Implementing Dependent Types in Pi-Forall

Stephanie Weirich

May 17, 2022

1 Goals and What to Expect

Over the next four lectures, I will describe a small dependently-typed language called "pi-forall" and walk through the implementation of its type checker.

What do I want you to get out of these lectures?

1. An understanding of how to translate mathematical specifications of type systems and logics into implementations, i.e. how to represent the syntax and implement a type checker. More generally, how to turn a declarative specification of a system of judgments into an algorithm.

2. Exposure to the issues in implmenting dependently-typed languages. Because there are only four lectures, my goal is breadth not depth. As a result, I will provide you with *simple* solutions to some of the problems you might face and sidestep other problems entirely. Overally, the solutions you see here will not be the best solution, but I will give you pointers if you want to go deeper.

3. Exposure to the Haskell programming language. I think Haskell is an awesome tool for this sort of work and, if there is an advanced feature that exactly addresses our design goal (e.g. monads, generic programming, laziness) I want to show you how that can work.

4. A tool that you can use as a basis for experimentation. How do you know what programs you can and cannot express in your new type system? Having an implementation around lets you work out (smallish) examples and will help to convince you (and your reviewers) that you are developing something useful.

5. Templates for writing about programming languages. The source files for these lecture notes are available.

Questions should you be thinking about during these sessions:

- How to represent the abstract syntax of the language, including variable binding?
- How much information do we need to include in terms to make type checking algorithmic?
- How can we include less?
- How to decide when types (and terms) are equal?
- How to decide what parts of the term are irrelevant during computation? Can be ignored when checking for equality?

Tools for typesetting Ott ottalt.sty mathpartir listings

2 A Simple Core language with Type-in-Type

Let's consider a simple dependently-typed lambda calculus. What should it contain? At the bare minimum we can start with the following five forms:

a, A	$::=$	x	variables
		$\lambda x.a$	lambda expressions (anonymous functions)
		$a\ b$	function applications
		$(x : A) \to B$	dependent function type, aka Π
		Type	the 'type' of types

Note that we are using the *same* syntax for expressions and types. For clarity, I'll used lowercase letters a for expressions and uppercase letters for their types A.

Note that λ and Π above are *binding forms*. They bind the variable x in a and B respectively.

2.1 When do expressions in this language type check?

We define the type system for this language using an inductively defined relation. This relation is between an expression, its type, and a typing context.

$$\boxed{\Gamma \vdash a : A}$$

The typing context is an ordered list of assumptions about the types of variables.

$$\Gamma ::= \emptyset \mid \Gamma, x : A$$

An initial set of typing rules: Variables and Functions If we know a variable's type because it is in the typing context, then that is its type:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \quad \text{T\_VAR}$$

Variables are introduced into the context when we type check abstractions.

$$\frac{\Gamma, x : A \vdash a : B}{\Gamma \vdash \lambda x.a : (x : A) \to B} \quad \text{T\_SLAMBDA}$$

Example: Polymorphic identity functions Note that the variable x is allowed to appear in B. Why is this useful? Well it gives us *parametric polymorphism* right off the bat. In Haskell, we write the identity function as follows:

```
id :: a -> a
id x = x
```

and Haskell automatically generalizes it to work for *all* types. We can do that here, except that we need to explicitly use lambda to make this function polymorphic. Instead of Haskell's

```
forall a. a -> a
```

we will write the type of the polymorphic identity function as

$(\texttt{x:Type}) \to (\texttt{y} : \texttt{x}) \to \texttt{x}$

The fact that the type of x is **Type** means that x is a type variable. Again, in this language we don't have a syntactic distinction between types and terms (or expressions). Types are anything of type **Type**.

