOcc (Lit _) = M.empty
freeVarsOcc (Lam v e) = freeVarsOcc e `M.difference` M.singleton (getVar v) 1
freeVarsOcc (App e1 e2) = M.unionWith (+) (freeVarsOcc e1) (freeVarsOcc e2)
freeVarsOcc (Case e alts) =
  let alts' = map (\(p, e) -> M.difference (freeVarsOcc e) (binders p)) alts
      maxed = foldl (M.unionWith max) M.empty alts'
  in M.unionWith (+) (freeVarsOcc e) maxed


freeVarsOcc (Con _) = M.empty
freeVarsOcc (Prim _ e1 e2) = M.unionWith (+) (freeVarsOcc e1) (freeVarsOcc e2
    )
freeVarsOcc (Blazed e) = freeVarsOcc e

binders :: Pattern -> Map Var Int
binders (VarAlt x) = M.singleton x 1
binders (ConAlt _ xs) = M.unions $ map binders xs
binders (PBlazed p) = binders p
binders _ = M.empty


getVar :: Expr -> Var
getVar (Var v) = v
getVar (Blazed e) = getVar e
getVar _ = error "getVar: not a variable"

getVarSafe :: Expr -> Maybe Var
getVarSafe (Var v) = Just v
getVarSafe (Blazed e) = getVarSafe e
getVarSafe _ = Nothing

fvs :: Expr -> [Var]
fvs = M.keys . freeVarsOcc

allVars :: Expr -> [Var]
allVars e = fvs e <> bvs e

bvs :: Expr -> [Var]
bvs (Var _) = []
bvs (Lit _) = []
bvs (Lam v e) = getVar v : bvs e
bvs (App e1 e2) = bvs e1 ++ bvs e2
bvs (Case e alts) =
  let e' = bvs e
      alts' = concatMap (\(p, e) -> M.keys (binders p) ++ bvs e) alts
  in e' ++ alts'
bvs (Con _) = []
bvs (Prim _ e1 e2) = bvs e1 ++ bvs e2
bvs (Blazed e) = bvs e

fresh' :: [Var] -> Var -> Var
fresh' vs v = if v `elem` vs then fresh' vs (v ++ "'") else v
```

B Preliminary sanity checks

```
{-# LANGUAGE LambdaCase #-}
module Check where
```

```haskell
import AST
import Data.Map (Map)
import qualified Data.Map.Strict as Map
import Control.Monad (when, (>=>))
import Data.Foldable (foldrM, foldr', foldl')
import Data.Either (either)
import Data.Bool
import Util

data ResolutionErr = ConfDef Var
                     | NotInScope Var
                     | WCNotAllowed
                     | LitNotAllowed Var
                     deriving(Show)

type Resolve b = Either ResolutionErr b

-- Checks that none of the arguments are equivalent -
-- both at value and type level are equivalent forall
checkDeclArg :: UncheckedDecl -> Resolve UncheckedDecl
checkDeclArg utd = case utd of
                     UTDecl (_, vs) _ -> check vs
                     UFDecl (_, vs) _ -> check vs
  where check vs = foldrM checkVar [] vs >> return utd
        checkVar :: Name -> [Name] -> Resolve [Name]
        checkVar v vs = bool (Left $ ConfDef v) (return $ v:vs) (v `notElem`
    vs)

-- Checks that the types are in scope
checkCon :: UncheckedDecl -> Resolve UncheckedDecl
checkCon utd@(UTDecl (_, vs) pats) = checkPats pats >> return utd
  where checkPats ps = mapM (`checkPat` vs) ps
        checkPat (ConAlt _ ps) _ = checkPats ps >> return ()
        checkPat (VarAlt v) vs = bool (Left $ NotInScope v) (return ()) (v `
    elem` vs)
-- Dont do anything for function Declarations
checkCon x = return x

-- Check that cases patterns dont use the same name
checkCase :: UncheckedDecl -> Resolve UncheckedDecl
checkCase ufd@(UFDecl _ body) = const (return ufd) =<< check body
  where check :: Expr -> Resolve ()
        check (Lam x b) = check b
        -- check (Let x b) = check (bindExpr x) >> check b
        check (App f x) = check f >> check x
        check (Case e alts) = check e >>
                              sequence (mapM (checkPat . fst) alts []) >>
                              mapM (check . snd) alts >> return ()
        check _ = return ()
        checkPat :: Pattern -> [Var] -> Resolve [Var]
        checkPat (ConAlt _ alts) seen = foldrM checkPat seen alts
        checkPat (VarAlt v) seen = bool (Left $ ConfDef v) (return $ v:seen)
    (v `notElem` seen)
        checkPat _ seen = return seen
-- no case expressions in types.
checkCase x = return x

-- checkRec :: Decl -> Resolve Decl
-- checkRec fd@(TDecl nm body) = return fd
-- checkRec fd@(FDecl nm body) = return $ bool fd (FDecl nm $ Fix . Lam (Var
    nm) $ body) (check body nm)
--   where check (Var x) nm = x == nm
--          -- check (Let _ b) nm = check b nm
--          check (Lam _ b) nm = check b nm
--          check (App f x) nm = check f nm || check x nm
--          check (Case e alts) nm = check e nm || any (flip check nm . snd)
    alts
--          check (Con _) nm = False
--          check _ _ = False


desugarArgs :: UncheckedDecl -> Resolve Decl
desugarArgs (UFDecl (nm,vs) body) = return $ FDecl nm $ mkLam body vs
desugarArgs (UTDecl h cons) = return $ TDecl h cons
  -- where constructor (ConAlt c args) = FDecl c $ mkLam $ foldl' vars []
    args
  --        vars (ConAlt _ args) vs = foldl' vars vs args
  --        vars (VarAlt v) vs = v:vs


toChecked :: UncheckedDecl -> Resolve Decl
toChecked = checkDeclArg >=> checkCon >=> checkCase >=> desugarArgs -- >=>
    checkRec
```

C Deforestation

