- Hindley-Milner Inference
    - Syntax
    - Polymorphism
    - Types
    - Context
    - Inference Monad
    - Substitution
    - Unification
    - Generalization and Instantiation
    - Typing Rules
    - Constraint Generation
    - Typing
    - Constraint Solver
    - Worked Examples
    - Interpreter
    - Interactive Shell
    - Observations
    - Full Source

Hindley-Milner Inference

> *There is nothing more practical than a good theory.*
>
> *— James C. Maxwell*

Hindley-Milner Inference

The Hindley-Milner type system (also referred to as Damas-Hindley-Milner or HM) is a family of type systems that admit the serendipitous property of having a tractable algorithm for determining types from untyped syntax. This is achieved by a process known as *unification*, whereby the types for a well-structured program give rise to a set of constraints that when solved always have a unique *principal type*.

The simplest Hindley Milner type system is defined by a very short set of rules. The first four rules describe the judgements by which we can map each syntactic construct (`Lam` , `App` , `Var` , `Let`) to their expected types. We'll elaborate on these rules shortly.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad (\textbf{T-Var})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \quad (\textbf{T-App})$$

$$\frac{\Gamma,\ x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda\, x\ .\ e : \tau_1 \rightarrow \tau_2} \quad (\textbf{T-Lam})$$

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma,\ x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau} \quad (\textbf{T-Let})$$

$$\frac{\Gamma \vdash e : \sigma \quad \overline{\alpha} \notin \texttt{ftv}(\Gamma)}{\Gamma \vdash e : \forall\, \overline{\alpha}\ .\ \sigma} \quad (\textbf{T-Gen})$$

$$\frac{\Gamma \vdash e : \sigma_1 \qquad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash e : \sigma_2} \quad (\textbf{T-Inst})$$

Milner's observation was that since the typing rules map uniquely onto syntax, we can in effect run the typing rules "backwards" and whenever we don't have a known type for a subexpression, we "guess" by putting a fresh variable in its place, collecting constraints about its usage induced by subsequent typing judgements. This is the essence of *type inference* in the ML family of languages, that by the generation and solving of a class of unification problems we can reconstruct the types uniquely from the syntax. The algorithm itself is largely just the structured use of a unification solver.

However full type inference leaves us in a bit a bind, since while the problem of inference is tractable within this simple language and trivial extensions thereof, but nearly any major addition to the language destroys the ability to infer types unaided by annotation or severely complicates the inference algorithm. Nevertheless the Hindley-Milner family represents a very useful, productive "sweet spot" in the design space.

Syntax

The syntax of our first type inferred language will effectively be an extension of our untyped lambda calculus, with fixpoint operator, booleans, integers, let, and a few basic arithmetic operations.

```haskell
type Name = String

data Expr
  = Var Name
  | App Expr Expr
  | Lam Name Expr
  | Let Name Expr Expr
  | Lit Lit
  | If Expr Expr Expr
  | Fix Expr
  | Op Binop Expr Expr
  deriving (Show, Eq, Ord)

data Lit
  = LInt Integer
  | LBool Bool
  deriving (Show, Eq, Ord)

data Binop = Add | Sub | Mul | Eql
  deriving (Eq, Ord, Show)

data Program = Program [Decl] Expr deriving Eq

type Decl = (String, Expr)
```

The parser is trivial, the only addition will be the toplevel let declarations (`Decl`) which are joined into the global `Program` . All toplevel declarations must be terminated with a semicolon, although they can span multiple lines and whitespace is ignored. So for instance:

```
-- SKI combinators
let I x = x;
let K x y = x;
let S f g x = f x (g x);
```

As before `let rec` expressions will expand out in terms of the fixpoint operator and are just syntactic sugar.

Polymorphism

We will add an additional constructs to our language that will admit a new form of *polymorphism* for our language. Polymorphism is the property of a term to simultaneously admit several distinct types for the same function implementation.

For instance the polymorphic signature for the identity function maps an input of type α

$$\begin{aligned} \texttt{id} &:: \forall \alpha. \alpha \to \alpha \\ \texttt{id} &= \lambda x : \alpha.\ x \end{aligned}$$

Now instead of having to duplicate the functionality for every possible type (i.e. implementing idInt, idBool, ...) we our type system admits any instantiation that is subsumed by the polymorphic type signature.

$$\begin{aligned} \texttt{id}_{\texttt{Int}} &= \texttt{Int} \to \texttt{Int} \\ \texttt{id}_{\texttt{Bool}} &= \texttt{Bool} \to \texttt{Bool} \end{aligned}$$

A rather remarkably fact of universal quantification is that many properties about inhabitants of a type are guaranteed by construction, these are the so-called *free theorems*. For instance any (nonpathological) inhabitant of the type `(a, b) -> a` must be equivalent to `fst` .

A slightly less trivial example is that of the `fmap` function of type `Functor f => (a -> b) -> f a -> f b` . The second functor law demands that:

```
forall f g. fmap f . fmap g = fmap (f . g)
```

However it is impossible to write down a (nonpathological) function for `fmap` that has the required type and doesn't have this property. We get the theorem for free!

