Mark P. Jones' *Typing Haskell In Haskell* (1999)

Matthew Doty

Introduction

This paper presents a formal description of the Haskell type system using the notation of Haskell itself as a specification language.

The type checker is available on Hackage:

https://hackage.haskell.org/package/thih

Preliminaries

For simplicity, we present the code for our typechecker as a single Haskell module. The program uses only a handful of standard prelude functions, like `map`, `concat`, `all`, `any`, `mapM`, etc., and a few operations from the `List` and `Monad` libraries:

```
{-# LANGUAGE DerivingVia, MultiWayIf #-}
module TypingHaskellInHaskell where

import Prelude
import Data.List (nub, (\\), intersect, union, partition)
import Control.Monad (msum, ap, liftM)
```

Notation: Table of Conventions

For the most part, our choice of variable names follows the notational conventions:

Description	Symbol	Type
kind	`k, ...`	`Kind`
type constructor	`tc, ...`	`Tycon`
type variable	`v, ...`	`Tyvar`
- “fixed”	`f, ...`	
- “generic”	`g, ...`	
type	`t, ...`	`Type`
class	`c, ...`	`Class`
instance	`it, ...`	`Inst`
predicate	`p, q, ...`	`Pred`
- “deferred”	`d, ...`	
- “retained”	`r, ...`	

Notation: Table of Conventions Cont.

Description	Symbol	Type
qualified type	`qt, ...`	`QualType`
class environment	`ce, ...`	`ClassEnv`
scheme	`sc, ...`	`Scheme`
substitution	`s, ...`	`Subst`
unifier	`u, ...`	`Subst`
assumption	`a, ...`	`Assump`
identifier	`i, ...`	`Id`
literal	`l, ...`	`Literal`
pattern	`pat, ...`	`Pat`
expression	`e, f, ...`	`Expr`
alternative	`alt, ...`	`Alt`
binding group	`bg, ...`	`BindGroup`

Notation: Variable Conventions

A trailing `s` on a variable name usually indicates a list. Numeric suffices or primes are used as further decoration where necessary. For example, we use `k` or `k'` for a kind, and `ks` or `ks'` for a list of kinds. The types and terms appearing in the table are described more fully in later sections.

Notation: Identifiers

Throughout this paper, we implement identifiers as strings, and assume that there is a simple way to generate identifiers from integers using the `enumId` function:

```
type Id  = String

enumId  :: Int → Id
enumId n = "v" ++ show n
```

The `enumId` function will be used in the definition of the `newTVar` operator in a *type inference monad* to describe the allocation of fresh type variables during type inference.

Kinds: Basics

To ensure that they are valid, Haskell type constructors are classified into different *kinds*: the kind `*` (pronounced “star”) represents the set of all simple (i.e., nullary) type expressions, like `Int` and `Char -> Bool`.

⟨ Note: As of GHC 4.9.0 `*` has been deprecated in favor of `Data.Kind.Type` ⟩

Kinds of the form `k1 -> k2` represent type constructors that take an argument type of kind `k1` to a result type of kind `k2`.

For example, the standard list, `Maybe` and `IO` constructors all have kind `* -> *`.

Kinds: ADT Representation

We will represent kinds as values of the following datatype:

```
data Kind  = Star
           | Kfun Kind Kind
             deriving Eq
```

Kinds: Extensions

Kinds play essentially the same role for type constructors as types do for values, but the kind system is clearly very primitive. There are a number of extensions that would make interesting topics for future research, including polymorphic kinds, subkinding, and record/product kinds.

Types: Basics

The next step is to define a representation for types. Stripping away syntactic sugar, Haskell type expressions are either type variables or constants (each of which has an associated kind), or applications of one type to another. Applying a type of kind $\mathsf{k1} \rightarrow \mathsf{k2}$ to a type of kind $\mathsf{k1}$ produces a type of kind $\mathsf{k2}$.

Types: ADT Representation

```
data Type  = TVar Tyvar
           | TCon Tycon
           | TAp  Type Type
           | TGen Int
             deriving Eq

data Tyvar = Tyvar Id Kind
             deriving Eq

data Tycon = Tycon Id Kind
             deriving Eq
```

Types: Generic Type Variables

This definition also includes types of the form `TGen n`, which represent "generic" or quantified type variables. The only place where `TGen` values are used is in the representation of type schemes.

Types: Primitive Datatype Types

The following examples show how standard primitive datatypes are represented as type constants:

```
tUnit, tChar, tInt, tInteger, tFloat, tDouble :: Type
tUnit    = TCon (Tycon "()" Star)
tChar    = TCon (Tycon "Char" Star)
tInt     = TCon (Tycon "Int" Star)
tInteger = TCon (Tycon "Integer" Star)
tFloat   = TCon (Tycon "Float" Star)
tDouble  = TCon (Tycon "Double" Star)
```

Types: Primitive Datatype Types (Cont.)

```
tList, tArrow, tTuple2 :: Type
tList    = TCon (Tycon "[]"
                       (Kfun Star Star))
tArrow   = TCon (Tycon "(→)"
                       (Kfun Star (Kfun Star Star)))
tTuple2  = TCon (Tycon "(,)"
                       (Kfun Star (Kfun Star Star)))
```

Types: Exercise

How do we represent the kind for the `StateT` *transformer?*

Types: Composing Type Representations

More complex types are built up from constants and variables using the `TAp` constructor. For example, the representation for the type `Int → [a]` is as follows:

```
TAp (TAp tArrow tInt) (TAp tList (TVar (Tyvar "a" Star)))
```

Types: Type Synonyms

We do not provide a representation for type synonyms, assuming instead that they have been fully expanded before typechecking. For example, the `String` type-a synonym for `[Char]`-is represented as:

```
tString    :: Type
tString     = list tChar
```

It is always possible for an implementation to expand synonyms in this way because Haskell prevents the use of a synonym without its full complement of arguments. Moreover, the process is guaranteed to terminate because recursive synonym definitions are prohibited.

Types: Construction Helper Functions

We end this section with the definition of a few helper functions. The first three provide simple ways to construct function, list, and pair types, respectively:

```
infixr      4 `fn`
fn         :: Type → Type → Type
a `fn` b    = TAp (TAp tArrow a) b

list       :: Type → Type
list t      = TAp tList t

pair       :: Type → Type → Type
pair a b    = TAp (TAp tTuple2 a) b
```

Types: Kind Function

We also define an overloaded function, **kind**, for type variables, constants, or expressions:

```
class HasKind t where kind :: t → Kind

instance HasKind Tyvar where kind (Tyvar _v k) = k
instance HasKind Tycon where kind (Tycon _v k) = k
instance HasKind Type where
  kind (TCon tc) = kind tc
  kind (TVar u) = kind u
  kind (TAp t _) =
    case (kind t) of
      (Kfun _ k) → k
      _          → error "Kind must be a Kfun!"
  kind (TGen _) = error "Cannot give a kind to a TGen!"
```

Types: Kind Function (Cont.)

Most of the cases here are straightforward.

Notice, however, that we can calculate the kind of an application `(TAp t t')` using only the kind of its first argument `t`.

Assuming that the type is well-formed, `t` must have a kind $k' \to k$, where k' is the kind of `t'` and k is the kind of the whole application.

This shows that we need only traverse the leftmost spine of a type expression to calculate its kind.

Substitutions

Substitutions-finite functions, mapping type variables to types-play a major role in type inference.

Substitutions are represented using association lists:

```
type Subst  = [(Tyvar, Type)]
```

To ensure that we work only with well-formed type expressions, we will be careful to construct only *kind-preserving* substitutions in which variables are mapped only to types of the same kind.

Substitutions: Null Substitution

The simplest substitution is the null substitution, represented by the empty list, which is obviously kind-preserving:

```
nullSubst  :: Subst
nullSubst   = []
```

Substitutions: Singleton Substitution

Almost as simple are the substitutions `(u +→ t)` that map a single variable `u` to a type `t` of the same kind:

```
(+→)       :: Tyvar → Type → Subst
u +→ t      = [(u, t)]
```

This is kind-preserving if, and only if, `kind u = kind t`.

Substitutions: Application

Substitutions can be applied to types-and, in fact, to any other value with type components-in a natural way.

We overload the operation to `apply` a substitution so that it can work on different types of objects:

```
class Types t where
  apply :: Subst → t → t
  tv    :: t → [Tyvar]
```

In each case, the purpose of applying a substitution is the same: To replace every occurrence of a type variable in the domain of the substitution with the corresponding type.

We also include a function `tv` that returns the set of type variables (i.e., `Tyvar`s) appearing in its argument, listed in order of first occurrence (from left to right), with no duplicates.

Substitutions: Application Instance for *Type*

The definitions of these operations for `Type` are as follows:

```
instance Types Type where
  apply s (TVar u)  = case lookup u s of
                        Just t  → t
                        Nothing → TVar u
  apply s (TAp l r) = TAp (apply s l) (apply s r)
  apply _s t        = t

  tv (TVar u)  = [u]
  tv (TAp l r) = tv l `union` tv r
  tv _t        = []
```

Substitutions: Application Instance for List

It is straightforward (and useful!) to extend these operations to work on lists:

```
instance Types a ⇒ Types [a] where
  apply s = map (apply s)
  tv      = nub . concat . map tv
```

Substitutions: Sequential Composition

The `apply` function can be used to build more complex substitutions. For example, composition of substitutions, satisfying `apply (s1 @@ s2) = apply s1 . apply s2`, can be defined using:

```
infixr 4 @@
(@@)        :: Subst → Subst → Subst
s1 @@ s2    = [ (u, apply s1 t) | (u,t) ← s2 ] ++ s1
```

Substitutions: Parallel Composition

We can also form a "parallel" composition `s1++s2` of two substitutions `s1` and `s2`, but the result is left-biased because bindings in `s1` take precedence over any bindings for the same variables in `s2`.

For a more symmetric version of this operation, we use a `merge` function, which checks that the two substitutions agree at every variable in the domain of both and hence guarantees that:

```
apply (s1++s2) = apply (s2++s1)
```

Clearly, this is a partial function, which we reflect by arranging for `merge` to return its result in a monad, using the standard `fail` function to provide a string diagnostic in cases where the function is undefined.

