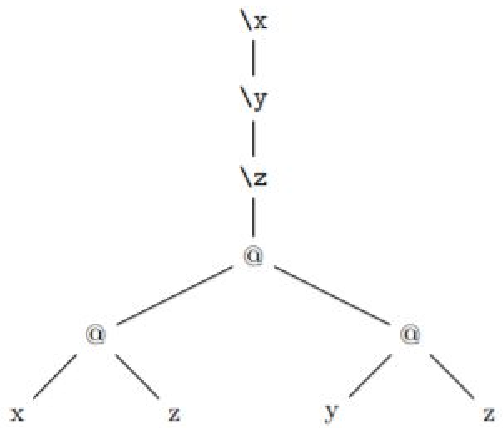
Implementing Type Inference

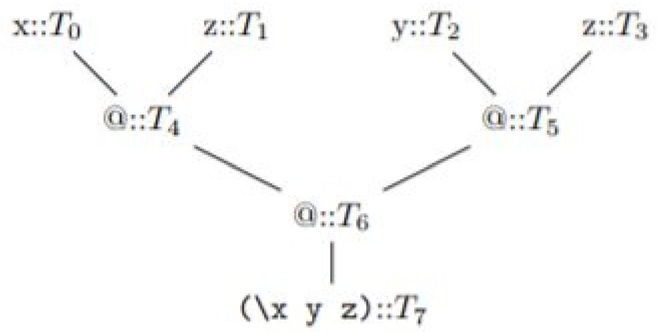
- Haskell is strongly typed - this allows incorrectly typed programs to be rejected at compile time
    - **Polymorphically typed** so types may be universally quantified overs sets of type variables
    - The compiler is capable of **automatic type inference** to discover function types
- For completeness we note:
    - **Type classes**
        - Functions can take sets of types to offer a kind of structured overloading
- **The type checker must be able to**
    - **Determine whether the program is well typed**
    - **If so, determine the type of any expression in the program**
- Programmer presumably thinks their program is well typed
    - They have an idea for the type of **each expression** in the program and could have explicitly provided them
    - In a sense the type checker recovers the lost labels on the parts of the program
- **A typical inference system** - generally regarded as 2 separate, but related, things
    - A set of type or **inference rules** which give the logical framework of the system - usually used to demonstrate the system validity
    - An **inference algorithm** which is used to deduce the types of expressions - determining a usable algorithm is usually non-trivial

An example

- Expression: \ x y z -> x z (y z)
- In tree form…
-



- Now we can add labels to the types we want to discover - **Type Variables**
- Inverting the tree will make this easier later on...



- For further convenience, redraw the tree to a type variable focused notation

$$\cfrac{\cfrac{x :: T_0 \qquad z :: T_1}{x\ z\ ::\ T_4}\,@ \qquad \cfrac{y :: T_2 \qquad z :: T_3}{y\ z\ ::\ T_5}\,@}{\cfrac{(x\ z)\ (y\ z)\ ::\ T_6}{(\lambda\ x\ y\ z)\ ::\ T_7}\,\lambda}\,@$$

  - **For any sub-part of the tree shaped like this:**

$$\frac{E_0 :: A \qquad E_1 :: B}{E_0\ E_1 :: C}\,@$$

  - **We know that this is function application**
  - **And E0 is the function, so its type will contain an arrow (some type -> some type)**

$$\frac{E_0 :: t_0 \rightarrow t_1 \qquad E_1 :: t_0}{E_0\ E_1 :: t_1}\ \mathrm{APP}$$

  - **The RHS label of the rule lets us name it**
- Application of the APP rule to the tree gives us some type equations

$$\begin{aligned} T_0 &= T_1 \rightarrow T_4 \\ T_2 &= T_3 \rightarrow T_5 \\ T_4 &= T_5 \rightarrow T_6 \end{aligned}$$