Types

The type language we'll use starts with the simple type system we used for our typed lambda calculus.

```haskell
newtype TVar = TV String
  deriving (Show, Eq, Ord)

data Type
  = TVar TVar
  | TCon String
  | TArr Type Type
  deriving (Show, Eq, Ord)

typeInt, typeBool :: Type
typeInt  = TCon "Int"
typeBool = TCon "Bool"
```

Type schemes model polymorphic types, they indicate that the type variables bound in quantifier are polymorphic across the enclosed type and can be instantiated with any type consistent with the signature. Intuitively the indicate that the implementation of the function

```haskell
data Scheme = Forall [TVar] Type
```

Type schemes will be written as σ in our typing rules.

$$\begin{aligned} \sigma ::= \; & \tau \\ & \forall \overline{\alpha}.\, \tau \end{aligned}$$

For example the `id` and the `const` functions would have the following types:

$$\begin{aligned} \text{id} &: \forall a.\, a \rightarrow a \\ \text{const} &: \forall ab.\, a \rightarrow b \rightarrow a \end{aligned}$$

We've now divided our types into two syntactic categories, the *monotypes* and the *polytypes*. In our simple initial languages type schemes will always be the representation of top level signature, even if there are no polymorphic type variables. In implementation terms this means when a monotype is yielded from our Infer monad after inference, we will immediately generalize it at the toplevel *closing over* all free type variables in a type scheme.

Context

The typing context or environment is the central container around which all information during the inference process is stored and queried. In Haskell our implementation will simply be a newtype wrapper around a Map of `Var` to `Scheme` types.

```haskell
newtype TypeEnv = TypeEnv (Map.Map Var Scheme)
```

The two primary operations are *extension* and *restriction* which introduce or remove named quantities from the context.

$$\Gamma \backslash x = \{ y : \sigma | y : \sigma \in \Gamma, x \neq y \}$$

$$\Gamma, x : \tau = (\Gamma \backslash x) \cup \{x : \tau\}$$

Operations over the context are simply the usual Set operations on the underlying map.

```haskell
extend :: TypeEnv -> (Var, Scheme) -> TypeEnv
extend (TypeEnv env) (x, s) = TypeEnv $ Map.insert x s env
```

Inference Monad

All our logic for type inference will live inside of the `Infer` monad. It is a monad transformer stack of `ExcpetT` + `State`, allowing various error reporting and statefully holding the fresh name supply.

```haskell
type Infer a = ExceptT TypeError (State Unique) a
```

Running the logic in the monad results in either a type error or a resulting type scheme.

```haskell
runInfer :: Infer (Subst, Type) -> Either TypeError Scheme
runInfer m = case evalState (runExceptT m) initUnique of
  Left err  -> Left err
  Right res -> Right $ closeOver res
```

Substitution

Two operations that will perform quite a bit are querying the free variables of an expression and applying substitutions over expressions.

$$\begin{aligned} \mathtt{fv}(x) &= x \\ \mathtt{fv}(\lambda x. e) &= \mathtt{fv}(e) - \{x\} \\ \mathtt{fv}(e_1 e_2) &= \mathtt{fv}(e_1) \cup \mathtt{fv}(e_2) \end{aligned}$$

The same pattern applies to type variables at the type level.

$$\begin{aligned} \mathtt{ftv}(\alpha) &= \{\alpha\} \\ \mathtt{ftv}(\tau_1 \rightarrow \tau_2) &= \mathtt{ftv}(\tau_1) \cup \mathtt{ftv}(\tau_2) \\ \mathtt{ftv}(\mathtt{Int}) &= \varnothing \\ \mathtt{ftv}(\mathtt{Bool}) &= \varnothing \\ \mathtt{ftv}(\forall x. t) &= \mathtt{ftv}(t) - \{x\} \end{aligned}$$

Substitutions over expressions apply the substitution to local variables, replacing the named subexpression if matched. In the case of name capture a fresh variable is introduced.

$$\begin{aligned} [x/e']x &= e' \\ [x/e']y &= y \quad (y \neq x) \\ [x/e'](e_1 e_2) &= ([x/e']\ e_1)([x/e']e_2) \\ [x/e'](\lambda y. e_1) &= \lambda y. [x/e']e \quad y \neq x, y \notin \mathtt{fv}(e') \end{aligned}$$

And likewise, substitutions can be applied element wise over the typing environment.

$$[t/s]\Gamma = \{y : [t/s]\sigma \mid y : \sigma \in \Gamma\}$$

Our implementation of a substitution in Haskell is simply a Map from type variables to types.

```haskell
type Subst = Map.Map TVar Type
```

Composition of substitutions ($s_1 \circ s_2$, `s1 `compose` s2`) can be encoded simply as operations over the underlying map. Importantly note that in our implementation we have chosen the substitution to be left-biased, it is up to the implementation of the inference algorithm to ensure that clashes do not occur between substitutions.

```haskell
nullSubst :: Subst
nullSubst = Map.empty

compose :: Subst -> Subst -> Subst
s1 `compose` s2 = Map.map (apply s1) s2 `Map.union` s1
```