```
{-# LANGUAGE LambdaCase #-}
module Deforest where

import Util
import AST
import qualified Data.Map as M
import Data.Foldable (foldl')
import Debug.Trace
import Data.List (nub)
import Control.Monad.RWS

import Pretty

blaze :: Expr -> Expr
blaze = \case
  Lam x e ->
    let x' = getVar x
    in if linear x' e
       then Lam x (blaze e)
       else Lam (Blazed x) $ blaze e

  e@(App e1 e2) ->
    let flat = flatten e
        (hd, tl) = (head flat, tail flat)
    in case hd of
      Con _ -> toTree $ map blaze flat
      _ -> toTree $ (blaze hd) : map (\x -> if compound x then Blazed $ blaze
     x else x) tl

  Case e alts ->
    let scrut = if compound e then Blazed e else e
        as = map (\(pat, alt) ->
                (subst pat $ M.mapWithKey (\k _ -> linear k alt) $ binders
    pat,
                 blaze alt)) alts
    in Case scrut as
    where
       subst (VarAlt x) env = case M.lookup x env of
                                Just False -> PBlazed $ VarAlt x
                                Just True -> VarAlt x
       subst (ConAlt c xs) env = ConAlt c $ map (\x -> subst x env) xs
       subst x env = x

  e@(Prim op e1 e2) ->
    let e1' = if compound e1 then Blazed e1 else e1
        e2' = if compound e2 then Blazed e2 else e2
    in Blazed $ Prim op e1' e2'
  -- do nothing to var, lit, con and already blazed
  x -> x


linear :: String -> Expr -> Bool
linear x e = case M.lookup x (freeVarsOcc e) of
               Just x -> not (x > 1)
               _ -> True


blaze_ :: Expr -> Expr
blaze_ = \case
  Blazed e -> Blazed $ e
  e -> Blazed $ e


type ForestEnv = M.Map String Expr

type ForestM = RWS ForestEnv [(Var, Expr)] Int

deforest :: ForestEnv -> Expr -> Expr
deforest env e =
  let (a,w) = evalRWS (deforest' e) env 0
  in trace (unlines $ map (\(nm, e) -> nm ++ " :: " ++ debug e) w) a

lit2Var :: Expr -> Expr
lit2Var = \case
  Lit (LInt i) -> Var $ "#" <> show i
  Lit (LChar c) -> Var $ "#$" <> show c
  Lit (LUnit) -> Var $ "#()"
  e -> e

literate :: Expr -> Expr
literate = \case
  Lit i -> lit2Var $ Lit i
  Lam x e -> Lam x $ literate e
  App e1 e2 -> App (literate e1) (literate e2)
  Case e alts -> Case (literate e) (map (\(pat, alt) -> (literate_pat pat,
    literate alt)) alts)
  Prim op e1 e2 -> Prim op (literate e1) (literate e2)
  Blazed e -> Blazed $ literate e
  e -> e

literate_pat :: Pattern -> Pattern
literate_pat = \case
  LitAlt (LInt i) -> VarAlt $ "#" <> show i
  LitAlt (LChar c) -> VarAlt $ "#$" <> show c
  LitAlt (LUnit) -> VarAlt $ "#()"
  VarAlt x -> VarAlt x
  ConAlt c xs -> ConAlt c $ map literate_pat xs
  PBlazed p -> PBlazed $ literate_pat p
```

```haskell

deforest' :: Expr -> ForestM Expr
deforest' = \case
  Blazed e -> blaze_ <$> deforest' e

  Var x -> return $ Var x -- $ do env <- ask
              -- case M.lookup x env of
              --   Just e -> deforest' e
              --   _ -> return $ Var x
  Lit l -> return $ lit2Var $ Lit l
  Con c -> return $ Con c

  Prim op e1 e2 -> do
    e1' <- deforest' e1
    e2' <- deforest' e2
    return $ Prim op e1' e2'

  Lam x e -> Lam x <$> deforest' e

  e@(App e1 e2) ->
    let flat = flatten e
        (hd, tl) = (head flat, tail flat)
    in case hd of
      -- rule 3
      Con c -> do
        ls <- trace("rule3") $ mapM (\x -> blaze_ <$> deforest' x) tl
        return $ toTree (hd:ls)

      Var x -> do env <- ask
                  case M.lookup x env of
                    -- rule2
                    Nothing -> do
                      ls <- trace("rule 2") $ mapM (\x -> blaze_ <$> deforest' x) tl
                      return $ toTree (hd:ls)
                    -- rule4
                    Just e' -> do
                      fvs <- fvforest e
                      trace "rule 4" $ handleF fvs x x e (toTree $ e' : tl)
      -- rule 5b
      Lam (Blazed x') e0 -> do
        let x = getVar x'
        -- let l' = Lam (Blazed . Var )
        let (e1, es) = (head tl, tail tl)
        lam <- trace "rule 5b" $ Lam (Blazed (Var x)) <$> (deforest' $ toTree $ e0 : es)
        App lam <$> (blaze_ <$> deforest' e1)