Substitutions: Parallel Composition (Cont.)

```
merge :: Monad m ⇒ Subst → Subst → m Subst
merge s1 s2 =
  if agree
    then return (s1 ++ s2)
    else fail "merge fails"
  where
    agree =
      all
        (\v → apply s1 (TVar v) == apply s2 (TVar v))
        (map fst s1 `intersect` map fst s2)
```

Substitutions: Conclusion

It is easy to check that both `(@@)` and `merge` produce kind-preserving results from kind-preserving arguments.

In the next section, we will see how the first of these composition operators is used to describe unification, while the second is used in the formulation of a matching operation.

Introduction To MGUs

The goal of unification is to find a substitution that makes two types equal-for example, to ensure that the domain type of a function matches up with the type of an argument value.

However, it is also important for unification to find as "small" a substitution as possible because that will lead to most general types.

More formally, a substitution `s` is a *unifier* of two types `t1` and `t2` if `apply s t1 = apply s t2`.

A *most general unifier*, or *mgu*, of two such types is a unifier `u` with the property that any other unifier `s` can be written as `s'@@u`, for some substitution `s'`.

MGU exercise

What are the MGUs (up to α-equivalency) for the following pairs?

```
f :: a → b → b
const :: c → d → c
```

```
id :: a → a
($) :: (b → c) → b → c
```

```
f :: a → b → b
g :: (c → d → e) → (c → d) → e → e
```

Unification: Decidability and Unity

The syntax of Haskell types has been chosen to ensure that, if two types have any unifying substitutions, then they have a most general unifier, which can be calculated.

One of the reasons for this is that there are no non-trivial equalities on types. Extending the type system with higher-order features (such as lambda expressions on types), or with other mechanisms that allow reductions or rewriting in the type language, could make unification undecidable, non-unitary (meaning that there may not be most general unifiers), or both.

This, for example, is why Haskell does not allow type synonyms to be partially applied (and interpreted as some restricted kind of lambda expression).

MGU Implementation

The calculation of most general unifiers is implemented by a pair of functions:

```
mgu     :: Monad m ⇒ Type → Type → m Subst
varBind :: Monad m ⇒ Tyvar → Type → m Subst
```

These functions return results in a monad, capturing the fact that unification is a partial function.

MGU Implementation (Cont. I)

The main algorithm is described by **mgu**, using the structure of its arguments to guide the calculation:

```
mgu (TAp l r) (TAp l' r') = do
  s1 ← mgu l l'
  s2 ← mgu (apply s1 r) (apply s1 r')
  return (s2 @@ s1)
mgu (TVar u) t            = varBind u t
mgu t (TVar u)            = varBind u t
mgu (TCon tc1) (TCon tc2)
               | tc1==tc2 = return nullSubst
mgu _t1 _t2               = fail "types do not unify"
```

MGU Implementation (Cont. II)

The `varBind` function is used for the special case of unifying a variable `u` with a type `t`.

At first glance, one might think that we could just use the substitution `(u+→t)` for this.

In practice, however, tests are required to ensure that this is valid, including an "occurs check" (`u `elem` tv t`) and a test to ensure that the substitution is kind-preserving:

```
varBind u t | t == TVar u      = return nullSubst
            | u `elem` tv t    = fail "occurs check fails"
            | kind u /= kind t = fail "kinds do not match"
            | otherwise        = return (u +→ t)
```

Unification: Matching

In the following sections, we will also make use of an operation called *matching* that is closely related to unification.

Given two types `t1` and `t2`, the goal of matching is to find a substitution `s` such that `apply s t1 = t2`.

Because the substitution is applied only to one type, this operation is often described as *one-way* matching.

Unification: Matching (Cont.)

Matching follows the same pattern as unification, except that it uses `merge` rather than `@@` to combine substitutions, and it does not allow binding of variables in `t2`:

```
match :: Monad m ⇒ Type → Type → m Subst
match (TAp l r) (TAp l' r') = do sl ← match l l'
                                 sr ← match r r'
                                 merge sl sr
match (TVar u)   t
         | kind u == kind t = return (u +→ t)
match (TCon tc1) (TCon tc2)
         | tc1==tc2         = return nullSubst
match _t1 _t2               = fail "types do not match"
```

Type Classes: Introduction

One of the most unusual features of the Haskell type system, at least in comparison to those of other polymorphically typed languages like ML, is the support that it provides for *type classes*.

A significant portion of the code presented in this paper, particularly in this section, is needed to describe the handling of type classes in Haskell.

Type Classes: Qualified Types Representation

A Haskell type class can be thought of as a set of types (of some particular kind), each of which supports a certain collection of *member functions* that are specified as part of the class declaration.

The types in each class (known as `instances`) are specified by a collection of instance declarations.

Haskell types can be *qualified* by adding a (possibly empty) list of *predicates*, or class constraints, to restrict the ways in which type variables are instantiated:

```
data Qual t = [Pred] :=> t
            deriving Eq
```

In a value of the form `ps :=> t`, we refer to `ps` as the *context* and to `t` as the *head*.

Type Classes: Type Predicates

Predicates themselves consist of a class identifier and a type; a predicate of the form `IsIn i t` asserts that `t` is a member of the class named `i`:

```
data Pred   = IsIn Id Type
            deriving Eq
```

Type Classes: Predicate/Qualified Type Example

For example, using the `Qual` and `Pred` datatypes, the type `(Num a) ⇒ a → Int` can be represented by:

```
[IsIn "Num" (TVar (Tyvar "a" Star))]
  :=> (TVar (Tyvar "a" Star) `fn` tInt)
```

Type Classes: Extensions

It would be easy to extend the `Pred` datatype to allow other forms of predicate, as is done with Trex records in Hugs.

Another frequently extension is to allow classes to accept multiple parameters, which would require a list of `Type`s rather than the single `Type` in the definition above.

Type Classes: Application Instances for Qual and Pred

```
instance Types t ⇒ Types (Qual t) where
  apply s (ps :=> t) = apply s ps :=> apply s t
  tv (ps :=> t)      = tv ps `union` tv t

instance Types Pred where
  apply s (IsIn i t) = IsIn i (apply s t)
  tv (IsIn _i t)     = tv t
```

Type Classes: MGUs And Matches for Predicates

```
mguPred, matchPred :: Pred → Pred → Maybe Subst
mguPred             = lift mgu
matchPred           = lift match

lift :: Monad m
     ⇒ (Type → Type → m a)
     → Pred
     → Pred
     → m a
lift m (IsIn i t) (IsIn i' t')
  | i == i' = m t t'
  | otherwise = fail "classes differ"
```

Type Classes: Representation

We will represent each class by a pair of lists, one containing the name of each superclass, and another containing an entry for each instance declaration:

```
type Class    = ([Id], [Inst])
type Inst     = Qual Pred
```

Type Classes: Representation Example

For example, a simplified version of the standard Haskell class `Ord` might be described by the following value of type `Class`:

```
( ["Eq"]
, [ [] :=> IsIn "Ord" tUnit
  , [] :=> IsIn "Ord" tChar
  , [] :=> IsIn "Ord" tInt
  , [ IsIn "Ord" (TVar (Tyvar "a" Star))
    , IsIn "Ord" (TVar (Tyvar "b" Star))]
    :=> IsIn "Ord" (pair (TVar (Tyvar "a" Star))
                         (TVar (Tyvar "b" Star)))
  ])
```

Type Classes: Representation Example (Cont.)

This structure captures the fact that `Eq` is a superclass of `Ord` (the only one in fact), and lists four instance declarations for the unit, character, integer, and pair types (if `a` and `b` are in `Ord`, then `(a,b)` is also in `Ord`).

Of course, this is only a fraction of the list of `Ord` instances that are defined in the full Haskell prelude.

Type Classes: Class Environments

The information provided by the class and instance declarations in a given program can be captured by a class environment of type:

```
data ClassEnv = ClassEnv { classes  :: Id → Maybe Class,
                           defaults :: [Type] }
```

The `classes` component in a `ClassEnv` value is a partial function that maps identifiers to `Class` values (or to `Nothing` if there is no class corresponding to the specified identifier).

Type Classes: Class Environment Accessors

We define helper functions `super` and `insts` to extract the list of superclass identifiers, and the list of instances, respectively, for a class name `i` in a class environment `ce`:

```
super     :: ClassEnv → Id → [Id]
super ce i = case classes ce i of
               Just (is, _its) → is
               Nothing → []

insts     :: ClassEnv → Id → [Inst]
insts ce i = case classes ce i of
               Just (_is, its) → its
               Nothing → []
```

Type Classes: Guard For Safe Class Enviroment Accessing

These functions are intended to be used only in cases where it is known that the class `i` is defined in the environment `ce`.

We can use a dynamic check by testing `defined (classes ce i)` before applying either function.

```
defined :: Maybe a → Bool
defined (Just _x) = True
defined Nothing   = False
```

Type Classes: Class Enviromnet Modification

We will also define a helper function, `modify`, to describe how a class environment can be updated to reflect a new binding of a `Class` value to a given identifier:

```
modify :: ClassEnv → Id → Class → ClassEnv
modify ce i c =
  ce
    { classes =
        \j →
          if i == j
            then Just c
            else classes ce j
    }
```

Type Classes: Defaulting

The `defaults` component of a `ClassEnv` value is used to provide a list of types for defaulting.

Haskell allows programmers to specify a value for this list using a `default` declaration; if no explicit declaration is given, then a `default (Integer,Double)` declaration is assumed.

It is easy to describe this using the `ClassEnv` type.

For example, `cedefaults=[tInt]` is the result of modifying a class environment `ce` to reflect the presence of a `default (Int)` declaration.

Type Classes: Intro To Building Class Environments

In the remainder of this section, we will show how to build an appropriate class environment for a given program, starting from an (almost) empty class environment, and extending it as necessary to reflect the effect of each class or instance declaration in the program.

Type Classes: Initial Class Environment

The initial class environment is defined as follows:

```
initialEnv :: ClassEnv
initialEnv =
  ClassEnv
    { classes = \_i → fail "class not defined"
    , defaults = [tInteger, tDouble]
    }
```

Type Classes: Class Environment Transformations

As we process each class or instance declaration in a program, we transform the initial class environment to add entries, either for a new class, or for a new instance, respectively.