The implementation in Haskell is via a series of implementations of a `Substitutable` typeclass which exposes an `apply` function which applies the substitution given over the structure of the type replacing type variables as specified.

```haskell
class Substitutable a where
  apply :: Subst -> a -> a
  ftv   :: a -> Set.Set TVar

instance Substitutable Type where
  apply _ (TCon a)       = TCon a
  apply s t@(TVar a)     = Map.findWithDefault t a s
  apply s (t1 `TArr` t2) = apply s t1 `TArr` apply s t2

  ftv TCon{}         = Set.empty
  ftv (TVar a)       = Set.singleton a
  ftv (t1 `TArr` t2) = ftv t1 `Set.union` ftv t2

instance Substitutable Scheme where
  apply s (Forall as t)   = Forall as $ apply s' t
                            where s' = foldr Map.delete s as
  ftv (Forall as t) = ftv t `Set.difference` Set.fromList as

instance Substitutable a => Substitutable [a] where
  apply = fmap . apply
  ftv   = foldr (Set.union . ftv) Set.empty

instance Substitutable TypeEnv where
  apply s (TypeEnv env) =  TypeEnv $ Map.map (apply s) env
  ftv (TypeEnv env) = ftv $ Map.elems env
```

Throughout both the typing rules and substitutions we will require a fresh supply of names. In this naive version we will simply use an infinite list of strings and slice into n'th element of list per an index that we hold in a State monad. This is a simplest implementation possible, and later we will adapt this name generation technique to be more robust.

```haskell
letters :: [String]
letters = [1..] >>= flip replicateM ['a'..'z']

fresh :: Infer Type
fresh = do
  s <- get
  put s{count = count s + 1}
  return $ TVar $ TV (letters !! count s)
```

The creation of fresh variables will be essential for implementing the inference rules. Whenever we encounter the first use of a variable within some expression we will create a fresh type variable.

Unification

Central to the idea of inference is the notion of *unification*. A unifier for two expressions e_1 and e_2 is a substitution s such that:

$$s := [n_0/m_0, n_1/m_1, \dots, n_k/m_k]$$
$$[s]e_1 = [s]e_2$$

Two terms are said to be *unifiable* if there exists a unifying substitution set between them. A substitution set is said to be *confluent* if the application of substitutions is independent of the order applied, i.e. if we always arrive at the same normal form regardless of the order of substitution chosen.

We'll adopt the notation

$$\tau \sim \tau' : s$$

for the fact that two types τ, τ' are unifiable by a substitution s, such that:

$$[s]\tau = [s]\tau'$$

Two identical terms are trivially unifiable by the empty unifier.

$$c \sim c : [\,]$$

The unification rules for our little HM language are as follows:

$$c \sim c : [] \qquad (\textbf{Uni-Const})$$

$$\alpha \sim \alpha : [] \qquad (\textbf{Uni-Var})$$

$$\frac{\alpha \notin \texttt{ftv}(\tau)}{\alpha \sim \tau : [\alpha/\tau]} \qquad (\textbf{Uni-VarLeft})$$

$$\frac{\alpha \notin \texttt{ftv}(\tau)}{\tau \sim \alpha : [\alpha/\tau]} \qquad (\textbf{Uni-VarRight})$$

$$\frac{\tau_1 \sim \tau_1' : \theta_1 \qquad [\theta_1]\tau_2 \sim [\theta_1]\tau_2' : \theta_2}{\tau_1\tau_2 \sim \tau_1'\tau_2' : \theta_2 \circ \theta_1} \qquad (\textbf{Uni-Con})$$

$$\frac{\tau_1 \sim \tau_1' : \theta_1 \qquad [\theta_1]\tau_2 \sim [\theta_1]\tau_2' : \theta_2}{\tau_1 \rightarrow \tau_2 \sim \tau_1' \rightarrow \tau_2' : \theta_2 \circ \theta_1} \qquad (\textbf{Uni-Arrow})$$

If we want to unify a type variable α with a type τ, we usually can just substitute the variable with the type: $[\alpha/\tau]$. However, our rules state a precondition known as the *occurs check* for that unification: the type variable α must not occur free in τ. If it did, the substitution would not be a unifier.

Take for example the problem of unifying α and $\alpha \rightarrow \beta$. The substitution $s = [\alpha/\alpha \rightarrow \beta]$ doesn't unify: we get

$$[s]\alpha = \alpha \rightarrow \beta$$

and

$$[s]\alpha \rightarrow \beta = (\alpha \rightarrow \beta) \rightarrow \beta.$$

Indeed, whatever substitution s we try, $[s]\alpha \rightarrow \beta$ will always be longer than $[s]\alpha$, so no unifier exists. The only chance would be to substitute with an infinite type: $[\alpha/(\dots((\alpha \rightarrow \beta) \rightarrow \beta) \rightarrow \dots \rightarrow \beta) \rightarrow \beta]$ would be a unifier, but our language has no such types.

If the unification fails because of the occurs check, we say that unification would give an infinite type.