      Lam x' e0 ->
        let x = getVar x'
            (e1, es) = (head tl, tail tl)
        in case e1 of
          -- rule 5a
          Blazed e1 -> do
            lam <- trace "rule 5a" $ Lam (Var x) <$> (deforest' . toTree $ e0 : es)
            App lam <$> (blaze_ <$> deforest' e1)
          -- rule 5c (n /= 0)
          _ -> do
              let e1' = lit2Var e1
              let sub = subst x e1' e0
              trace ("rule 5c: " ++ debug e ++ "\n\n" ++ debug sub) $
    deforest'  . toTree $ sub : es

  -- rule 6
  e@(Case (Blazed e0) alts) -> do
    e0' <- trace ("rule 6") $ blaze_ <$> deforest' e0
    as <- mapM (\(pat, e') -> deforest' e' >>= \x -> return (pat, x)) alts
    return $ Case e0' as

  -- rules 7-11
  e@(Case e0 alts) ->
    let e0' = flatten e0
        (hd, case_es) = (head e0', tail e0')
    in case hd of
      -- rule 8
      Con c -> do
        case matchCon c case_es alts of
          Just e' -> trace ("rule 8 " ++ debug e ++ "\n" ++ show case_es ++
    "\n" ++ debug e') $ deforest' . toTree $ e' : case_es


      Var x -> do env <- ask
                  case M.lookup x env of
                    -- rule 7
                    Nothing -> trace "rule 7" $ deforest'  . blaze_ $ e0

                    -- rule 9
                    Just e' -> do
                      fvs <- fvforest e
                      trace "rule 9" $ handleF fvs x x e (Case (toTree (e' :
    case_es)) alts)
      -- rule 10b
      Lam (Blazed x') e0 -> do
        let x = getVar x'
        let (e1, es) = (head case_es, tail case_es)
        let case' = Case (toTree $ e0 : es) alts -- tail is is e2..en
        lam <- trace "rule 10b" $ Lam (Blazed (Var x)) <$> deforest' case'
        App lam  <$> (blaze_ <$> deforest' e1)

      Lam x' e0 ->
        let x = getVar x'
            (e1, es) = (head case_es, tail case_es)
        in case e1 of
          -- rule 10a
          Blazed e1 -> do
            let case' = Case (toTree $ e0 : es) alts
            lam <- trace "rule 10a" $ Lam (Var x) <$> deforest'  case'
            App lam <$> (blaze_ <$> deforest' e1)

          -- rule 10c
          _ -> trace "rule 10c" $ deforest' $ Case (toTree $ subst x e1 e0 :
    es) alts

      -- rule 11
      Case e0 alts' -> do
        let outer_e = map (\(p, e) -> (p, Case e alts)) alts'
        trace ("rule 11 " ++ debug e ++ "\n" ++ debug (Case e0 outer_e))  $
    deforest' $ Case e0 outer_e



subst :: String -> Expr -> Expr -> Expr
subst x m = \case
```

```haskell
  Var y -> if x == y then m else Var y
  Prim op e1 e2 -> Prim op (subst x m e1) (subst x m e2)
  Lam y e -> let y' = getVar y
             in if y' == x then Lam y e
                else if y' `elem` (fvs m) then
                  let banlist = fvs m <> allVars e
                  in subst x m $ rename (fresh' banlist y') $ Lam y e
             else Lam y $ subst x m e
  App e1 e2 -> App (subst x m e1) (subst x m e2)
  Case e alts ->
    let binds = map (binders . fst) alts
        subs' = map (\(binds, (p, e')) -> if M.member x binds then (p, e') else
                                     (p, subst x m e')) $ zip binds alts
    in Case (subst x m e) subs'
  Blazed e -> Blazed $ subst x m e
  e -> e

rename :: String -> Expr -> Expr
rename x (Lam y e) = let y' = getVar y in Lam (Var x) $ subst y' (Var x) e

basicname :: String -> String
basicname ('#':xs) = basicname xs
basicname xs = xs

fvforest :: Expr -> ForestM [Var]
fvforest e = do env <- ask
                let (fvs, fs) = go e
                -- return $ filter (\x -> M.notMember x env && x `notElem` fs) $ nub fvs
                return $ filter (\x -> M.notMember x env) $ nub fvs
  where
    go :: Expr -> ([Var], [Var])
    go = \case
       Blazed e -> go e
       Var x -> ([x], [])
       Lit (LInt i) -> (["#" ++ show i], [])
       Lit (LChar c) -> (["#$" ++ show c], [])
       Lit LUnit -> (["#()"], [])
       Con _ -> ([], [])
       Prim _ e1 e2 ->
         let (e1', f1s) = go e1
             (e2', f2s) = go e2
          in (e1' <> e2', f1s <> f2s)
       Lam x e -> let x'= getVar x in let (e', fs) = go e in (filter (/=x') e', fs)
       App e1 e2 -> case getVarSafe e1 of
                      Just v -> let (e', fs) = go e2 in (e', v:fs)
                      Nothing ->
                        let (e1', f1s) = go e1
                            (e2', f2s) = go e2
                        in (e1' <> e2', f1s <> f2s)
       Case e alts ->
         let (e', fvs) = go e
             (es', fs) = foldr1 (\(e1, f1) (e2, f2) -> (e1 <> e2, f1 <> f2)) $ map (\(p, e) ->
                          let (fvs, fs) = go e
                          in (filter (`notElem` M.keys (binders p)) fvs, fs)) alts
         in (e' <> es', fvs <> fs)

getLambdas :: Expr -> Int -> ([Var], Expr)
getLambdas e 0 = ([], e)
getLambdas (Lam x e) n = let x' = getVar x
                             (xs, e') = getLambdas e (n-1)
                         in (x':xs, e')
getLambdas e _ = ([], e)