In either case, there is a possibility that the new declaration might be incompatible with the previous declarations, attempting, for example, to redefine an existing class or instance.

For this reason, we will describe transformations of a class environment as functions of the `EnvTransformer` type, using a `Maybe` type to allow for the possibility of errors:

```
type EnvTransformer = ClassEnv → Maybe ClassEnv
```

Type Classes: Composing Class Environment Transformers

The sequencing of multiple transformers can be described by a (forward) composition operator `(<:>)`:

```
infixr 5 <:>

(<:>) :: EnvTransformer → EnvTransformer → EnvTransformer
(f <:> g) ce = do
  ce' ← f ce
  g ce'
```

Some readers will recognize this as a special case of the more general Kleisli composition operator `(>=>)`.

Type Classes: Adding Classes To Environments

To add a new class to an environment, we must check that there is not already a class with the same name, and that all of the named superclasses are already defined.

This is a simple way of enforcing Haskell's restriction that the superclass hierarchy be acyclic.

```
addClass :: Id → [Id] → EnvTransformer
addClass i is ce
  | defined (classes ce i) =
      fail "class already defined"
  | any (not . defined . classes ce) is =
      fail "superclass not defined"
  | otherwise = return (modify ce i (is, []))
```

Type Classes: Adding Prelude Classes (Example)

For example, we can describe the effect of the class declarations in the Haskell prelude using the following transformer:

```
addPreludeClasses :: EnvTransformer
addPreludeClasses  = addCoreClasses <:> addNumClasses
```

This definition breaks down the set of standard Haskell classes into two separate pieces, `addCoreClasses` and `addNumClasses`.

Type Classes: Adding Prelude Classes (Example, Cont. I)

The core classes are described as follows:

```
addCoreClasses ::   EnvTransformer
addCoreClasses  =
      addClass "Eq" []
  <:> addClass "Ord" ["Eq"]
  <:> addClass "Show" []
  <:> addClass "Read" []
  <:> addClass "Bounded" []
  <:> addClass "Enum" []
  <:> addClass "Functor" []
  <:> addClass "Monad" []
```

Type Classe: Adding Prelude Classes (Example, Cont. II)

The hierarchy of numeric classes is captured separately in the following definition:

```
addNumClasses ::   EnvTransformer
addNumClasses  =
      addClass "Num" ["Eq", "Show"]
  <:> addClass "Real" ["Num", "Ord"]
  <:> addClass "Fractional" ["Num"]
  <:> addClass "Integral" ["Real", "Enum"]
  <:> addClass "RealFrac" ["Real", "Fractional"]
  <:> addClass "Floating" ["Fractional"]
  <:> addClass "RealFloat" ["RealFrac", "Floating"]
```

Type Classes: Adding Instances (Implementation)

To add a new instance to a class, we must check that the class to which the instance applies is defined, and that the new instance does not overlap with any previously declared instance:

```
addInst :: [Pred] → Pred → EnvTransformer
addInst ps p@(IsIn i _) ce
 | not (defined (classes ce i)) =
     fail "no class for instance"
 | any (overlap p) qs           =
     fail "overlapping instance"
 | otherwise                    =
     return (modify ce i c)
   where its = insts ce i
         qs  = [ q | (_ :=> q) ← its ]
         c   = (super ce i, (ps:=>p) : its)
```

Type Classes: Overlapping Instances

Two instances for a class are said to *overlap* if there is some predicate that is a substitution instance of the heads of both instance declarations. It is easy to test for overlapping predicates using the functions that we have defined previously:

```
overlap :: Pred → Pred → Bool
overlap p q    = defined (mguPred p q)
```

We treat overlapping instances as an error. Modern extensions exist to provide support for them.

Type Classes: Adding Instances (Example)

To illustrate how the `addInst` function might be used, the following definition shows how the standard prelude class environment can be extended to include the four instances for `Ord`:

```
exampleInsts :: EnvTransformer
exampleInsts =
      addPreludeClasses
  <:> addInst [] (IsIn "Ord" tUnit)
  <:> addInst [] (IsIn "Ord" tChar)
  <:> addInst [] (IsIn "Ord" tInt)
  <:> addInst [IsIn "Ord" (TVar (Tyvar "a" Star)),
               IsIn "Ord" (TVar (Tyvar "b" Star))]
              (IsIn "Ord" (pair (TVar (Tyvar "a" Star))
                                (TVar (Tyvar "b" Star))))
```

Type Classes: Entailment

In this section, we describe how class environments can be used to answer questions about which types are instances of particular classes.

More generally, we consider the treatment of *entailment*: given a predicate `p` and a list of predicates `ps`, our goal is to determine whether `p` will hold whenever all of the predicates in `ps` are satisfied.

In the special case where `p = IsIn i t` and `ps = []`, this amounts to determining whether `t` is an instance of the class `i`.

Type Classes: Super Classes

As a first step, we can ask how information about superclasses and instances can be used independently to help reason about entailments.

For example, if a type is an instance of a class `i`, then it must also be an instance of any superclasses of `i`.

Hence, using only superclass information, we can be sure that, if a given predicate `p` holds, then so too must all of the predicates in the list `bySuper p`:

```
bySuper :: ClassEnv → Pred → [Pred]
bySuper ce p@(IsIn i t)
 = p : concat [ bySuper ce (IsIn i' t)
              | i' ← super ce i ]
```

Type Classes: Entailment By Instances

Next we consider how information about instances can be used. Of course, for a given predicate `p = IsIn i t`, we can find all the directly relevant instances in a class environment `ce` by looking in `insts ce i`.

As we have seen, individual instance declarations are mapped into clauses of the form `ps :=> h`.

The head predicate `h` describes the general form of instances that can be constructed from this declaration, and we can use `matchPred` to determine whether this instance is applicable to the given predicate `p`.

If it is applicable, then matching will return a substitution `u`, and the remaining subgoals are the elements of `map (apply u) ps`.

Type Classes: Entailment By Instances Implementation

The following function uses these ideas to determine the list of subgoals for a given predicate:

```
byInst                          :: ClassEnv → Pred → Maybe [Pred]
byInst ce p@(IsIn i _t)    = msum [ tryInst it | it ← insts ce i ]
 where tryInst (ps :=> h) = do u ← matchPred h p
                               Just (map (apply u) ps)
```

Because we are not supporting overlapping instances, there is at most one applicable instance for any given `p`, and we can be sure that the first defined element will actually be the *only* defined element in this list.

Type Classes: Entailment Implementation

The `bySuper` and `byInst` functions can be used in combination to define a general entailment operator, `entail`.

Given a particular class environment `ce`, the intention here is that `entail ce ps p` will be `True` if, and only if, the predicate `p` will hold whenever all of the predicates in `ps` are satisfied:

```
entail        :: ClassEnv → [Pred] → Pred → Bool
entail ce ps p = any (p `elem`) (map (bySuper ce) ps) ||
                 case byInst ce p of
                   Nothing → False
                   Just qs → all (entail ce ps) qs
```

Type Classes: Entailment Implementation (Cont.)

The first step here is to determine whether `p` can be deduced from `ps` using only superclasses.

If that fails, we look for a matching instance and generate a list of predicates `qs` as a new goal, each of which must, in turn, follow from `ps`.

Type Classes: Entailment Conclusion

Conditions specified in the Haskell 98 report-namely that the class hierarchy is acyclic and that the types in any instance declaration are strictly smaller than those in the head-translate into conditions on the values for the `ClassEnv` that can be passed in as `ce`, and these are enough to guarantee that tests for entailment will terminate.

Completeness of the algorithm is also important: will `entail ce ps p` always return `True` whenever there is a way to prove `p` from `ps`?

Type Classes: Entailment Properties (Cont. I)

In fact our algorithm does not cover all possible cases: it does not test to see if `p` is a superclass of some other predicate `q` for which `entail ce ps q` is `True`.

Extending the algorithm to test for this would be very difficult because there is no obvious way to choose a particular `q`, and, in general, there will be infinitely many potential candidates to consider.

Type Classes: Entailment Properties (Cont. II)

Fortunately, a technical condition in the Haskell 98 report [Condition 1 on Page 47] reassures us that this is not necessary.

If `p` can be obtained as an immediate superclass of some predicate `q` that was built using an instance declaration in an entailment `entail ce ps q`, then `ps` must already be strong enough to deduce `p`.

Thus, although we have not formally proved these properties, we believe that our algorithm is sound, complete, and guaranteed to terminate.

Type Classes: Context Reduction

Class environments also play an important role in an aspect of the Haskell type system that is known as *context reduction*.

The basic goal of context reduction is to reduce a list of predicates to an equivalent but, in some sense, simpler list.

Type Classes: Context Reduction Example

One way to simplify a list of predicates is to simplify the type components of individual predicates in the list.

For example, given the instance declarations in the Haskell standard prelude, we could replace any occurrences of predicates like `Eq [a]`, `Eq (a,a)`, or `Eq ([a],Int)` with `Eq a`.

This is valid because, for any choice of `a`, each one of these predicates holds if, and only if, `Eq a` holds.

Type Classes: Context Reduction Example (Cont. I)

Notice that, in some cases, an attempt to simplify type components-for example, by replacing `Eq (a, b)` with `(Eq a, Eq b)`-may increase the number of predicates in the list.

The extent to which simplifications like this are used in a system of qualified types has an impact on the implementation and performance of overloading in practical systems.

Type Classes: Context Reduction & Head Normal Form

In Haskell, however, the decisions are made for us by a syntactic restriction that forces us to simplify predicates until we obtain types in a kind of "head-normal form".

This terminology is motivated by similarities with the concept of *head-normal forms* in the λ-calculus.

More precisely, the syntax of Haskell requires class arguments to be of the form `v t1 ... tn`, where `v` is a type variable, and `t1`,...,`tn` are types.