- **So we can sub these back into the tree**

$$\cfrac{\cfrac{x :: T_1 \rightarrow T_4 \qquad z :: T_1}{x\ z\ ::\ T_5 \rightarrow T_6}\,@ \qquad \cfrac{y :: T_3 \rightarrow T_5 \qquad z :: T_3}{y\ z\ ::\ T_5}\,@}{\cfrac{(x\ z)\ (y\ z)\ ::\ T_6}{(\lambda\ x\ y\ z)\ ::\ T_7}\,\lambda}\,@$$

-

- So what happens with z? It has two type variables T1 and T3 associated with its 2 instances.
- **Let's assume that they're equal, in which case we can add these equations:**

$$\begin{aligned} T_1 &= & T_3 \\ T_7 &= & T_0 \to T_2 \to T_1 \to T_6 \end{aligned}$$

-
- The T7 equation comes from our knowledge of the lambda abstraction.
- We can make that a rule also:

$$\frac{x :: A \qquad E :: B}{\lambda\, x.E \;::\; A \to B}\ \text{ABS}$$

-
  - **Note that the ABS rule is forcing us to conclude that all occurances of a lambda bound variable have the same type**
  - This is a requirement if we are to be sure that the lambda abstraction will be usable in all contexts
- **Substitute that back in and we get…**

$$\cfrac{\cfrac{\cfrac{x :: T_1 \to T_4 \qquad z :: T_1}{x\ z\ :: \ T_5 \to T_6}\,@ \qquad \cfrac{y :: T_3 \to T_5 \qquad z :: T_3}{y\ z\ ::\ T_5}\,@}{(x\ z)\ (y\ z)\ ::\ T_6}\,@}{(\lambda\ x\ y\ z)\ ::\ (T_1 \to T_5 \to T_6) \to (T_1 \to T_5) \to T_1 \to T_6}\ \lambda\ x\ y\ z$$

-
  - (T1->T5->T6) is the type of x - we have deduced this, as it is a function that takes z to produce a function that takes a T5 to produce a function that produces a T6
    - Whew
    - z in the top right function should be of type T1
  - (T1->T5) is the type of y - we have deduced this as y takes a T1 (which is z, same first arg as x), and produces a T5
  - T1 is the type of z
  - The entire lambda expression produces a value of type T6

Type environments

- **We often know some facts about the names that are used in expressions.. E.g...**

$$\cfrac{\cfrac{map :: T_0 \qquad f :: T_1}{map\ f\ ::\ T_3}\,@ \qquad \cfrac{}{[1,2,3]\ ::\ T_4}}{map\ f\ [1,2,3]\ ::\ T_5}\,@$$

We actually know that (for example):

$$\begin{aligned} T_4 &= & [Int] \\ T_0 &= & (T_7 \to T_8) \to [T_7] \to [T_8] \end{aligned}$$

-

- We represent **the set of "known facts" about variables by a mapping, Γ, from names to types, which we call the "type environment"**
- All the globally defined names in the program have entries:

$$\begin{aligned} 1 ::& \quad Int \\ [1,2,3] ::& \quad Int \\ \text{map} ::& \quad (T_0 \to T_1) \to [T_0] \to [T_1] \end{aligned}$$

-
- As we infer facts about variables and functions, we can add them to the environment
- **Type Rules**
- The standard type rules refer to this environment
- Some key rules:

$$\frac{}{\Gamma \cup \{x : t\} \vdash x : t}\ \text{Var}$$

$$\frac{\Gamma e : t' \to t \qquad \Gamma e' : t'}{\Gamma e\ e' : t}\ \text{App}$$

$$\frac{\Gamma \cup \{x : t'\} \vdash e : t}{\Gamma \vdash \lambda x.e : t' \to t}\ \text{Abs}$$

$$\frac{\Gamma \vdash_p e : \sigma \qquad \Gamma \cup \{x : \sigma\} \vdash_p e' : \tau}{\Gamma \vdash_p \textbf{let } x = e \textbf{ in } e' : \tau}\ \text{Let}$$