Note that unifying $\alpha \rightarrow \beta$ and α is exactly what we would have to do if we tried to type check the omega combinator $\lambda x.xx$, so it is ruled out by the occurs check, as are other pathological terms we discussed when covering the untyped lambda calculus.

```
occursCheck ::  Substitutable a => TVar -> a -> Bool
occursCheck a t = a `Set.member` ftv t
```

The unify function lives in the Infer monad and yields a subsitution:

```haskell
unify ::  Type -> Type -> Infer Subst
unify (l `TArr` r) (l' `TArr` r')  = do
    s1 <- unify l l'
    s2 <- unify (apply s1 r) (apply s1 r')
    return (s2 `compose` s1)

unify (TVar a) t = bind a t
unify t (TVar a) = bind a t
unify (TCon a) (TCon b) | a == b = return nullSubst
unify t1 t2 = throwError $ UnificationFail t1 t2

bind ::  TVar -> Type -> Infer Subst
bind a t | t == TVar a     = return nullSubst
         | occursCheck a t = throwError $ InfiniteType a t
         | otherwise       = return $ Map.singleton a t
```

Generalization and Instantiation

At the heart of Hindley-Milner is two fundamental operations:

- **Generalization**: Converting a τ type into a σ type by closing over all free type variables in a type scheme.
- **Instantiation**: Converting a σ type into a τ type by creating fresh names for each type variable that does not appear in the current typing environment.

$$\frac{\Gamma \vdash e : \sigma \quad \overline{\alpha} \notin \mathtt{ftv}(\Gamma)}{\Gamma \vdash e : \forall \, \overline{\alpha} \, . \, \sigma} \quad (\textbf{T-Gen})$$

$$\frac{\Gamma \vdash e : \sigma_1 \qquad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash e : \sigma_2} \quad (\textbf{T-Inst})$$

The $\sqsubseteq$ operator in the $(\textbf{T-Inst})$ rule indicates that a type is an *instantiation* of a type scheme.

$$\forall \overline{\alpha}. \tau_2 \sqsubseteq \tau_1$$

A type τ_1 is a instantiation of a type scheme $\sigma = \forall \overline{\alpha}. \tau_2$ if there exists a substitution $[s]\beta = \beta$ for all $\beta \in \mathtt{ftv}(\sigma)$ so that $\tau_1 = [s]\tau_2$. Some examples:

$$\begin{aligned} \forall a. a \to a &\sqsubseteq \mathtt{Int} \to \mathtt{Int} \\ \forall a. a \to a &\sqsubseteq b \to b \\ \forall ab. a \to b \to a &\sqsubseteq \mathtt{Int} \to \mathtt{Bool} \to \mathtt{Int} \end{aligned}$$

These map very intuitively into code that simply manipulates the Haskell `Set` objects of variables and the fresh name supply:

```haskell
instantiate ::  Scheme -> Infer Type
instantiate (Forall as t) = do
```

```
  as' <- mapM (const fresh) as
  let s = Map.fromList $ zip as as'
  return $ apply s t

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

By convention let-bindings are generalized as much as possible. So for instance in the following definition `f` is generalized across the body of the binding so that at each invocation of `f` it is instantiated with fresh type variables.

```
Poly> let f = (\x -> x) in let g = (f True) in f 3
3 : Int
```

In this expression, the type of `f` is generated at the let definition and will be instantiated with two different signatures. At call site of `f` it will unify with `Int` and the other unify with `Bool`.

By contrast, binding `f` in a lambda will result in a type error.

```
Poly> (\f -> let g = (f True) in (f 3)) (\x -> x)
Cannot unify types:
    Bool
with
    Int
```

This is the essence of *let generalization*.

Typing Rules

And finally with all the typing machinery in place, we can write down the typing rules for our simple little polymorphic lambda calculus.

```
infer :: TypeEnv -> Expr -> Infer (Subst, Type)
```

The `infer` maps the local typing environment and the active expression to a 2-tuple of the partial unifier solution and the intermediate type. The AST is traversed bottom-up and constraints are solved at each level of recursion by applying partial substitutions from unification across each partially inferred subexpression and the local environment. If an error is encountered the `throwError` is called in the `Infer` monad and an error is reported.

```
infer :: TypeEnv -> Expr -> Infer (Subst, Type)
infer env ex = case ex of

  Var x -> lookupEnv env x

  Lam x e -> do
    tv <- fresh
    let env' = env `extend` (x, Forall [] tv)
    (s1, t1) <- infer env' e
    return (s1, apply s1 tv `TArr` t1)

  App e1 e2 -> do
```

```haskell
    tv <- fresh
    (s1, t1) <- infer env e1
    (s2, t2) <- infer (apply s1 env) e2
    s3       <- unify (apply s2 t1) (TArr t2 tv)
    return (s3 `compose` s2 `compose` s1, apply s3 tv)

  Let x e1 e2 -> do
    (s1, t1) <- infer env e1
    let env' = apply s1 env
        t'   = generalize env' t1
    (s2, t2) <- infer (env' `extend` (x, t')) e2
    return (s1 `compose` s2, t2)