Type Classes: Head-Normal Form Check Implementation

The following function allows us to determine whether a given predicate is in head-normal form:

```
inHnf          :: Pred → Bool
inHnf (IsIn _c t) = hnf t
 where hnf (TVar _v)   = True
       hnf (TCon _tc)  = False
       hnf (TGen _i)   = False
       hnf (TAp t' _)  = hnf t'
```

Type Classes: Head-Norm Form Reduction

Predicates that are not in head-normal form must be broken down using `byInst`.

In some cases, this will result in predicates being eliminated altogether.

In others, where `byInst` fails, it will indicate that a predicate is unsatisfiable, and will trigger an error diagnostic.

Type Classes: Head-Norm Form Reduction Implementation

Head-normal form reduction is captured in the following definition:

```
toHnfs :: Monad m ⇒ ClassEnv → [Pred] → m [Pred]
toHnfs ce ps = do
  pss ← mapM (toHnf ce) ps
  return (concat pss)

toHnf :: Monad m ⇒ ClassEnv → Pred → m [Pred]
toHnf ce p
  | inHnf p = return [p]
  | otherwise =
    case byInst ce p of
      Nothing → fail "context reduction"
      Just ps → toHnfs ce ps
```

Type Classes: Context Simplification

Another way to simplify a list of predicates is to reduce the number of elements that it contains.

There are several ways that this might be achieved: by removing duplicates (e.g., reducing `(Eq a, Eq a)` to `Eq a`); by eliminating predicates that are already known to hold (e.g., removing any occurrences of `Num Int`); or by using superclass information (e.g., reducing `(Eq a, Ord a)` to `Ord a`).

In each case, the reduced list of predicates, is equivalent to the initial list, meaning that all the predicates in the first will be satisfied if, and only if, all of the predicates in the second are satisfied.

Type Classes: Context Simplification (Cont.)

The simplification algorithm that we will use here is based on the observation that a predicate `p` in a list of predicates `(p:ps)` can be eliminated if `p` is entailed by `ps`.

As a special case, this will eliminate duplicated predicates: if `p` is repeated in `ps`, then it will also be entailed by `ps`.

Type Classes: Context Simplification Implementation

These ideas are used in the following definition of the `simplify` function, which loops through each predicate in the list and uses an accumulating parameter to build up the final result.

Each time we encounter a predicate that is entailed by the others, we remove it from the list.

```
simplify :: ClassEnv → [Pred] → [Pred]
simplify ce = loop []
  where
    loop rs [] = rs
    loop rs (p:ps)
      | entail ce (rs ++ ps) p = loop rs ps
      | otherwise = loop (p : rs) ps
```

Type Classes: Context Reduction Implementation

Now we can describe the particular form of context reduction used in Haskell as a combination of `toHnfs` and `simplify`.

Specifically, we use `toHnfs` to reduce the list of predicates to head-normal form, and then simplify the result:

```
reduce      :: Monad m ⇒ ClassEnv → [Pred] → m [Pred]
reduce ce ps = do qs ← toHnfs ce ps
                  return (simplify ce qs)
```

Type Classes: Context Reduction Implementation Notes

As a technical aside, we note that there is some redundancy in the definition of `reduce`.

The `simplify` function is defined in terms of `entail`, which makes use of the information provided by both superclass and instance declarations.

The predicates in `qs`, however, are guaranteed to be in head-normal form, and hence will not match instance declarations that satisfy the syntactic restrictions of Haskell.

Type Classes: Class Entailment

It follows that we could make do with a version of `simplify` that used only the following function in determining (superclass) entailments:

```
scEntail          :: ClassEnv → [Pred] → Pred → Bool
scEntail ce ps p = any (p `elem`) (map (bySuper ce) ps)
```

Type Schemes: Representation

Type schemes are used to describe polymorphic types, and are represented using a list of kinds and a qualified type:

```
data Scheme = Forall [Kind] (Qual Type)
            deriving Eq
```

Type Schemes: Representation (Cont.)

In a type scheme `Forall ks qt`, each type of the form `TGen n` that appears in the qualified type `qt` represents a generic, or universally quantified type variable whose kind is given by `ks !! n`.

This is the only place where we will allow `TGen` values to appear in a type.

Type Schemes: Types Instance For Scheme

```
instance Types Scheme where
  apply s (Forall ks qt) = Forall ks (apply s qt)
  tv (Forall _ks qt)     = tv qt
```

Type Schemes: Construction

Type schemes are constructed by quantifying a qualified type `qt` with respect to a list of type variables `vs`:

```
quantify       :: [Tyvar] → Qual Type → Scheme
quantify vs qt = Forall ks (apply s qt)
 where vs' = [ v | v ← tv qt, v `elem` vs ]
       ks  = map kind vs'
       s   = zip vs' (map TGen [0..])
```

Type Schemes: Construction (Canonical Form)

Note that the order of the kinds in `ks` is determined by the order in which the variables `v` appear in `tv qt`, and not by the order in which they appear in `vs`.

So, for example, the leftmost quantified variable in a type scheme will always be represented by `TGen 0`.

By insisting that type schemes are constructed in this way, we obtain a *unique canonical form* for `Scheme` values.

This is important because it means that we can test whether two type schemes are the same without having to tests for α-equivalence.

Type Schemes: Type To Scheme Conversion

In practice, we sometimes need to convert a `Type` into a `Scheme` without adding any qualifying predicates or quantified variables. For this special case, we can use the following function instead of `quantify`:

```
toScheme      :: Type → Scheme
toScheme t     = Forall [] ([] :=> t)
```

Type Schemes: Quantified Variables

To complete our description of type schemes, we need to be able to instantiate the quantified variables in `Scheme` values.

In fact, for the purposes of type inference, we only need the special case that instantiates a type scheme with fresh type variables.

We therefore defer further description of instantiation to §**Type Inference Monad** where the mechanisms for generating fresh type variables are introduced.

Assumptions

Assumptions about the type of a variable are represented by values of the `Assump` datatype, each of which pairs a variable name with a type scheme:

```
data Assump = Id :>: Scheme
```

Assumptions: `Types` Class Instantiation

Once again, we can extend the `Types` class to allow the application of a substitution to an assumption:

```
instance Types Assump where
  apply s (i :>: sc) = i :>: (apply s sc)
  tv (_i :>: sc)     = tv sc
```

Thanks to the instance definition for `Types` on lists (§**Substitutions**), we can also use the `apply` and `tv` operators on the lists of assumptions that are used to record the type of each program variable during type inference.

Assumptions: Finding The Types of Variables

We will also use the following function to find the type of a particular variable in a given set of assumptions:

```
find :: Monad m ⇒ Id → [Assump] → m Scheme
find i [] = fail ("unbound identifier: " ++ i)
find i ((i' :>: sc):as) =
  if i == i'
    then return sc
    else find i as
```

A Type Inference Monad

The purpose of this section is to define the monad that will be used in the description of the main type inference algorithm in §**Type Inference**.

Our choice of monad is motivated by the needs of maintaining a “current substitution” and of generating fresh type variables during typechecking.

A Type Inference Monad (Cont. I)

In a more realistic implementation, we might also want to add error reporting facilities, but in this paper the crude but simple `fail` function from the Haskell prelude is all that we require.

It follows that we need a simple state monad with only a substitution and an integer (from which we can generate new type variables) as its state.

A Type Inference Monad (Cont. II)

Note: Let's use Daniel Brice's "`deriving via`*" trick to skip some boilerplate!*

```
newtype MonadInstances m a = MonadInstances (m a)
  deriving Monad via m

instance Monad m ⇒ Functor (MonadInstances m) where
  fmap = liftM

instance Monad m ⇒ Applicative (MonadInstances m) where
  pure = return
  (<*>) = ap
```

https://twitter.com/fried_brice/status/1172271334170214400

A Type Inference Monad (Cont. III)

```
newtype TI a = TI (Subst → Int → (Subst, Int, a))
  deriving (Functor, Applicative) via MonadInstances TI

instance Monad TI where
  return x   = TI (\s n → (s,n,x))
  TI f >>= g = TI (\s n → case f s n of
                            (s',m,x) → let TI gx = g x
                                       in  gx s' m)

runTI              :: TI a → a
runTI (TI f)        = x where (_s,_n,x) = f nullSubst 0
```

Current Substitution `ask`

The `getSubst` operation returns the current substitution, while `unify` extends it with a most general unifier of its arguments:

```
getSubst    :: TI Subst
getSubst     = TI (\s n → (s,n,s))
```

Unification

The function `extSubst` extends the current substitution.

Unify takes two types, computes their most general unifier σ, and extends the current substitution with σ.

```
extSubst   :: Subst → TI ()
extSubst s' = TI (\s n → (s' <> s, n, ()))

unify      :: Type → Type → TI ()
unify t1 t2 = do s ← getSubst
                 u ← mgu (apply s t1) (apply s t2)
                 extSubst u
```

Fresh Type Variables

The function **newTVar** gets a new type variable and bumps the type variable counter in the state monad.

```
newTVar     :: Kind → TI Type
newTVar k    = TI (\s n → let v = Tyvar (enumId n) k
                          in  (s, n+1, TVar v))
```

Fresh Type Variables (Usage)

One place where `newTVar` is useful is in instantiating a type scheme with new type variables of appropriate kinds:

```
freshInst                :: Scheme → TI (Qual Type)
freshInst (Forall ks qt) = do ts ← mapM newTVar ks
                              return (inst ts qt)
```

The structure of this definition guarantees that `ts` has exactly the right number of type variables, and each with the right kind, to match `ks`.

Hence, if the type scheme is well-formed, then the qualified type returned by `freshInst` will not contain any unbound generics of the form `TGen n`.

Instantiating Quantified `TGen`

`freshInst` relies an auxiliary function `inst`.

This is a variation of `apply` that works on generic variables.

In other words, `inst ts t` replaces each occurrence of a generic variable `TGen n` in `t` with `ts !! n`.

Instantiating Quantified `TGen` (Cont.)

```
class Instantiate t where
  inst  :: [Type] → t → t
instance Instantiate Type where
  inst ts (TAp l r) = TAp (inst ts l) (inst ts r)
  inst ts (TGen n)  = ts !! n
  inst _ts t        = t
instance Instantiate a ⇒ Instantiate [a] where
  inst ts = map (inst ts)
instance Instantiate t ⇒ Instantiate (Qual t) where
  inst ts (ps :=> t) = inst ts ps :=> inst ts t
instance Instantiate Pred where
  inst ts (IsIn c t) = IsIn c (inst ts t)
```

Type Inference (Intro)

With this section we have reached the heart of the paper, detailing our algorithm for type inference.