-
- **Worked Example - consider this small example language**

$E ::= c$	constant
$\|x$	variable
$\|\lambda\ x.E$	Abstraction
$\|(E1\ E2)$	Application
$\|let\ x\ =\ E_1\ in\ E_2$	let block

-
- **With a little language for types…**

$\tau ::= l$	base types
$\|t$	type variable
$\|\tau_0 \to \tau_1$	Function types

-

- **And type schemes…**

$$\sigma ::= \tau$$
$$|\forall t.\sigma ?$$

-
- **And type environments…**

$$T\ E ::= Identifiers \rightarrow \sigma$$

-
- **A complete implementation of the type inference engine for this language is to follow**
- Before looking at the code, look at the inference engine overview
    - **Four main components**
        - **Types defining the AST of the expression and type languages**
        - **Definitions that support the type environment**
            - Including definitions of operations on types and environments
        - **The implementation of the type inference rules**
        - **The implementation of the unification algorithm** which resolves type constraints
- **First - the AST for the language**

```haskell
data Exp = Var String
         | Const Val
         | App Exp Exp
         | Abs String Exp
         | Let String Exp Exp
     deriving (Eq,Ord)

data Val = I Integer | B Bool
     deriving (Eq,Ord)
```

-
- A small type language

```haskell
data Type = TVar String
          | TInt | TBool
          | TFun Type Type
     deriving (Eq,Ord)
```

We also have parameterised type *schemes*

```haskell
data Scheme = Scheme [String] Type
```

-

- We need to create a type environment

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

remove                    ::  TypeEnv -> String -> TypeEnv
remove (TypeEnv env) var   =  TypeEnv (Map.delete var env)
```

-
    - Nothing more than a mapping of Strings (variable names) to Schemes (type schemes)
- **We need the notion of type substitutions** - mappings from type variables to types

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

For example:

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

We need a function to find *free type variables* in a type expression, and a function to apply *type substutions.*

These can be applied to our types and to type schemes, so we make a class:

```haskell
class Types a where
  ftv   :: a -> Set.Set String
  apply :: Subst -> a -> a
```

-
    - **ftv = free type variables**
    - **Apply applies the substitutions**
- **The apply function can be used when composing substitutions**

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

instance Types Type where
    ftv (TVar n)      =  Set.singleton n
    ftv TInt          =  Set.empty
    ftv TBool         =  Set.empty
    ftv (TFun t1 t2)  =  ftv t1 `Set.union` ftv t2

    apply s (TVar n)      =  case Map.lookup n s of
                               Nothing  -> TVar n
                               Just t   -> t
    apply s (TFun t1 t2)  = TFun (apply s t1) (apply s t2)
    apply s t             = t

instance Types Scheme where
    ftv (Scheme vars t)
        =  (ftv t) `Set.difference` (Set.fromList vars)

    apply s (Scheme vars t)
        =  Scheme vars (apply (foldr Map.delete s vars) t)
```

For convenience we will extend these operations over lists as well:

```haskell
instance Types a => Types [a] where
    apply s  =  map (apply s)
    ftv l    =  foldr Set.union Set.empty (map ftv l)

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

-

```haskell
generalize        ::  TypeEnv -> Type -> Scheme
generalize env t  =   Scheme vars t
  where vars = Set.toList ((ftv t) `Set.difference` (ftv env))
```

- We will need a way to supply fresh names whenever we invent new type variables during reconstruction
- Easiest way to do this is **by supplying a state monad which can generate names and run the type checker within it**