  If cond tr fl -> do
    (s1, t1) <- infer env cond
    (s2, t2) <- infer env tr
    (s3, t3) <- infer env fl
    s4 <- unify t1 typeBool
    s5 <- unify t2 t3
    return (s5 `compose` s4 `compose` s3 `compose` s2 `compose` s1, apply s5 t2)

  Fix e1 -> do
    (s1, t) <- infer env e1
    tv <- fresh
    s2 <- unify (TArr tv tv) t
    return (s2, apply s1 tv)

  Op op e1 e2 -> do
    (s1, t1) <- infer env e1
    (s2, t2) <- infer env e2
    tv <- fresh
    s3 <- unify (TArr t1 (TArr t2 tv)) (ops Map.! op)
    return (s1 `compose` s2 `compose` s3, apply s3 tv)

  Lit (LInt _)  -> return (nullSubst, typeInt)
  Lit (LBool _) -> return (nullSubst, typeBool)
```

Let's walk through each of the rule derivations and look how it translates into code:

T-Var

The `T-Var` rule, simply pull the type of the variable out of the typing context.

```haskell
  Var x -> lookupEnv env x
```

The function `lookupVar` looks up the local variable reference in typing environment and if found it instantiates a fresh copy.

```haskell
lookupEnv :: TypeEnv -> Var -> Infer (Subst, Type)
lookupEnv (TypeEnv env) x = do
  case Map.lookup x env of
    Nothing -> throwError $ UnboundVariable (show x)
    Just s  -> do t <- instantiate s
                  return (nullSubst, t)
```

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad (\textbf{T-Var})$$

T-Lam

For lambdas the variable bound by the lambda is locally scoped to the typing environment and then the body of the expression is inferred with this scope. The output type is a fresh type variable and is unified with the resulting inferred type.

```
Lam x e -> do
  tv <- fresh
  let env' = env `extend` (x, Forall [] tv)
  (s1, t1) <- infer env' e
  return (s1, apply s1 tv `TArr` t1)
```

$$\frac{\Gamma,\ x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda\, x\ .\ e : \tau_1 \rightarrow \tau_2} \quad (\textbf{T-Lam})$$

T-App

For applications, the first argument must be a lambda expression or return a lambda expression, so know it must be of form `t1 -> t2` but the output type is not determined except by the confluence of the two values. We infer both types, apply the constraints from the first argument over the result second inferred type and then unify the two types with the excepted form of the entire application expression.

```
App e1 e2 -> do
  tv <- fresh
  (s1, t1) <- infer env e1
  (s2, t2) <- infer (apply s1 env) e2
  s3       <- unify (apply s2 t1) (TArr t2 tv)
  return (s3 `compose` s2 `compose` s1, apply s3 tv)
```

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \quad (\textbf{T-App})$$

T-Let

As mentioned previously, let will be generalized so we will create a local typing environment for the body of the let expression and add the generalized inferred type let bound value to the typing environment of the body.

```
Let x e1 e2 -> do
  (s1, t1) <- infer env e1
  let env' = apply s1 env
      t'   = generalize env' t1
```

```
    (s2, t2) <- infer (env' `extend` (x, t')) e2
    return (s1 `compose` s2, t2)
```

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau} \quad (\textbf{T-Let})$$

T-BinOp

There are several builtin operations, we haven't mentioned up to now because the typing rules are trivial. We simply unify with the preset type signature of the operation.

```
 Op op e1 e2 -> do
    (s1, t1) <- infer env e1
    (s2, t2) <- infer env e2
    tv <- fresh
    s3 <- unify (TArr t1 (TArr t2 tv)) (ops Map.! op)
    return (s1 `compose` s2 `compose` s3, apply s3 tv)
```

```
ops :: Map.Map Binop Type
ops = Map.fromList [
      (Add, (typeInt `TArr` (typeInt `TArr` typeInt)))
    , (Mul, (typeInt `TArr` (typeInt `TArr` typeInt)))
    , (Sub, (typeInt `TArr` (typeInt `TArr` typeInt)))
    , (Eql, (typeInt `TArr` (typeInt `TArr` typeBool)))
  ]
```

$$\begin{aligned}
(+) &\quad : \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int} \\
(\times) &\quad : \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int} \\
(-) &\quad : \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int} \\
(=) &\quad : \mathtt{Int} \to \mathtt{Int} \to \mathtt{Bool}
\end{aligned}$$

Literals

The type of literal integer and boolean types is trivially their respective types.

$$\frac{}{\Gamma \vdash n : \mathtt{Int}} \quad (\textbf{T-Int})$$

$$\frac{}{\Gamma \vdash \mathtt{True} : \mathtt{Bool}} \quad (\textbf{T-True})$$

$$\frac{}{\Gamma \vdash \mathtt{False} : \mathtt{Bool}} \quad (\textbf{T-False})$$

Constraint Generation

The previous implementation of Hindley Milner is simple, but has this odd property of intermingling two separate processes: constraint solving and traversal. Let's discuss another implementation of the inference algorithm that does not do this.

In the *constraint generation* approach, constraints are generated by bottom-up traversal, added to a ordered container, canonicalized, solved, and then possibly back-substituted over a typed AST. This will be the approach we will use from here out, and while there is an equivalence between the "on-line solver", using the separate constraint solver becomes easier to manage as our type system gets more complex and we start building out the language.

Our inference monad now becomes a `RWST` (Reader-Writer-State Transformer) + `Except` for typing errors. The inference state remains the same, just the fresh name supply.