It is here that we finally see how the machinery that has been built up in earlier sections is actually put to use.

We develop the complete algorithm in stages, working through the abstract syntax of the input language from the simplest part (literals) to the most complex (binding groups).

Type Inference Type Alias

Most of the typing rules are expressed by functions whose types are simple variants of the following synonym:

```
type Infer e t =    ClassEnv
                 → [Assump]
                 → e
                 → TI ([Pred], t)
```

Type Inference Formalization

In more theoretical treatments, it would not be surprising to see the rules expressed in terms of judgments $G; P | A \vdash \texttt{e} : \texttt{t}$, where:

- G is a class environment
- P is a set of predicates
- A is a set of assumptions
- `e` is an expression
- `t` is a corresponding type

Type Inference Formalization (Cont.)

Judgments like this can be thought of as 5-tuples, and the typing rules themselves just correspond to a 5-place relation.

Exactly the same structure shows up in types of the form `Infer e t`, except that, by using functions, we distinguish very clearly between input and output parameters.

(Note: Here's the declaration of `Infer` again for convenience.)

```
type Infer e t =    ClassEnv
                 → [Assump]
                 → e
                 → TI ([Pred], t)
```

Literals

Like other languages, Haskell provides special syntax for constant values of certain primitive datatypes, including numerics, characters, and strings.

We will represent these *literal* expressions as values of the `Literal` datatype:

```
data Literal = LitInt  Integer
             | LitChar Char
             | LitRat  Rational
             | LitStr  String
```

Type Inference For Literals

Type inference for literals is straightforward.

For characters, we just return `tChar`.

For integers, we return a new type variable `v` together with a predicate to indicate that `v` must be an instance of the `Num` class.

The cases for `String` and floating point literals follow similar patterns.

Type Inference For Literals (Cont.)

```
tiLit             :: Literal → TI ([Pred],Type)
tiLit (LitChar _) = return ([], tChar)
tiLit (LitInt _)  = do v ← newTVar Star
                       return ([IsIn "Num" v], v)
tiLit (LitStr _)  = return ([], tString)
tiLit (LitRat _)  = do v ← newTVar Star
                       return ([IsIn "Fractional" v], v)
```

Patterns

Patterns are used to inspect and deconstruct data values in lambda abstractions, function and pattern bindings, list comprehensions, do notation, and case expressions.

We will represent patterns using values of the `Pat` datatype:

```
data Pat         = PVar Id
                 | PWildcard
                 | PAs  Id Pat
                 | PLit Literal
                 | PNpk Id Integer
                 | PCon Assump [Pat]
```

Patterns (Cont. I)

A `PVar i` pattern matches any value and binds the result to the variable `i`.

A `PWildcard` pattern, corresponding to an underscore `_` in Haskell syntax, matches any value, but does not bind any variables.

A pattern of the form `(PAs i pat)`, known as an "as-pattern" and written using the syntax `i@pat` in Haskell, binds the variable `i` to any value that matches the pattern `pat`, while also binding any variables that appear in `pat`.

Patterns (Cont. II)

A `PLit l` pattern matches only the particular value denoted by the literal `l`.

A pattern `(PNpk i k)` is an `(n+k)` pattern, which matches any positive integral value `m` that is greater than or equal to `k`, and binds the variable `i` to the difference `(m-k)`.

Finally, a pattern of the form `PCon a pats` matches only values that were built using the constructor function `a` with a sequence of arguments that matches `pats`.

We use values `a` of type `Assump` to represent constructor functions; all that we really need for type-checking is the type, although the name is useful for debugging.

Patterns (Cont. III)

Most Haskell patterns have a direct representation in `Pat`, but extensions would be needed to account for patterns using labeled fields.

This is not difficult, but adds some complexity, which we prefer to avoid in this presentation.

Type Inference For Patterns

Type inference for patterns has two goals:

1. To calculate a type for each bound variable, and
2. to determine what type of values the whole pattern might match.

This leads us to look for a function:

```
tiPat :: Pat → TI ([Pred], [Assump], Type)
```

Note that we do not need to pass in a list of assumptions here; by definition, any occurrence of a variable in a pattern would hide rather than refer to a variable of the same name in an enclosing scope.

Type Inference For Patterns (`PVar`)

For a variable pattern, `PVar i`, we just return a new assumption, binding `i` to a fresh type variable.

```
tiPat (PVar i) = do v ← newTVar Star
                    return ([], [i :>: toScheme v], v)
```

Haskell does not allow multiple use of any variable in a pattern, so we can be sure that this is the first and only occurrence of `i` that we will encounter in the pattern.

Type Inference For Patterns (`PWildcard`)

Wildcards are typed in the same way except that we do not need to create a new assumption:

```
tiPat PWildcard   = do v ← newTVar Star
                       return ([], [], v)
```

Type Inference For Patterns (`PAs`)

To type an as-pattern `PAs i pat`, we calculate a set of assumptions and a type for the `pat` pattern, and then add an extra assumption to bind `i`:

```
tiPat (PAs i pat) = do (ps, as, t) ← tiPat pat
                       return (ps, (i:>:toScheme t):as, t)
```

Type Inference For Patterns (`PLit`)

For literal patterns, we use `tiLit` from the **§Type Inference For Literals** section:

```
tiPat (PLit l) = do (ps, t) ← tiLit l
                    return (ps, [], t)
```

Type Inference For Patterns (`PNpk`)

The rule for `(n+k)` patterns does not fix a type for the bound variable, but adds a predicate to constrain the choice to instances of the `Integral` class:

```
tiPat (PNpk i _k) = do
  t ← newTVar Star
  return ([IsIn "Integral" t], [i :>: toScheme t], t)
```

Type Inference For Patterns (PCon)

The case for constructed patterns is slightly more complex:

```
tiPat (PCon (_i :>: sc) pats) = do
  (ps, as, ts) ← tiPats pats
  t' ← newTVar Star
  (qs :=> t) ← freshInst sc
  unify t (foldr fn t' ts)
  return (ps ++ qs, as, t')
```

Type Inference For Patterns (`PCon` Explanation)

1. First use the `tiPats` function, defined below, to calculate types `ts` for each sub-pattern in `pats` together with corresponding lists of assumptions in `as` and predicates in `ps`.

2. Then generate a new type variable `t'` that will be used to capture the (as yet unknown) type of the whole pattern. From this information, we would expect the constructor function at the head of the pattern to have type `foldr fn t' ts`. We can check that this is possible by instantiating the known type `sc` of the constructor and unifying.

Inference For Lists of Patterns

The `tiPats` function is a variation of `tiPat` that takes a list of patterns as input, and returns a list of types (together with a list of predicates and a list of assumptions) as its result.

```
tiPats :: [Pat] → TI ([Pred], [Assump], [Type])
tiPats pats = do
  psasts ← mapM tiPat pats
  let ps = concat [ps' | (ps', _, _) ← psasts]
      as = concat [as' | (_, as', _) ← psasts]
      ts = [t | (_, _, t) ← psasts]
  return (ps, as, ts)
```

Aside from **PCon** handling, this function is useful in other situations where lists of patterns are used.

An example is the left hand side of an equation in a function definition.

Expressions

Next we describe type inference for expressions, represented by the `Expr` datatype:

```
data Expr = Var   Id
          | Lit   Literal
          | Const Assump
          | Ap    Expr Expr
          | Let   BindGroup Expr
```

Expressions (Cont.)

- The `Var` and `Lit` constructors are used to represent variables and literals, respectively.
- The `Const` constructor is used to deal with named constants, such as the constructor or selector functions associated with a particular datatype or the member functions that are associated with a particular class. We use values of type `Assump` to supply a name and type scheme.
- Function application is represented using the `Ap` constructor.
- `Let` is used to represent let expressions.

(Note that the definition of the `BindGroup` type, used here to represent binding groups, will be delayed to Binding Groups.)

Type Inference For Expressions

```
tiExpr :: Infer Expr Type
tiExpr _ce as (Var i) = do
  sc ← find i as
  (ps :=> t) ← freshInst sc
  return (ps, t)
tiExpr _ce _as (Const (_i :>: sc)) = do
  (ps :=> t) ← freshInst sc
  return (ps, t)
tiExpr _ce _as (Lit l) = do
  (ps, t) ← tiLit l
  return (ps, t)
```

Type Inference For Expressions (Cont. I)

```
tiExpr ce as (Ap e f) = do
  (ps, te) ← tiExpr ce as e
  (qs, tf) ← tiExpr ce as f
  t ← newTVar Star
  unify (tf `fn` t) te
  return (ps ++ qs, t)
tiExpr ce as (Let bg e) = do
  (ps, as') ← tiBindGroup ce as bg
  (qs, t) ← tiExpr ce (as' ++ as) e
  return (ps ++ qs, t)
```

Type Inference For Expressions (Cont. II)

The final case here, for `Let` expressions, uses the function `tiBindGroup` presented in Binding Groups, to generate a list of assumptions `as'` for the variables defined in `bg`.

All of these variables are in scope when we calculate a type `t` for the body `e`, which also serves as the type of the whole expression.

Alternatives

The representation of function bindings in following sections uses *alternatives*, represented by values of type `Alt`:

```
type Alt = ([Pat], Expr)
```

An `Alt` specifies the left and right hand sides of a function definition.

With a more complete syntax for `Expr`, values of type `Alt` might also be used in the representation of λ and `case` expressions.

Type Inference For Alternatives

For type inference, we begin by using `tiPats` to infer a type for each of the patterns, and to build a new list `as'` of assumptions for any bound variables, as described in Patterns.

Next, we calculate the type of the body in the scope of the bound variables, and combine this with the types of each pattern to obtain a single (function) type for the whole `Alt`.