```haskell
type TI a = ErrorT String (ReaderT TIEnv (StateT TIState IO)) a

data TIEnv = TIEnv  {}

data TIState = TIState {  tiSupply :: Int,
                          tiSubst :: Subst}

runTI :: TI a -> IO (Either String a, TIState)
runTI t =
    do (res, st) <- runStateT (runReaderT (runErrorT t)
                                initTIEnv) initTIState
       return (res, st)
  where initTIEnv = TIEnv {}
        initTIState = TIState { tiSupply = 0,
                                tiSubst = Map.empty }

newTyVar :: String -> TI Type
newTyVar prefix =
    do  s <- get
        put s { tiSupply = tiSupply s + 1 }
        return (TVar  (prefix ++ show (tiSupply s)))
```

  - TI = type inventor
- The final preliminary is an instantiation function which replaces bound type variables with fresh new type variables

```haskell
instantiate :: Scheme -> TI Type
instantiate (Scheme vars t) =
     do  nvars <- mapM (\ _ -> newTyVar "a") vars
         let s = Map.fromList (zip vars nvars)
         return $ apply s t
```

Now that we have all the machinery in place we can begin inferring types...

```haskell
tiConst :: TypeEnv -> Val -> TI (Subst, Type)
tiConst _ (I _)   = return (nullSubst, TInt)
tiConst _ (B _)  =  return (nullSubst, TBool)
```

- **Now for the algorithm to infer types for expressions**

```haskell
ti          ::  TypeEnv -> Exp -> TI (Subst, Type)
```

-

```haskell
ti (TypeEnv env) (Var n) =
  case Map.lookup n env of
     Nothing    ->  throwError $ "unbound variable: " ++ n
     Just sigma ->  do t <- instantiate sigma
                        return (nullSubst, t)

ti env (Const l) = tiConst env l

ti env (Abs n e) =
 do tv <- newTyVar "a"
    let TypeEnv env' = remove env n
        env'' = TypeEnv (env' `Map.union`
                     (Map.singleton n (Scheme [] tv)))
    (s1, t1) <- ti env'' e
    return (s1, TFun (apply s1 tv) t1)

ti env (App e1 e2) =
    do  tv <- newTyVar "a"
        (s1, t1) <- ti env e1
        (s2, t2) <- ti (apply s1 env) e2
        s3 <- mgu (apply s2 t1) (TFun t2 tv)
        return (s3 `composeSubst` s2
                   `composeSubst` s1, apply s3 tv)

ti env (Let x e1 e2) =
    do  (s1, t1) <- ti env e1
        let TypeEnv env' = remove env x
            t' = generalize (apply s1 env) t1
            env'' = TypeEnv (Map.insert x t' env')
        (s2, t2) <- ti (apply s1 env'') e2
        return (s1 `composeSubst` s2, t2)
```

-
    - Infer the type of some variable n
    - Infer the type of some constant l
    - Infer the type of an abstraction using the ABS rule
    - Infer the type of an application using the APP rule
    - Infer the type of a let using the LET rule
- **When inferring types we often need to resolve potential conflicts**
    - We deferred this to the mgu function
- **This is the unification algorithm of Robinson**

```
mgu :: Type -> Type -> TI Subst
mgu (TFun l r) (TFun l' r')
   =  do  s1 <- mgu l l'
          s2 <- mgu (apply s1 r) (apply s1 r')
          return (s1 `composeSubst` s2)

mgu (TVar u) t               =  varBind u t
mgu t (TVar u)               =  varBind u t

mgu TInt TInt                =  return nullSubst
mgu TBool TBool              =  return nullSubst
mgu t1 t2

  =  throwError $ "types do not unify: " ++ show t1 ++
                  " vs. " ++ show t2

varBind :: String -> Type -> TI Subst
varBind u t
   | t == TVar u           =  return nullSubst
   | u `Set.member` ftv t  =  throwError $
                                "occur check fails: " ++
                                " vs. " ++ show t
   | otherwise             =  return (Map.singleton u t)
```

All done! We just need an entry point and we are good to go:

```
typeInference :: Map.Map String Scheme -> Exp -> TI Type
typeInference env e =
    do  (s, t) <- ti (TypeEnv env) e
        return (apply s t)
```