```haskell
-- | Inference monad
type Infer a = (RWST
                  Env             -- Typing environment
                  [Constraint]    -- Generated constraints
                  InferState      -- Inference state
                  (Except         -- Inference errors
                    TypeError)
                  a)              -- Result

-- | Inference state
data InferState = InferState { count :: Int }
```

Instead of unifying type variables at each level of traversal, we will instead just collect the unifiers inside the Writer and emit them with the `uni` function.

```haskell
-- | Unify two types
uni :: Type -> Type -> Infer ()
uni t1 t2 = tell [(t1, t2)]
```

Since the typing environment is stored in the Reader monad, we can use the `local` to create a locally scoped additions to the typing environment. This is convenient for typing binders.

```haskell
-- | Extend type environment
inEnv :: (Name, Scheme) -> Infer a -> Infer a
inEnv (x, sc) m = do
  let scope e = (remove e x) `extend` (x, sc)
  local scope m
```

Typing

The typing rules are identical, except they now can be written down in a much less noisy way that isn't threading so much state. All of the details are taken care of under the hood and encoded in specific combinators manipulating the state of our Infer monad in a way that lets focus on the domain logic.

```haskell
infer :: Expr -> Infer Type
infer expr = case expr of
  Lit (LInt _)  -> return $ typeInt
  Lit (LBool _) -> return $ typeBool

  Var x -> lookupEnv x
  Lam x e -> do
    tv <- fresh
    t <- inEnv (x, Forall [] tv) (infer e)
    return (tv `TArr` t)

  App e1 e2 -> do
    t1 <- infer e1
    t2 <- infer e2
    tv <- fresh
    uni t1 (t2 `TArr` tv)
    return tv

  Let x e1 e2 -> do
    env <- ask
    t1 <- infer e1
    let sc = generalize env t1
    t2 <- inEnv (x, sc) (infer e2)
    return t2

  Fix e1 -> do
    t1 <- infer e1
    tv <- fresh
    uni (tv `TArr` tv) t1
    return tv

  Op op e1 e2 -> do
    t1 <- infer e1
    t2 <- infer e2
    tv <- fresh
    let u1 = t1 `TArr` (t2 `TArr` tv)
        u2 = ops Map.! op
    uni u1 u2
    return tv

  If cond tr fl -> do
    t1 <- infer cond
    t2 <- infer tr
    t3 <- infer fl
    uni t1 typeBool
    uni t2 t3
    return t2
```

Constraint Solver

The Writer layer for the Infer monad contains the generated set of constraints emitted from inference pass. Once inference has completed we are left with a resulting type signature full of meaningless unique fresh variables and a set of constraints that we must solve to refine the type down to its principal type.

The constraints are pulled out solved by a separate `Solve` monad which holds the Unifier (most general unifier) solution that when applied to generated signature will yield the solution.

```
type Constraint = (Type, Type)

type Unifier = (Subst, [Constraint])
```

```
-- | Constraint solver monad
type Solve a = StateT Unifier (ExceptT TypeError Identity) a
```

The unification logic is also identical to before, except it is now written independent of inference and stores its partial state inside of the Solve monad's state layer.

```
unifies :: Type -> Type -> Solve Unifier
unifies t1 t2 | t1 == t2 = return emptyUnifer
unifies (TVar v) t = v `bind` t
unifies t (TVar v) = v `bind` t
unifies (TArr t1 t2) (TArr t3 t4) = unifyMany [t1, t2] [t3, t4]
unifies t1 t2 = throwError $ UnificationFail t1 t2

unifyMany :: [Type] -> [Type] -> Solve Unifier
unifyMany [] [] = return emptyUnifer
unifyMany (t1 : ts1) (t2 : ts2) =
  do (su1,cs1) <- unifies t1 t2
     (su2,cs2) <- unifyMany (apply su1 ts1) (apply su1 ts2)
     return (su2 `compose` su1, cs1 ++ cs2)
unifyMany t1 t2 = throwError $ UnificationMismatch t1 t2
```

The solver function simply iterates over the set of constraints, composing them and applying the resulting constraint solution over the intermediate solution eventually converting on the *most general unifier* which yields the final subsitution which when applied over the inferred type signature, yields the principal type solution for the expression.

```
-- Unification solver
solver :: Solve Subst
solver = do
  (su, cs) <- get
  case cs of
    [] -> return su
    ((t1, t2): cs0) -> do
      (su1, cs1)  <- unifies t1 t2
      put (su1 `compose` su, cs1 ++ (apply su1 cs0))
      solver
```

This a much more elegant solution than having to intermingle inference and solving in the same pass, and adapts itself well to the generation of a typed Core form which we will discuss in later chapters.

Worked Examples

Let's walk through two examples of how inference works for simple functions.

Example 1

Consider:

```
\x y z -> x + y + z
```

The generated type from the `infer` function consists simply of a fresh variable for each of the arguments and the return type.