-- fvs are the free variables in the expression to deforest
-- f is the original function name
-- f_cur is the current name
-- e is the original definition
-- e' is the e for f
handleF :: [Var] -> Var -> Var -> Expr -> Expr -> ForestM Expr
handleF fvs f f_cur' e e' = do
  let f_cur = "#" <> f_cur'
  env <- ask
  case M.lookup f_cur env of
    Just e'' -> do
      let (fvs',inner_e) = getLambdas e'' $ length fvs
      let e_check = foldr (\(v',v) acc -> subst v (Var v) acc) inner_e $ zip
    fvs' fvs
      if e == e_check then
        trace ("wtf" ++ debug e) $ return $ toTree $ (Var f_cur) : map Var
    fvs
      else do
        trace ("damn") $ handleF fvs f f_cur e e'
    _ -> do
      modify (+1)
      st <- get
      if st > 10 then trace "error" $ return $ Lit LUnit
      else do
        let f' = mkLam e fvs
        e'' <- trace(f_cur ++ debug e' ++ "\n" ++ debug f') $ local (M.insert
     f_cur f') (deforest' $ e')
        let f'' = mkLam e'' fvs
        trace ("huh?" ++ debug e'') $ tell [(f_cur, f'')]
        return $ toTree (Var f_cur : map Var fvs)

matchCon :: Name -> [Expr] -> [AST.Alt] -> Maybe Expr
matchCon c es [] = error $ "matchCon: " ++ show c ++ " " ++ show es
matchCon c es ((pi, ei):alts) =
    case matchF (Con c) es pi of
      Nothing -> matchCon c es alts
      Just env ->
        let vs = M.keys env
            lam = mkLam ei vs
        in trace ("matcon" ++ debug pi ++ show vs) $ return lam

matchF :: Expr -> [Expr] -> Pattern -> Maybe (M.Map Name Expr)
matchF _ _ WildCard = return mempty
matchF (Lit (LInt i)) _ (LitAlt (LInt i')) | i == i' = return mempty
matchF (Lit (LChar c)) _ (LitAlt (LChar c')) | c == c' = return mempty
matchF (Lit LUnit) _ (LitAlt LUnit) = return mempty
matchF v es (PBlazed e) = matchF v es e
matchF v [] (VarAlt x) = return $ M.singleton x v
matchF (Con c) es (ConAlt c' ps) =
  if c == c' then foldl' merge (Just mempty) $ zipWith (\x y -> matchF x [] y
    ) es ps
  else Nothing
  where merge Nothing _ = Nothing
        merge _ Nothing = Nothing
        merge (Just env) (Just env') = return $ env `M.union` env'
matchF _ _ _ = Nothing
```

D Type inference

```haskell
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE LambdaCase #-}
module Type where

import AST
import Data.List (nub, union)
import Util

data TVar = TV String Kind
  deriving (Eq, Ord, Show)

data TCon = TC String Kind
  deriving (Eq, Ord, Show)

data Kind = Star
  | Kind :-> Kind
  deriving (Eq, Ord, Show)

infixr 4 :->

data Ty
  = TVar TVar
  | TCon TCon
  | TAp Ty Ty
  | TGen Int
  deriving (Eq, Ord, Show)

infixr      4 `fn`
fn         :: Ty -> Ty -> Ty
a `fn` b    = TAp (TAp tArrow a) b

data Scheme = Forall [Kind] Ty
  deriving (Eq, Ord, Show)

toScheme :: Ty -> Scheme
toScheme t = Forall [] t

fvTy :: Ty -> [TVar]
fvTy (TVar a)   = [a]
fvTy (TAp t s)  = fvTy t `union` fvTy s
fvTy _ = mempty

tInt, tChar, tUnit, tList, tArrow :: Ty
tInt  = TCon $ TC "Int" Star
tChar = TCon $ TC "Char" Star
tUnit = TCon $ TC "()" Star
tList = TCon $ TC "[]" (Star :-> Star)
tBool = TCon $ TC "Bool" Star
tArrow = TCon $ TC "(->)" (Star :-> Star :-> Star)

class IsFn t where
  isFn :: t -> Bool

instance IsFn Ty where
  isFn (TAp (TAp (TCon (TC "(->)" _)) _) _) = True
  isFn _ = False

instance IsFn Scheme where
  isFn (Forall _ t) = isFn t
```

```haskell

tOp :: Op -> Ty
tOp Add = tInt `fn` tInt `fn` tInt
tOp Sub = tInt `fn` tInt `fn` tInt
tOp Mul = tInt `fn` tInt `fn` tInt
tOp Div = tInt `fn` tInt `fn` tInt
tOp Lt = tInt `fn` tInt `fn` tBool
tOp Gt = tInt `fn` tInt `fn` tBool
tOp Eq = tInt `fn` tInt `fn` tBool
tOp Neq = tInt `fn` tInt `fn` tBool
tOp Leq = tInt `fn` tInt `fn` tBool
tOp Geq = tInt `fn` tInt `fn` tBool