Type Inference For Alternatives (Cont. I)

```
tiAlt                  :: Infer Alt Type
tiAlt ce as (pats, e) = do (ps, as', ts) ← tiPats pats
                           (qs,t)  ← tiExpr ce (as'++as) e
                           return (ps++qs, foldr fn t ts)
```

Type Inference For Alternatives (Cont. II)

In practice, we will often run the typechecker over a list of alternatives, `alts`, and check that the returned type in each case agrees with some known type `t`. This process can be packaged up in the following definition:

```
tiAlts :: ClassEnv → [Assump] → [Alt] → Type →
TI [Pred]
tiAlts ce as alts t = do
  psts ← mapM (tiAlt ce as) alts
  _ ← mapM (unify t) (map snd psts)
  return (concat (map fst psts))
```

Type Inference For Alternatives (Cont. III)

Although we do not need it here, the signature for `tiAlts` would allow an implementation to push the type argument inside the checking of each `Alt`, interleaving unification with type inference instead of leaving it to the end.

This is essential in extensions like the support for Rank-2 polymorphism in Hugs where explicit type information plays a key role. Even in an unextended Haskell implementation, this could still help to improve the quality of type error messages.

Of course, we can still use `tiAlts` to infer a type from scratch.

All this requires is that we generate and pass in a fresh type variable `v` in the parameter `t` to `tiAlts`, and then inspect the value of `v` under the current substitution when it returns.

From Types to Type Schemes

We have seen how lists of predicates are accumulated during type inference; now we will describe how those predicates are used to construct inferred types.

This process is sometimes referred to as *generalization* because the goal is always to calculate the most general types that are possible.

In a standard Hindley-Milner system, we can usually calculate most general types by quantifying over all relevant type variables that do not appear in the assumptions.

Applied Hindley-Milner

In this section, we will describe how the Hindley-Milner type inference algorithm can be modified to deal with the predicates in Haskell types.

Hindley-Milner Crash Course

From

https://en.wikipedia.org/wiki/Hindley–Milner_type_system

Expressions

$$
\begin{array}{rcll}
e & = & x & \text{variable} \\
 & | & e_1\, e_2 & \text{application} \\
 & | & \lambda\, x\, .\, e & \text{abstraction} \\
 & | & \texttt{let}\, x = e_1\, \texttt{in}\, e_2 &
\end{array}
$$

Types

$$
\begin{array}{rrcll}
\text{mono} & \tau & = & \alpha & \text{variable} \\
 & & | & C\, \tau \ldots \tau & \text{application} \\
\text{poly} & \sigma & = & \tau & \\
 & & | & \forall\, \alpha\, .\, \sigma & \text{quantifier}
\end{array}
$$

Hindley-Milner Crash Course (Cont. I)

Context And Typing

$$
\begin{array}{lrcl}
\text{Context} & \Gamma & = & \epsilon\,(\texttt{empty}) \\
 & & \mid & \Gamma,\, x : \sigma \\
\text{Typing} & & = & \Gamma \vdash e : \sigma
\end{array}
$$

Free Type Variables

$$
\begin{array}{lcl}
\texttt{free}(\,\alpha\,) & = & \{\alpha\} \\
\texttt{free}(\,C\,\tau_1 \ldots \tau_n\,) & = & \bigcup_{i=1}^{n} \text{free}(\,\tau_i\,) \\
\texttt{free}(\,\Gamma\,) & = & \bigcup_{x:\sigma \in \Gamma} \text{free}(\,\sigma\,)
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{free}(\,\forall\,\alpha\,.\,\sigma\,) & = & \text{free}(\,\sigma\,) - \{\alpha\} \\
\texttt{free}(\,\Gamma \vdash e : \sigma\,) & = & \text{free}(\,\sigma\,) - \text{free}(\,\Gamma\,)
\end{array}
$$

Hindley-Milner Crash Course (Cont. II)

$S\tau$ denotes applying the substitution $S = \{\ a_i \mapsto \tau_i,\ \ldots\ \}$ to type τ.

A type σ is *more general* than σ' if some quantified variable in σ is consistently substituted such that one gains σ'. This rule is formalized as:

$$\frac{\tau' = \{\alpha_i \mapsto \tau_i\}\,\tau \quad \beta_i \notin \texttt{free}(\forall\alpha_1 \ldots \forall\alpha_n.\tau)}{\forall\alpha_1 \ldots \forall\alpha_n.\tau \sqsubseteq \forall\beta_1 \ldots \forall\beta_m.\tau'}$$

The relation $\sqsubseteq$ is a *partial order* and $\forall\alpha.\alpha$ is its smallest element.

The relation $\sqsubseteq$ lifts to *substitutions* in the following manner: we say $S \subseteq T$ if and only if whenever $\alpha \mapsto \tau \in S$ and $\alpha \mapsto \tau' \in T$ then $tau \sqsubseteq \tau'$.

Hindley-Milner Crash Course (Cont. III)

The function $\texttt{mgu}(\tau, \tau')$ from **§MGU Implementation** is obeys the laws:

- $(\exists S. S\tau = S\tau') \implies \texttt{mgu}(\tau, \tau')\tau = \texttt{mgu}(\tau, \tau')\tau'$
- $\forall S. (S\tau = S\tau' \implies \texttt{mgu}(\tau, \tau') \sqsubseteq S)$

Hindley-Milner Crash Course (Cont. IV)

Here is the traditional Hindley-Milner type inference algorithm:

$$\frac{x : \sigma \in \Gamma \quad \tau = inst(\sigma)}{\Gamma \vdash_W x : \tau, \emptyset} \quad [\texttt{Var}]$$

$$\frac{\begin{array}{ll} \Gamma \vdash_W e_0 : \tau_0, S_0 & S_0\Gamma \vdash_W e_1 : \tau_1, S_1 \\ \tau' = newvar & S_2 = \texttt{mgu}(S_1\tau_0,\ \tau_1 \to \tau') \end{array}}{\Gamma \vdash_W e_0\ e_1 : S_2\tau', S_2S_1S_0} \quad [\texttt{App}]$$

$$\frac{\tau = newvar \quad \Gamma,\ x : \tau \vdash_W e : \tau', S}{\Gamma \vdash_W \lambda\, x\, .\, e : S\tau \to \tau', S} \quad [\texttt{Abs}]$$

$$\frac{\Gamma \vdash_W e_0 : \tau, S_0 \quad S_0\Gamma,\ x : \overline{S_0\Gamma}(\tau) \vdash_W e_1 : \tau', S_1}{\Gamma \vdash_W \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : \tau', S_1S_0} \quad [\texttt{Let}]$$

Hindley-Milner Applied To Haskell

To understand the basic type inference problem in Haskell, suppose that we have run the type checker over the body of a function `h` to obtain a list of predicates `ps` and a type `t`.

At this point, to obtain the most general result, we could infer a type for `h` by forming the qualified type `qt = (ps :=> t)` and then quantifying over any variables in `qt` that do not appear in the assumptions.

Hindley-Milner Applied To Haskell (Cont. I)

While this is permitted by the theory of qualified types, it is often not the best thing to do in practice. For example:

1. The list `ps` can often be simplified using the context reduction process described in Context Reduction. This will also ensure that the syntactic restrictions of Haskell are met, requiring all predicates to be in head-normal form.

2. Some of the predicates in `ps` may contain only "fixed" variables (i.e., variables appearing in the assumptions), so including those constraints in the inferred type will not be of any use. These predicates should be "deferred" to the enclosing bindings.

3. Some of the predicates in `ps` could result in *ambiguity*, and require defaulting to avoid a type error. This aspect of Haskell's type system will be described shortly in Ambiguity and Defaults.

In this paper we use a function called `split` to address these issues.

Hindley-Milner `split`

For the situation described previously where we have inferred a type `t` and a list of predicates `ps` for a function `h`, we can use `split` to rewrite and break `ps` into a pair `(ds, rs)` of deferred predicates `ds` and "retained" predicates `rs`.

The predicates in `rs` will be used to form an inferred type `(rs :=> t)` for `h`, while the predicates in `ds` will be passed out as constraints to the enclosing scope.

Hindley-Milner `split` (Cont. I)

We use the following definition for `split`:

```
split ::
      Monad m
   ⇒ ClassEnv
   → [Tyvar]
   → [Tyvar]
   → [Pred]
   → m ([Pred], [Pred])
split ce fs gs ps = do
   ps' ← reduce ce ps
   let (ds, rs) = partition (all (`elem` fs) . tv) ps'
   rs' ← defaultedPreds ce (fs ++ gs) rs
   return (ds, rs \\ rs')
```

Hindley-Milner `split` (Cont. II)

In addition to a list of predicates `ps`, the `split` function is parameterized by two lists of type variables.

The first, `fs`, specifies the set of "fixed" variables, which are just the variables appearing free in the assumptions.

The second, `gs`, specifies the set of variables over which we would like to quantify; for the example above, it would just be the variables in `(tv t \\ fs)`.

It is possible for `ps` to contain variables that are not in either `fs` or `gs` (and hence not in the parameter `(fs++gs)` that is passed to `defaultedPreds`).

In Ambiguity and Defaults we will see that this is an indication of ambiguity.

Hindley-Milner `split` (Cont. III)

There are three stages in the `split` function, corresponding directly to the three points listed previously.

1. The first stage uses `reduce` to perform context reduction.
2. The second stage uses the standard prelude function `partition` to identify the deferred predicates, `ds`; these are just the predicates in `ps'` that contain only fixed type variables.
3. The third stage determines whether any of the predicates in `rs` should be eliminated using Haskell's defaulting mechanism, and produces a list of all such predicates in `rs'`.

Hence the final set of retained predicates is produced by the expression `rs \\ rs'` in the last line of the definition.

Ambiguity and Defaults

In the terminology of Haskell , a type scheme `ps ⇒ t` is *ambiguous* if `ps` contains generic variables that do not also appear in `t`.

This condition is important because theoretical studies have shown that, in the general case, we can only guarantee a well-defined semantics for a term if its most general type is not ambiguous.

As a result, expressions with ambiguous types are considered ill-typed in Haskell and will result in a static error.

Ambiguity and Defaults: Example

The following definition shows a fairly typical example illustrating how ambiguity problems can occur:

```
stringInc x = show (read x + 1)
```

The intention here is that a string representation of a number will be parsed (using the prelude function `read`), incremented, and converted back to a string (using the prelude function `show`).