```
a -> b -> c -> e
```

The constraints induced from **T-BinOp** are emitted as we traverse both of the addition operations.

1. `a -> b -> d ~ Int -> Int -> Int`
2. `d -> c -> e ~ Int -> Int -> Int`

Here `d` is the type of the intermediate term `x + y`. By applying **Uni-Arrow** we can then deduce the following set of substitutions.

1. `a ~ Int`
2. `b ~ Int`
3. `c ~ Int`
4. `d ~ Int`
5. `e ~ Int`

Substituting this solution back over the type yields the inferred type:

```
Int -> Int -> Int -> Int
```

Example 2

```
compose f g x = f (g x)
```

The generated type from the `infer` function consists again simply of unique fresh variables.

```
a -> b -> c -> e
```

Induced by two cases of the **T-App** rule we get the following constraints:

1. `b ~ c -> d`
2. `a ~ d -> e`

Here `d` is the type of `(g x)`. The constraints are already in a canonical form, by applying **Uni-VarLeft** twice we get the following set of substitutions:

1. `b ~ c -> d`
2. `a ~ d -> e`

So we get this type:

```
compose :: forall c d e. (d -> e) -> (c -> d) -> c -> e
```

If desired, we can rename the variables in alphabetical order to get:

```
compose :: forall a b c. (a -> b) -> (c -> a) -> c -> b
```

Interpreter

Our evaluator will operate directly on the syntax and evaluate the results in into a `Value` type.

```
data Value
  = VInt Integer
  | VBool Bool
  | VClosure String Expr TermEnv
```

The interpreter is set up an Identity monad. Later it will become a more complicated monad, but for now its quite simple. The value environment will explicitly threaded around, and whenever a closure is created we simply store a copy of the local environment in the closure.

```
type TermEnv = Map.Map String Value
type Interpreter t = Identity t
```

Our logic for evaluation is an extension of the lambda calculus evaluator implemented in previous chapter. However you might notice quite a few incomplete patterns used throughout evaluation. Fear not though, the evaluation of our program cannot "go wrong". Each of these patterns represents a state that our type system guarantees will never happen. For example, if our program did have not every variable referenced in scope then it would never reach evaluation to begin with and would be rejected in our type checker. We are morally correct in using incomplete patterns here!

```
eval :: TermEnv -> Expr -> Interpreter Value
eval env expr = case expr of
  Lit (LInt k)  -> return $ VInt k
  Lit (LBool k) -> return $ VBool k

  Var x -> do
    let Just v = Map.lookup x env
    return v

  Op op a b -> do
    VInt a' <- eval env a
    VInt b' <- eval env b
    return $ (binop op) a' b'

  Lam x body ->
    return (VClosure x body env)

  App fun arg -> do
    VClosure x body clo <- eval env fun
    argv <- eval env arg
    let nenv = Map.insert x argv clo
    eval nenv body

  Let x e body -> do
    e' <- eval env e
    let nenv = Map.insert x e' env
    eval nenv body

  If cond tr fl -> do
    VBool br <- eval env cond
    if br == True
    then eval env tr
    else eval env fl

  Fix e -> do
    eval env (App e (Fix e))
```

```haskell
binop :: Binop -> Integer -> Integer -> Value
binop Add a b = VInt $ a + b
binop Mul a b = VInt $ a * b
binop Sub a b = VInt $ a - b
binop Eql a b = VBool $ a == b
```

Interactive Shell

Our language has now grown out the small little shells we were using before, and now we need something much more robust to hold the logic for our interactive interpreter.

We will structure our REPL as a monad wrapped around IState (the interpreter state) datatype. We will start to use the repline library from here out which gives us platform independent readline, history, and tab completion support.

```haskell
data IState = IState
  { tyctx :: TypeEnv  -- Type environment
  , tmctx :: TermEnv  -- Value environment
  }

initState :: IState
initState = IState emptyTyenv emptyTmenv

type Repl a = HaskelineT (StateT IState IO) a

hoistErr :: Show e => Either e a -> Repl a
hoistErr (Right val) = return val
hoistErr (Left err) = do
  liftIO $ print err
  abort
```

Our language can be compiled into a standalone binary by GHC:

```
$ ghc --make Main.hs -o poly
$ ./poly
Poly>
```

At the top of our program we will look at the command options and allow three variations of commands.

```
$ poly                  # launch shell
$ poly input.ml         # launch shell with 'input.ml' loaded
$ poly test input.ml    # dump test for 'input.ml' to stdout
```

```haskell
main :: IO ()
main = do
  args <- getArgs
  case args of
    []      -> shell (return ())
    [fname] -> shell (load [fname])
    ["test", fname] -> shell (load [fname] >> browse [] >> quit ())
    _ -> putStrLn "invalid arguments"
```

The shell command takes a `pre` action which is run before the shell starts up. The logic simply evaluates our Repl monad into an IO and runs that from the main function.