class HasKind t where
  kind :: t -> Kind

instance HasKind TVar where
  kind (TV _ k) = k

instance HasKind TCon where
  kind (TC _ k) = k

instance HasKind Ty where
  kind (TVar v)   = kind v
  kind (TCon c)   = kind c
  kind (TAp t s)  = case kind t of _ :-> k -> k; k -> error $ "kind error: "
    ++ show (TAp t s) ++ "kind " ++ show k
```

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
module TyEnv where

import AST
import Type
import Subst

newtype TyEnv = TyEnv { tys :: [(Name, Scheme)] }
  deriving (Eq, Show, Semigroup, Monoid)

instance Substitutable TyEnv where
  apply s (TyEnv env) = TyEnv $ fmap (\(a, t) -> (a, apply s t)) env

  ftv (TyEnv env) = ftv $ fmap snd env

empty :: TyEnv
empty = TyEnv []

extend :: Name -> Scheme -> TyEnv -> TyEnv
extend x t (TyEnv tys) = TyEnv ((x, t) : tys)

remove :: Name -> TyEnv -> TyEnv
remove x (TyEnv tys) = TyEnv (filter ((/= x) . fst) tys)

lookup :: Name -> TyEnv -> Maybe Scheme
lookup x (TyEnv tys) = Prelude.lookup x tys

merge :: TyEnv -> TyEnv -> TyEnv
merge (TyEnv tys1) (TyEnv tys2) = TyEnv (tys1 ++ tys2)

-- mergeMany :: [TyEnv] -> TyEnv
-- mergeMany = foldr merge new

singleton :: Name -> Scheme -> TyEnv
singleton x t = TyEnv [(x, t)]

keys :: TyEnv -> [Name]
keys (TyEnv tys) = map fst tys

-- generalize :: TyEnv -> Ty -> Scheme
-- generalize env t = Forall as t
--   where as = Set.toList $ ftv t `Set.difference` ftv env
```

```haskell
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
module Subst where

import Type
import Data.List (nub, union)

newtype Subst = Subst { subst :: [(TVar, Ty)] }
  deriving (Eq, Show, Ord, Semigroup, Monoid)

(@@) :: Subst -> Subst -> Subst
(@@) (Subst s1) (Subst s2) = Subst $ [(u, apply (Subst s1) t) | (u, t) <- s2]
     <> s1

(+->) :: TVar -> Ty -> Subst
(+->) a t = Subst [(a, t)]

class Substitutable a where
  apply :: Subst -> a -> a
  ftv   :: a -> [TVar]

instance Substitutable Ty where
  apply (Subst s) t@(TVar a) = maybe t id (Prelude.lookup a s)
  apply s (TAp a b) = TAp (apply s a) (apply s b)
  apply s t = t

  ftv = fvTy

instance Substitutable Scheme where
  apply s (Forall ks t) = Forall ks $ apply s t

  ftv (Forall ks t) = ftv t

instance Substitutable a => Substitutable [a] where
  apply = fmap . apply
  ftv   = nub . concatMap ftv
```

```haskell
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE LambdaCase #-}
module Infer where

import TyEnv
import Type
import Subst
import AST
import Util

import Control.Monad.Except
import Control.Monad.RWS
import Control.Monad.State
import Control.Monad.Reader

import qualified Data.Set as Set

import Debug.Trace

type Infer a = RWST
  TyEnv
  [Constraint]
  InferState
  (Except InferErr)
  a

newtype InferState = InferState { count :: Int }

initInfer :: InferState
initInfer = InferState { count = 0 }

data InferErr =
    Unified Ty Ty
  | Infinite TVar Ty
  | UnboundVariable String
  | KindMismatch Ty Ty
  | Ambigious [Constraint]

-- | Run the inference monad
runInfer :: TyEnv -> Infer Ty -> Either InferErr (Ty, [Constraint])
runInfer env m = runExcept $ evalRWST m env initInfer

inferFDecl :: TyEnv -> Decl -> Either InferErr Scheme
inferFDecl env (FDecl f m) = do
  (ty,cs) <- runInfer env $ do
                tv <- fresh Star
                let sc = toScheme tv
                local (extend f sc) $ infer m
  subst <- runSolve cs
  let t = apply subst ty
  return $ (quantify (ftv t) t)

constructTyCon :: TyEnv -> Decl -> TyEnv
constructTyCon tenv (TDecl (tcon, args) pats)
  | tcon `elem` keys tenv = error $ "type constructor " ++ tcon ++ " already
    defined"
  | otherwise =
    let tcon' = TCon $ TC tcon $ foldr (:->) Star (map (const Star) args)
        tvs = map (\x -> TV x Star) args
        tvars = map TVar tvs
        basic = foldl TAp tcon' tvars
        tenv' = extend tcon (quantify tvs basic) tenv
        pats' = TyEnv $ map (\(ConAlt nm ps) ->
                       (nm, quantify tvs $ foldr fn basic (map (patternToTy
    tenv') ps))) pats
    in pats' `merge` tenv

inferExpr :: TyEnv -> Expr -> Either InferErr Scheme
inferExpr env m = do
  (ty,cs) <- runInfer env (infer m)
  subst <- runSolve cs
  let t = apply subst ty
  return $ quantify (ftv t) t

tryFind :: Name -> Infer Ty
tryFind x = do
  env <- ask