But there is a genuine ambiguity because there is nothing to specify which type of number is intended, and because different choices can lead to different semantics.

For example, `stringInc "1.5"` might produce a result of `2.5` if floating point numbers are used, or a parse error if integers are used.

Ambiguity and Defaults: Example (Cont.)

This semantic ambiguity is reflected by a syntactic ambiguity in the inferred type of `stringInc`:

```
stringInc :: (Read a, Num a, Show a) ⇒ String → String
```

A programmer can fix this particular problem quite easily by picking a particular type for `a`, and by adding an appropriate type annotation:

```
stringInc x = show (read x + 1 :: Int)
```

Ambiguity and Defaults: Haskell's Answer

In some situations involving numeric types the potential for ambiguity was significant enough to become quite a burden on programmers.

Haskell's `default` mechanism was therefore introduced as a pragmatic compromise that is convenient because it automates the task of picking types for otherwise ambiguous variables.

But it is also dangerous-because it involves making choices about the semantics of a program in ways that are not always directly visible to programmers.

For this latter reason, the use of defaulting is restricted so that it will only apply under certain, fairly restrictive circumstances.

The remainder of this section explains in more detail how ambiguities in Haskell programs can be detected and, when appropriate, eliminated by a suitable choice of defaults.

Ambiguity and Defaults & Where To Find Them

The first step is to identify any sources of ambiguity.

Suppose, for example, that we are about to qualify a type with a list of predicates `ps` and that `vs` lists all known variables, both fixed and generic. An ambiguity occurs precisely if there is a type variable that appears in `ps` but not in `vs` (i.e., in `tv ps \\ vs`).

The goal of defaulting is to bind each ambiguous type variable `v` to a monotype `t`.

The type `t` must be chosen so that all of the predicates in `ps` that involve `v` will be satisfied once `t` has been substituted for `v`.

Ambiguity and Defaults - `ambiguities`

The following function calculates the list of ambiguous variables and pairs each one with the list of predicates that must be satisfied by any choice of a default:

```
type Ambiguity = (Tyvar, [Pred])

ambiguities ::     ClassEnv
               → [Tyvar]
               → [Pred]
               → [Ambiguity]
ambiguities _ce vs ps = [ (v, filter (elem v . tv) ps)
                        | v ← tv ps \\ vs]
```

Ambiguity and Defaults According To Haskell 98

Given one of these pairs `(v,qs)`, and as specified by the Haskell 98 report, defaulting is permitted if, and only if, all of the following conditions are satisfied:

1. All of the predicates in `qs` are of the form `IsIn c (TVar v)` for some class `c`.
2. At least one of the classes involved in `qs` is a standard numeric class. The list of these class names is provided by a constant:

```
numClasses :: [Id]
numClasses  = [ "Num", "Integral", "Floating"
              , "Fractional", "Real", "RealFloat"
              , "RealFrac" ]
```

Ambiguity and Defaults According To Haskell 98 (Cont.)

3. All of the classes involved in `qs` are standard classes, defined either in the standard prelude or standard libraries. Again, the list of these class names is provided by a constant:

```
stdClasses :: [Id]
stdClasses  = ["Eq", "Ord", "Show", "Read", "Bounded"
              , "Enum", "Ix", "Functor", "Monad"
              , "MonadPlus"] ++ numClasses
```

4. That there is at least one type in the list of default types for the enclosing module that is an instance of all of the classes mentioned in `qs`. The first such type will be selected as the default. The list of default types can be obtained from a class environment by using the `defaults` function that was described in Class Environments.

Ambiguity and Defaults - `withDefaults`

These conditions are captured rather more succinctly in the following definition, which we use to calculate the candidates for resolving a particular ambiguity:

```
candidates :: ClassEnv → Ambiguity → [Type]
candidates ce (v, qs) =
  [ t'
  | let is = [i | IsIn i _t ← qs]
        ts = [t | IsIn _i t ← qs]
  , all ((TVar v) ==) ts
  , any (`elem` numClasses) is
  , all (`elem` stdClasses) is
  , t' ← defaults ce
  , all (entail ce []) [IsIn i t' | i ← is]
  ]
```

Ambiguity and Defaults - `withDefaults` (Cont. I)

If `candidates` returns an empty list for a given ambiguity, then defaulting cannot be applied to the corresponding variable, and the ambiguity cannot be avoided. On the other hand, if the result is a non-empty list `ts`, then we will be able to substitute `head ts` for `v` and remove the predicates in `qs` from `ps`. The calculations for the defaulting substitution, and for the list of predicates that it eliminates follow very similar patterns, which we capture by defining them in terms of a single, higher-order function:

```
withDefaults :: Monad m ⇒ ([Ambiguity] → [Type] → a)
                        → ClassEnv → [Tyvar] → [Pred] → m a
withDefaults f ce vs ps
    | any null tss  = fail "cannot resolve ambiguity"
    | otherwise     = return (f vps (map head tss))
      where vps = ambiguities ce vs ps
            tss = map (candidates ce) vps
```

Ambiguity and Defaults - `defaultPreds`

The `withDefaults` function takes care of picking suitable defaults, and of checking whether there are any ambiguities that cannot be eliminated.

If defaulting succeeds, then the list of predicates that can be eliminated is obtained by concatenating the predicates in each `Ambiguity` pair:

```
defaultedPreds :: Monad m
               ⇒ ClassEnv
               → [Tyvar]
               → [Pred]
               → m [Pred]
defaultedPreds =
  withDefaults (\vps _ts → concat (map snd vps))
```

Ambiguity and Defaults - `defaultSubst`

In a similar way, the defaulting substitution can be obtained by zipping the list of variables together with the list of defaults:

```
defaultSubst :: Monad m
             ⇒ ClassEnv
             → [Tyvar]
             → [Pred]
             → m Subst
defaultSubst = withDefaults (\vps ts → zip (map fst vps) ts)
```

Ambiguity and Defaults - `defaultSubst` (Cont.)

One might wonder why the defaulting substitution is useful to us here; if the ambiguous variables don't appear anywhere in the assumptions or in the inferred types, then applying this substitution to those components would have no effect.

In fact, we will only need `defaultSubst` at the top-level, when type inference for an entire module is complete.

In this case, it is possible that Haskell's infamous "monomorphism restriction" (see Implicitly Typed Bindings) may prevent generalization over some type variables.

But Haskell does not allow the types of top-level functions to contain unbound type variables. Instead, any remaining variables are considered ambiguous, even if they appear in inferred types; the substitution is needed to ensure that they are bound correctly.

Binding Groups

Our last remaining technical challenge is to describe typechecking for binding groups.

This area is neglected in most theoretical treatments of type inference, often being regarded as a simple exercise in extending basic ideas.

In Haskell, at least, nothing could be further from the truth!

With interactions between overloading, polymorphic recursion, and the mixing of both explicitly and implicitly typed bindings, this is the most complex, and most subtle component of type inference.

We will start by describing the treatment of explicitly typed bindings and implicitly typed bindings as separate cases, and then show how these can be combined.

Explicitly Typed Bindings

The simplest case is for explicitly typed bindings, each of which is described by the name of the function that is being defined, the declared type scheme, and the list of alternatives in its definition:

```
type Expl = (Id, Scheme, [Alt])
```

Haskell requires that each `Alt` in the definition of a given identifier has the same number of left-hand side arguments, but we do not need to enforce that here.

Explicitly Typed Bindings - Inference

Type inference for an explicitly typed binding is fairly easy; we need only check that the declared type is valid, and do not need to infer a type from first principles.

To support the use of polymorphic recursion,we will assume that the declared typing for `i` is already included in the assumptions.

Explicitly Typed Bindings - Inference (Cont. I)

```
tiExpl :: ClassEnv → [Assump] → Expl → TI [Pred]
tiExpl ce as (_i, sc, alts) = do
  (qs :=> t) ← freshInst sc
  ps ← tiAlts ce as alts t
  s ← getSubst
  let qs' = apply s qs
      t' = apply s t
      fs = tv (apply s as)
      gs = tv t' \\ fs
      sc' = quantify gs (qs' :=> t')
      ps' = filter (not . entail ce qs') (apply s ps)
  (ds, rs) ← split ce fs gs ps'
  if | sc /= sc' → fail "signature too general"
     | not (null rs) → fail "context too weak"
     | otherwise → return ds
```

Explicitly Typed Bindings - Inference (Cont. II)

This code begins by instantiating the declared type scheme `sc` and checking each alternative against the resulting type `t`.

When all of the alternatives have been processed, the inferred type for `i` is `qs' :=> t'`.

If the type declaration is accurate, then this should be the same, up to renaming of generic variables, as the original type `qs:=>t`.

If the type signature is too general, then the calculation of `sc'` will result in a type scheme that is more specific than `sc` and an error will be reported.

Explicitly Typed Bindings - Inference (Cont. III)

In the meantime, we must discharge any predicates that were generated while checking the list of alternatives.

Predicates that are entailed by the context `qs'` can be eliminated without further ado.

Any remaining predicates are collected in `ps'` and passed as arguments to `split` along with the appropriate sets of fixed and generic variables.

If there are any retained predicates after context reduction, then an error is reported, indicating that the declared context is too weak.

Implicitly Typed Bindings

Two complications occur when we deal with implicitly typed bindings.

The first is that we must deal with groups of mutually recursive bindings as a single unit rather than inferring types for each binding one at a time. The second is Haskell's monomorphism restriction, which restricts the use of overloading in certain cases.