```
shell :: Repl a -> IO ()
shell pre
  = flip evalStateT initState
  $ evalRepl "Poly> " cmd options completer pre
```

The `cmd` driver is the main entry point for our program, it is executed every time the user enters a line of input. The first argument is the line of user input.

```
cmd :: String -> Repl ()
cmd source = exec True (L.pack source)
```

The heart of our language is then the `exec` function which imports all the compiler passes, runs them sequentially threading the inputs and outputs and eventually yielding a resulting typing environment and the evaluated result of the program. These are monoidally joined into the state of the interpreter and then the loop yields to the next set of inputs.

```
exec :: Bool -> L.Text -> Repl ()
exec update source = do
  -- Get the current interpreter state
  st <- get

  -- Parser ( returns AST )
  mod <- hoistErr $ parseModule "<stdin>" source

  -- Type Inference ( returns Typing Environment )
  tyctx' <- hoistErr $ inferTop (tyctx st) mod

  -- Create the new environment
  let st' = st { tmctx = foldl' evalDef (tmctx st) mod
               , tyctx = tyctx' <> (tyctx st)
               }

  -- Update the interpreter state
  when update (put st')
```

Repline also supports adding special casing certain sets of inputs so that they map to builtin commands in the compiler. We will implement three of these.

Command	Action
`:browse`	Browse the type signatures for a program
`:load <file>`	Load a program from file
`:type`	Show the type of an expression
`:quit`	Exit interpreter

Their implementations are mostly straightforward.

```
options :: [(String, [String] -> Repl ())]
options = [
```

```haskell
    ("load"   , load)
  , ("browse" , browse)
  , ("quit"   , quit)
  , ("type"   , Main.typeof)
  ]

-- :browse command
browse :: [String] -> Repl ()
browse _ = do
  st <- get
  liftIO $ mapM_ putStrLn $ ppenv (tyctx st)

-- :load command
load :: [String] -> Repl ()
load args = do
  contents <- liftIO $ L.readFile (unwords args)
  exec True contents

-- :type command
typeof :: [String] -> Repl ()
typeof args = do
  st <- get
  let arg = unwords args
  case Infer.typeof (tyctx st) arg of
    Just val -> liftIO $ putStrLn $ ppsignature (arg, val)
    Nothing -> exec False (L.pack arg)

-- :quit command
quit :: a -> Repl ()
quit _ = liftIO $ exitSuccess
```

Finally tab completion for our shell will use the interpreter's typing environment keys to complete on the set of locally defined variables. Repline supports prefix based tab completion where the prefix of the current command will be used to determine what to tab complete. In the case where we start with the command `:load` we will instead tab complete on filenames in the current working directly instead.

```haskell
completer :: CompleterStyle (StateT IState IO)
completer = Prefix (wordCompleter comp) defaultMatcher

-- Prefix tab completer
defaultMatcher :: MonadIO m => [(String, CompletionFunc m)]
defaultMatcher = [
    (":load"  , fileCompleter)
  ]

-- Default tab completer
comp :: (Monad m, MonadState IState m) => WordCompleter m
comp n = do
  let cmds = [":load", ":browse", ":quit", ":type"]
  TypeEnv ctx <- gets tyctx
  let defs = Map.keys ctx
  return $ filter (isPrefixOf n) (cmds ++ defs)
```

Observations

There we have it, our first little type inferred language! Load the `poly` interpreter by running `ghci Main.hs` and the call the `main` function.

```
$ ghci Main.hs
λ: main
Poly> :load test.ml
Poly> :browse
```

Try out some simple examples by declaring some functions at the toplevel of the program. We can query the types of expressions interactively using the `:type` command which effectively just runs the expression halfway through the pipeline and halts after typechecking.

```
Poly> let id x = x
Poly> let const x y = x
Poly> let twice x = x + x

Poly> :type id
id : forall a. a -> a

Poly> :type const
const : forall a b. a -> b -> a

Poly> :type twice
twice : Int -> Int
```

Notice several important facts. Our type checker will now flat our reject programs with scoping errors before interpretation.

```
Poly> \x -> y
Not in scope: "y"
```

Also programs that are also not well-typed are now rejected outright as well.

```
Poly> 1 + True
Cannot unify types:
    Bool
with
    Int
```

The omega combinator will not pass the occurs check.

```
Poly> \x -> x x
Cannot construct the the infinite type: a = a -> b
```

The file `test.ml` provides a variety of tests of the little interpreter. For instance both `fact` and `fib` functions uses the fixpoint to compute Fibonacci numbers or factorials.

```
let fact = fix (\fact -> \n ->
  if (n == 0)
    then 1
    else (n * (fact (n-1))));
```

```
let rec fib n =
  if (n == 0)
    then 0
    else if (n==1)
      then 1
      else ((fib (n-1)) + (fib (n-2)));
```

```
Poly> :type fact
fact : Int -> Int

Poly> fact 5
120

Poly> fib 16
610
```

Full Source

- Poly
- Poly - Constraint Generation