  case TyEnv.lookup x env of
```

```
    Nothing -> throwError $ UnboundVariable x
    Just t -> inst t

quantify :: [TVar] -> Ty -> Scheme
quantify vs t = Forall ks (apply s t)
  where vs' = [ v | v <- ftv t, v `elem` vs ]
        ks  = fmap kind vs'
        s   = Subst $ zip vs' (map TGen [0..])


patternToTy :: TyEnv -> Pattern -> Ty
patternToTy env = \case
  LitAlt (LInt _) -> tInt
  LitAlt (LChar _) -> tChar
  LitAlt LUnit -> tUnit

  VarAlt v -> case TyEnv.lookup v env of
    Just (Forall _ t) -> t
    Nothing -> TVar $ TV v Star
  ConAlt c ps -> case TyEnv.lookup c env of
    Just (Forall _ t) -> t
    Nothing -> error $ "constructor " ++ c ++ " not found in patternToTy"

  WildCard -> error "wildcard in patternToTy"

inst :: Scheme -> Infer Ty
inst (Forall ks t) = do
  ts <- mapM fresh ks
  return $ instantiate ts t

class Instantiate a where
  instantiate :: [Ty] -> a -> a

instance Instantiate Ty where
  instantiate ts (TAp l r) = TAp (instantiate ts l) (instantiate ts r)
  instantiate ts (TGen n) = ts !! n
  instantiate _ t = t

instance Instantiate a => Instantiate [a] where
  instantiate ts = map (instantiate ts)


fresh :: Kind -> Infer Ty
fresh k = do
  s <- get
  put s { count = count s + 1 }
  return $ TVar $ TV (letters !! count s) k


infer :: Expr -> Infer Ty
infer = \case
  Lit (LInt _) -> return tInt
  Lit (LChar _) -> return tChar
  Lit LUnit -> return tUnit

  Var x -> tryFind x

  Lam x m -> do
    tv <- fresh Star
    -- for now we only allow variables in the lambda
    x' <- case x of
      (Var x) -> return x
```

```haskell
      _ -> throwError $ UnboundVariable "lambda"
    t <- local (extend x' $ toScheme tv) $ infer m
    return $ tv `fn` t

  App m n -> do
    t1 <- infer m
    t2 <- infer n
    tv <- fresh Star
    tell [t1 :~: (t2 `fn` tv)]
    return tv

  Case m alts -> do
    tv <- fresh Star
    t <- infer m
    inferAlts alts tv t

  Con c -> tryFind c

  -- Fix m -> do
  --   t <- infer m
  --   tv <- fresh Star
  --   tell [t :~: (tv `fn` tv)]
  --   return tv

  Prim op m n -> do
    t1 <- infer m
    t2 <- infer n
    tv <- fresh Star
    let t = t1 `fn` t2 `fn` tv
    tell [t :~: (tOp op)]
    return tv

inferPat :: Pattern -> Infer (Ty, TyEnv)
inferPat = \case
  LitAlt (LInt _) -> return (tInt, mempty)
  LitAlt (LChar _) -> return (tChar, mempty)
  LitAlt LUnit -> return (tUnit, mempty)

  VarAlt v -> do
    tv <- fresh Star
    return (tv, extend v (toScheme tv) mempty)

  ConAlt c ps -> do
   (ts, envs) <- inferPats ps
   t' <- fresh Star
   t <- tryFind c
   tell [t :~: foldr fn t' ts]
   return (t', envs)

  WildCard -> do
    tv <- fresh Star
    return (tv, mempty)

inferPats :: [Pattern] -> Infer ([Ty], TyEnv)
inferPats ps = do
  x <- mapM inferPat ps
  let ts = map fst x
      envs = map snd x
  return (ts, foldr merge mempty envs)

inferAlt :: Ty -> AST.Alt -> Infer Ty
inferAlt t0 (p, m) = do
```

```haskell
  (ts, env) <- inferPat p
  tell [t0 :~: ts]
  t' <- local (merge env) $ infer m
  return $ t'

inferAlts :: [AST.Alt] -> Ty -> Ty -> Infer Ty
inferAlts alts t t0 = do
  ts <- mapM (inferAlt t0) alts
  tell $ map (t :~:) ts
  return t



-- Constraint Solving

data Constraint = Ty :~: Ty
  deriving (Show)

instance Substitutable Constraint where
  apply s (t1 :~: t2) = apply s t1 :~: apply s t2

  ftv (t1 :~: t2) = ftv t1 <> ftv t2

type Unif = (Subst, [Constraint])

type Solver a = Except InferErr a

runSolve :: [Constraint] -> Either InferErr Subst
runSolve cs = runExcept $ solve (mempty, cs)

solve :: Unif -> Solver Subst
solve (sub, []) = return sub
solve (sub, (t1 :~: t2) : cs) = do
  sub' <- unify (apply sub t1) (apply sub t2)
  solve (sub' @@ sub, apply sub' cs)

unify :: Ty -> Ty -> Solver Subst
unify (TVar v) t = bind v t
unify t (TVar v) = bind v t
unify (TAp a b) (TAp a' b') = do
  s <- unify a a'
  s' <- unify (apply s b) (apply s b')
  return (s' @@ s)
unify t1 t2 | t1 == t2 = return mempty
            | otherwise =  throwError $ Unified t1 t2

bind :: TVar -> Ty -> Solver Subst
bind v t | t == TVar v = return mempty
         | v `elem` ftv t = throwError $ Infinite v t -- occurs check
         | kind v /= kind t = throwError $ KindMismatch (TVar v) t
         | otherwise = return $ v +-> t
```