A single implicitly typed binding is described by a pair containing the name of the variable and a list of alternatives:

```
type Impl   = (Id, [Alt])
```

Implicitly Typed Bindings - Monomorphism Restriction

The monomorphism restriction is invoked when one or more of the entries in a list of implicitly typed bindings is simple, meaning that it has an alternative with no left-hand side patterns. The following function provides a way to test for this:

```
restricted   :: [Impl] → Bool
restricted bindings = any simple bindings
 where simple (_i,alts) = any (null . fst) alts
```

Implicitly Typed Bindings - Inference

```
tiImpls :: Infer [Impl] [Assump]
tiImpls ce as bs = do
  ts ← mapM (\_ → newTVar Star) bs
  let (is, altss) = unzip bs
      scs         = map toScheme ts
      as'         = zipWith (:>:) is scs ++ as
  pss ← sequence (zipWith (tiAlts ce as') altss ts)
  s ← getSubst
  let ps' = apply s (concat pss)
      ts' = apply s ts
      fs = tv (apply s as)
      vss = map tv ts'
      gs = foldr1 union vss \\ fs
  (ds, rs) ← split ce fs (foldr1 intersect vss) ps'
  if restricted bs
    then let gs' = gs \\ tv rs
             scs' = map (quantify gs' . ([] :=>)) ts'
         in return (ds ++ rs, zipWith (:>:) is scs')
    else let scs' = map (quantify gs . (rs :=>)) ts'
         in return (ds, zipWith (:>:) is scs')
```

Implicitly Typed Bindings - Inference (Cont. I)

Type inference for groups of mutually recursive, implicitly typed bindings is described by `tiImpls`. It works as follows:

1. we extend `as` with assumptions binding each identifier defined in `bs` to a new type variable, and use these to type check each alternative in each binding. This is necessary to ensure that each variable is used with the same type at every occurrence within the defining list of bindings. (Lifting this restriction makes type inference undecidable.)

2. Next we use `split` to break the inferred predicates in `ps'` into a list of deferred predicates `ds` and retained predicates `rs`.

3. The list `gs` collects all the generic variables that appear in one or more of the inferred types `ts'`, but not in the list `fs` of fixed variables. Note that a different list is passed to `split`, including only variables that appear in *all* of the inferred types. This is important because all of those types will eventually be qualified by the same set of predicates, and we do not want any of the resulting type schemes to be ambiguous.

Implicitly Typed Bindings - Inference (Cont. II)

4. The final step begins with a test to see if the monomorphism restriction should be applied, and then continues to calculate an assumption containing the principal types for each of the defined values.
    - For an unrestricted binding, this is simply a matter of qualifying over the retained predicates in `rs` and quantifying over the generic variables in `gs`.
    - If the binding group is restricted, then we must defer the predicates in `rs` as well as those in `ds`, and hence we can only quantify over variables in `gs` that do not appear in `rs`.

Combined Binding Groups

Haskell requires a process of *dependency analysis* to break down complete sets of bindings-either at the top-level of a program, or within a local definition-into the smallest possible groups of mutually recursive definitions, and ordered so that no group depends on the values defined in later groups.

This is necessary to obtain the most general types possible.

For example, consider the following fragment from a standard prelude for Haskell:

```
foldr f a (x:xs) = f x (foldr f a xs)
foldr f a []     = a
and xs           = foldr (&&) True xs
```

Combined Binding Groups (Cont. I)

If the definitions `foldr` and `and` were placed in the same binding group, then we would not obtain the most general possible type for `foldr`.

All occurrences of a variable are required to have the same type at each point within the defining binding group, which would lead to the following type for `foldr`:

```
(Bool → Bool → Bool) → Bool → [Bool] → Bool
```

Combined Binding Groups (Cont. II)

To avoid this problem, we need only notice that the definition of `foldr` does not depend in any way on &&, and hence we can place the two functions in separate binding groups, inferring first the most general type for `foldr`, and then the correct type for `and`.

In the presence of explicitly typed bindings, we can refine the dependency analysis process a little further. For example, consider the following pair of bindings:

```
f :: Eq a ⇒ a → Bool
f x  = (x==x) || g True
g y  = (y≤y) || f True
```

Combined Binding Groups (Cont. III)

Although these bindings are mutually recursive, we do not need to infer types for `f` and `g` at the same time. Instead, we can use the declared type of `f` to infer a type:

```
g    :: Ord a ⇒ a → Bool
```

and then use this to check the body of `f`, ensuring that its declared type is correct.

Combined Binding Groups - Representation

Motivated by these observations, we will represent Haskell binding groups using the following datatype:

```
type BindGroup  = ([Expl], [[Impl]])
```

The first component in each such pair lists any explicitly typed bindings in the group.

The second component provides an opportunity to break down the list of any implicitly typed bindings into several smaller lists, arranged in dependency order.

Combined Binding Groups - Representation (Cont. I)

In other words, if a binding group is represented by a pair `(es,[is_1, ... ,is_n])`, then the implicitly typed bindings in each `is_i` should depend only on the bindings in `es, is_1, ... , is_i`, and not on any bindings in `is_j` when `j>i`.

- Bindings in `es` could depend on any of the bindings in the group, but will presumably depend on at least those in `is_n`, or else the group would not be minimal.
- Note also that if `es` is empty, then `n` must be `1`.

Combined Binding Groups - Representation (Cont. II)

In choosing this representation, we have assumed that dependency analysis has been carried out prior to type checking, and that the bindings in each group have been organized into values of type `BindGroup` as appropriate.

By separating out implicitly typed bindings as much as possible, we can potentially increase the degree of polymorphism in inferred types.

For a correct implementation of the semantics specified in the Haskell report, a simpler but less flexible approach is required: all implicitly typed bindings must be placed in a single list, even if a more refined decomposition would be possible.

Combined Binding Groups - Representation (Cont. III)

In addition, if the group is restricted, then we must also ensure that none of the explicitly typed bindings in the same `BindGroup` have any predicates in their type, even though this is not strictly necessary.

With hindsight, these are restrictions that we might prefer to avoid in any future revision of Haskell.

Combined Binding Groups - Haskell 98 Ambiguity

A more serious concern is that the Haskell 98 report does not indicate clearly whether the previous example defining `f` and `g` should be valid.

At the time of writing, some implementations accept it, while others do not.

Combined Binding Groups - Haskell 98 Ambiguity (Cont. I)

This is exactly the kind of problem that can occur when there is no precise, formal specification!

Curiously, however, the report does indicate that a modification of the example to include an explicit type for `g` would be illegal.

This is a consequence of a throw-away comment specifying that all explicit type signatures in a binding group must have the same context up to renaming of variables[1].

[1] See §4.5.2 of the Haskell 98 report

Combined Binding Groups - Haskell 98 Ambiguity (Cont. II)

This "α-equivalent context" syntactic restriction can easily be checked prior to type checking.

Our comments here, however, suggest that it is unnecessarily restrictive.

In addition to the function bindings that we have seen already, Haskell allows variables to be defined using pattern bindings of the form `pat = expr`.

We do not need to deal directly with such bindings because they are easily translated into the simpler framework used in this paper.

Combined Binding Groups - Haskell 98 Ambiguity (Cont. III)

For example, a binding:

```
(x,y) = expr
```

can be rewritten as:

```
nv = expr
x  = fst nv
y  = snd nv
```

where `nv` is a new variable.

Combined Binding Groups - Haskell 98 Ambiguity (Cont. IV)

The precise definition of the monomorphism restriction in Haskell makes specific reference to pattern bindings, treating any binding group that includes one as restricted.

So it may seem that the definition of restricted binding groups in this paper is not quite accurate.

However, if we use translations as suggested here, then it turns out to be equivalent: even if the programmer supplies explicit type signatures for x and y in the original program, the translation will still contain an implicitly typed binding for the new variable nv.

Combined Binding Groups - Implementation

Now, at last, we are ready to present the algorithm for type inference of a complete binding group, as implemented by the following function:

```
tiBindGroup :: Infer BindGroup [Assump]
tiBindGroup ce as (es,iss) =
  do let as' = [ v:>:sc | (v,sc,_alts) ← es ]
     (ps, as'') ← tiSeq tiImpls ce (as'++as) iss
     qss        ← mapM (tiExpl ce (as''++as'++as)) es
     return (ps++concat qss, as''++as')
```

Combined Binding Groups - Implementation (Cont. I)

The structure of this definition is quite straightforward.

1. We form a list of assumptions `as'` for each of the explicitly typed bindings in the group.
2. We use this to check each group of implicitly typed bindings, extending the assumption set further at each stage.
3. Finally, we return to the explicitly typed bindings to verify that each of the declared types is acceptable.

Combined Binding Groups - Implementation (Cont. I)

In dealing with the list of implicitly typed binding groups, we use the following utility function, which typechecks a list of binding groups and accumulates assumptions as it runs through the list:

```
tiSeq :: Infer bg [Assump] → Infer [bg] [Assump]
tiSeq _ti _ce _as [] = return ([], [])
tiSeq ti ce as (bs:bss) = do
  (ps, as') ← ti ce as bs
  (qs, as'') ← tiSeq ti ce (as' ++ as) bss
  return (ps ++ qs, as'' ++ as')
```

Top-level Binding Groups

At the top-level, a Haskell program can be thought of as a list of binding groups:

```
type Program = [BindGroup]
```

Even the definitions of member functions in class and instance declarations can be included in this representation; they can be translated into top-level, explicitly typed bindings.

Top-level Binding Groups - Implementation

The type inference process for a program takes a list of assumptions giving the types of any primitives, and returns a set of assumptions for any variables.

```
tiProgram :: ClassEnv → [Assump] → Program → [Assump]
tiProgram ce as bgs =
  runTI $ do
    (ps, as') ← tiSeq tiBindGroup ce as bgs
    s ← getSubst
    rs ← reduce ce (apply s ps)
    s' ← defaultSubst ce [] rs
    return (apply (s' @@ s) as')
```

This completes our presentation of the Haskell type system.

Conclusions

We have presented a complete Haskell program that implements a type checker for the Haskell language. In the process, we have clarified certain aspects of the current design, as well as identifying some ambiguities in the existing, informal specification.

The type checker has been developed, type-checked, and tested using the "Haskell 98 mode" of Hugs 98.

The full program includes many additional functions, not shown in this paper, to ease the task of testing, debugging, and displaying results.

Final Word

We have also translated several large Haskell programs-including the Standard Prelude, the Maybe and List libraries, and the source code for the type checker itself-into the representations described in Type Inference, and successfully passed these through the type checker.

As a result of these and other experiments we have good evidence that the type checker is working as intended, and in accordance with the expectations of Haskell programmers.

We believe that this typechecker can play a useful role, both as a formal specification for the Haskell type system, and as a testbed for experimenting with future extensions.