E Parsing

```haskell
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE NamedFieldPuns #-}
{-# LANGUAGE FlexibleContexts #-}
module Parser where

import Control.Monad ( liftM2, void )
import           Data.Char           (isLower, isUpper)
import           Data.Function       (on)
import           Text.Parsec
import qualified Text.Parsec.Token as P
import Control.Monad.Combinators.Expr
import AST
import Data.Bool
import Data.Foldable (Foldable(foldr'))

type Parser a = Parsec String () a

parseFiles :: String -> Either String [Import]
parseFiles = apihelper parseLoad

parseLoad :: Parser [Import]
parseLoad = semiSep $ do f <- filename; inc <- incl; p <- prefix; return (f,
    p, inc)
  where incl = optionMaybe $ braces (many ident)
        prefix = option "" $ do reserved "as";
                                i <- ident;
                                bool (fail "Prefix must be Capitalized")
                                     (return i)
                                     (isCon i)

parseStringDecls :: String -> Either String Prog
parseStringDecls = apihelper $  declP `sepEndBy` (P.semi lexer)

apihelper p s = case parse (p <* eof) "" s of
                      Right x -> return x
                      Left e -> Left $ "parse-error:\n " <> show e

declP :: Parser UncheckedDecl
declP = do
  i <- ident
  args <- many ident
  op "="
  bool (UFDecl (i, args) <$> top_exprP)
       (UTDecl (i,args) <$> sepBy1 altP (op "|"))
       (isCon i)

consP :: Parser Pattern
consP = do
  i <- ident
  bool (fail "Constructor must be Capitalized")
       (ConAlt i <$> argsP) (isCon i)
  where
    varP = do v <- ident;
              bool (fail "var must be lowerletter")
                   (return $ VarAlt v) (not . isCon $ v)
    argsP = many (parens consP <|> varP)

isCon :: String -> Bool
isCon = isUpper . head

parseTopTerm :: String -> Either String Expr
parseTopTerm = apihelper top_exprP

style :: P.LanguageDef st
style = P.LanguageDef
  { P.commentStart = "{-"
  , P.commentEnd = "-}"
  , P.commentLine = "--"
  , P.nestedComments = True
  , P.identStart = letter
  , P.identLetter = alphaNum <|> oneOf "_'"
  , P.opStart = P.opLetter style
  , P.opLetter = oneOf ":!#$%&*+./<=>?@\\^|-~"
  , P.reservedOpNames = ["::","..","=","\\","|","<-","->","@","~","=>"]
  , P.reservedNames = [ "case", "of"
                      , "let", "in"
                      ]
  , P.caseSensitive = True
  }

-- expressions
appP, top_exprP, exprP, varConP, varP, litP, lamP,  caseP :: Parser Expr

appP = exprP >>= \x ->
  try (many1 exprP >>= \xs -> return $ foldl App x xs)
  <|> return x

top_exprP = makeExprParser appP table

table   = [
          [ binary "*" Mul, binary "/" Div ]
          , [ binary "+" Add, binary "-" Sub ]
          , [ binrel "<" Lt, binrel "<=" Leq, binrel ">" Gt, binrel ">=" Geq, binrel "==" Eq, binrel "/=" Neq]
          ]

binrel  name fun = InfixN (do { op name; return $ \e1 e2 -> Prim fun e1 e2 })
binary  name fun = InfixL (do { op name; return $ \e1 e2 -> Prim fun e1 e2 })

exprP = choice [ litP
               , lamP
               , caseP
               , varConP
               , parens top_exprP
               ]

varConP = do
  i <- ident
  return $ bool (Var i) (Con i) (isCon i)

varP = Var <$> ident

litP = Lit <$> lit

lit :: Parser Literal
lit = choice [ LInt <$> integer
             -- , try $  symbol "(" >> char '-' >> (LInt . negate) <$> integer <* symbol ")"
             , LChar <$> ticks (alphaNum <|> oneOf "_")
             , op "()" >> return LUnit
             ]

lamP = do
  op "\\"; args <- many1 varP; op "->"; body <- top_exprP; return $ go body args
  where go = foldr' Lam

-- letP = do
--   reserved "let"; bindId <- varP; op "="; bindExpr <- appP; reserved "in";
--   Let (NonRec {bindId, bindExpr}) <$> appP
```

```

caseP = do
  t0 <- between (reserved "case") (reserved "of") top_exprP
  Case t0 <$> sepBy1 alts (op "|")
  where alts = do p <- altP; op "->"; t <- top_exprP; return (p, t)

altP :: Parser Pattern
altP = (LitAlt <$> lit) <|> idaltP <|> wildcard <|> parens altP
  where
      idaltP = do i <- ident;
                  bool (return $ VarAlt i) (ConAlt i <$> many altP) (isCon i)
      wildcard = op "_" >> return WildCard



lexer = P.makeTokenParser style

ident :: Parser String
ident = P.identifier lexer

filename :: Parser String
filename = P.lexeme lexer $ many1 (alphaNum <|> oneOf "-._/~")

parens, braces, ticks :: Parser a -> Parser a

parens = P.parens lexer

braces = P.braces lexer

ticks x = (between `on` char) '\'' '\'' x <* P.whiteSpace lexer

semiSep = P.semiSep lexer

integer :: Parser Integer
integer = P.natural lexer

reserved, op, symbol :: String -> Parser ()
reserved = P.reserved lexer
op = P.reservedOp lexer
symbol a = P.symbol lexer a >> return ()
```

F Pretty printing

```
{-# LANGUAGE FlexibleInstances, LambdaCase #-}
module Pretty where

import Type
import AST
import Data.Map (Map)
import qualified Data.Map.Strict as M
import Data.List (intercalate)
import Prettyprinter
import Prettyprinter.Util
import Infer (InferErr(..), Constraint(..))
import Check (ResolutionErr(..))
import Util (letters, compound)
import Eval (RuntimeErr(..), Value(..))

mkPretty :: Pretty e => e -> IO ()
mkPretty e = putDocW 80 (pretty e) >> putStrLn ""
```

```

debug :: Pretty e => e -> String
debug = show . pretty

data TypeOf = MkTo Expr Scheme

instance Pretty TypeOf where
  pretty (MkTo n t) = pretty n <+> pretty "::" <+> pretty t

instance Pretty Literal where
  pretty (LInt i) = pretty i
  pretty (LChar c) = pretty c
  pretty (LUnit) = pretty "()"

instance Pretty Op where
  pretty = \case
    Add -> pretty "+"
    Sub -> pretty "-"
    Mul -> pretty "*"
    Div -> pretty "/"
    -- Mod -> pretty "%"
    Eq -> pretty "=="
    Neq -> pretty "/="
    Lt -> pretty "<"
    Gt -> pretty ">"
    Leq -> pretty "<="
    Geq -> pretty ">="
    -- And -> pretty "&&"
    -- Or -> pretty "||"
    -- Not -> pretty "not"
    -- Neg -> pretty "negate"

instance Pretty Expr where
  pretty = \case
    Var v -> pretty v
    Lit l -> pretty l
    Lam x e -> hang 2 (pretty "  " <> pretty x <+> pretty "->" <+> softline <> pretty e)
    App f@(Lam x b) e -> wrap (const True) f <+> wrap compound e
    App f e -> pretty f <+> wrap compound e
    Case e alts -> hang 2 (pretty "case" <+> pretty e <+> pretty "of"
                  <> hardline <> (vsep (map prettyAlt alts)))
    Con c -> pretty c -- <+> hsep (map pretty es)
    Prim op m n -> pretty "(" <> pretty m <+> pretty op <+> pretty n <> pretty ")"
    Blazed e -> wrap compound e <> pretty "   "


wrap :: Pretty a => (a -> Bool) -> a -> Doc ann
wrap f x = if f x then pretty "(" <> pretty x <> pretty ")" else pretty x

instance Pretty Pattern where
  pretty = \case
    VarAlt v -> pretty v
    LitAlt l -> pretty l
    WildCard -> pretty "_"
    ConAlt n ps -> group (pretty n <+> hsep (map pretty ps))
    PBlazed p -> pretty p <> pretty "   "

  -- We wanna split after -> if not possible to have
prettyAlt :: Alt -> Doc ann
prettyAlt (p, e) = hang 2 (pretty p <+> pretty "->" <+> softline <> (pretty e
```

```haskell
    ))

instance Pretty TVar where
  pretty (TV v k) = pretty v

instance Pretty TCon where
  pretty (TC v k) = pretty v

instance Pretty Ty where
  pretty = \case
    TVar v -> pretty v
    TCon c -> pretty c
    TGen i -> pretty $ letters !! i
    TAp (TAp (TCon (TC "(->)" _)) t1) t2 -> wrap nested t1 <+> pretty "->" <+> pretty t2
    TAp t1 t2 -> pretty t1 <+> pretty t2

nested :: Ty -> Bool
nested = \case
  TAp (TAp (TCon (TC "(->)" _)) _) _ -> True
  TAp t1 t2 -> nested t1 || nested t2
  _ -> False

instance Pretty Scheme where
  pretty (Forall [] t) = pretty t
  pretty (Forall vs t) = pretty "   " <> hsep (map (pretty . snd) $ zip vs letters) <> pretty "." <+> pretty t

instance Pretty Constraint where
  pretty (t1 :~: t2) = pretty t1 <+> pretty "~" <+> pretty t2

instance Pretty InferErr where
  pretty (Unified t1 t2) = pretty "Cannot unify " <+> pretty t1 <+> pretty " with" <+> pretty t2
  pretty (Infinite v t) = pretty "Infinite type:" <+> pretty v <+> pretty "=" <+> pretty t
  pretty (UnboundVariable x) = pretty "Unbound variable:" <+> pretty x
  pretty (KindMismatch t1 t2) = pretty "Kind mismatch:" <+> pretty t1 <+> pretty t2
  pretty (Ambigious cs) = pretty "Ambigious constraints:" <+> pretty cs

instance Pretty ResolutionErr where
  pretty (ConfDef x) = pretty "Conflicting definitions for" <+> pretty x
  pretty (NotInScope x) = pretty "Not in scope:" <+> pretty x
  pretty (WCNotAllowed) = pretty "Wildcards not allowed in this context:"
  pretty (LitNotAllowed x) = pretty "Literals not allowed in this context:" <+> pretty x

instance Pretty RuntimeErr where
  pretty (MatchErr p) = pretty "Pattern match failure:" <+> pretty p
  pretty (DivByZero) = pretty "Division by zero"

instance Pretty Value where
  pretty VUnit = pretty "()"
  pretty (VInt i) = pretty i
  pretty (VChar c) = pretty c
  pretty (VCon c []) = pretty c
  pretty (VCon c args) = pretty c <+> hsep (map (wrap compoundV) args)
  pretty VClosure{} = pretty "??? can't print closure ???"

compoundV :: Value -> Bool
compoundV = \case
```

```
  VUnit -> False
  VInt _ -> False
  VChar _ -> False
  VCon _ [] -> False
  VCon _ _ -> True
  _ -> True
```