Expressions are things of type A where A has type **Type**.

$$\dfrac{\dfrac{\dfrac{}{x:\mathbf{Type}, y:x \vdash y : x}\ \text{VAR}}{x:\mathbf{Type} \vdash \lambda y.y : (y:x) \to x}\ \text{SLAMBDA}}{\vdash \lambda x.\lambda y.y : (x:\mathbf{Type}) \to (y:x) \to x}\ \text{SLAMBDA}$$

In pi-forall, we should eventually be able to write

```
id :  (x:Type) → (y : x) → x
id = λx. λy. y
```

or even (with some help from the parser)

```
id :  (x:Type) → x → x
id = λx y . y
```

More typing rules: Types Actually, I lied. The real typing rule that we want for lambda has an additional precondition. We need to make sure that when we add assumptions to the context, those assumptions really are types. Otherwise, the rules would allow us to derive this type for the polymorphic lambda calculus:

$$\vdash \lambda x.\lambda y.y : (x:\mathbf{True}) \to (y:x) \to x$$

So the real rule has an extra precondition that checks to make sure that A is actually a type.

$$\dfrac{\begin{array}{c}\Gamma, x:A \vdash a : B \\ \Gamma \vdash a : \mathbf{Type}\end{array}}{\Gamma \vdash \lambda x.a : (x:A) \to B}\ \text{T\_LAMBDA}$$

This precondition means that we need some rules that conclude that types are actually types. For example, the type of a function is a type, so we will declare it with this rule (which also ensures that the domain and range of the function are also types).

$$\dfrac{\begin{array}{c}\Gamma \vdash A : \mathbf{Type} \\ \Gamma, x:A \vdash B : \mathbf{Type}\end{array}}{\Gamma \vdash (x:A) \to B : \mathbf{Type}}\ \text{T\_PI}$$

Likewise, for polymorphism we need this, rather perplexing rule:

$$\dfrac{}{\Gamma \vdash \mathbf{Type} : \mathbf{Type}}\ \text{T\_TYPE}$$

Because the type of the polymorphic identity function starts with $(x:\mathbf{Type}) \to \ldots$ the rule T-PI rule means that **Type** must be a type for this pi type to make sense. We declare this by fiat using the rule T-TYPE rule.

Note that, sadly, this rule make our language inconsistent as a logic. cf. Girard's paradox.

More typing rules: Application Application requires that the type of the argument matches the domain type of the function. However, note that because the type 'B' could have x free in it, we need to substitute the argument for x in the result.

$$\dfrac{\begin{array}{c}\Gamma \vdash a : (x:A) \to B \\ \Gamma \vdash b : A\end{array}}{\Gamma \vdash a\ b : B\{b/x\}}\ \text{T\_APP}$$

Example: applying the polymorphic identity function In pi-forall we should be able to apply the polymorphic identity function to itself. When we do this, we need to first provide the type of 'id', then we can apply 'id' to 'id'.

```
idid : (x:Type) → (y : x) → x
idid = id ((x:Type) → (y : x) → x) id
```

Example: Church booleans Because we have (impredicative) polymorphism, we can *encode* familiar types, such as booleans. The idea behind this encoding is to represent terms by their eliminators. In other words, what is important about the value true? The fact that when you get two choices, you pick the first one. Likewise, false "means" that with the same two choices, you should pick the second one. With parametric polymorphism, we can give the two terms the same type, which we'll call bool.

```
bool : Type
bool = (x : Type) → x → x → x

true : bool
true = λx. λy. λz. y

false : bool
false = λx. λy. λz. z
```

Thus, a conditional expression just takes a boolean and returns it.

```
cond : bool → (x:Type) → x → x → x
cond = λb. b
```

Example: logical "and" (i.e. product types) We can also encode a logical "and" data structure.

```
and : Type → Type → Type
and = λp. λq. (c: Type) → (p → q → c) → c

conj : (p:Type) → (q:Type) → p → q → and p q
conj = λp.λq. λx.λy. λc. λf. f x y

proj1 : (p:Type) → (q:Type) → and p q → p
proj1 = λp. λq. λa. a p (λx.λy.x)

proj2 : (p:Type) → (q:Type) → and p q → q
proj2 = λp. λq. λa. a q (λx.λy.y)

and_commutes : (p:Type) → (q:Type) → and p q → and q p
and_commutes = λp. λq. λa. conj q p (proj2 p q a) (proj1 p q a)
```

3 From typing rules to a typing algorithm

So the rules that we have developed so far are great for saying *what* terms should type check, but they don't say *how*. In particular, we've developed these rules without thinking about how we would implement them.

A type system is called *syntax-directed* if it is readily apparent how to turn the typing rules into code. In other words, we would like to implement the following function (in Haskell), that when given a term and a typing context produces the type of the term (if it exists).

```
inferType :: Term -> Ctx -> Maybe Type
```

Let's look at our rules. Is this straightforward? For example, for the variable rule as long as we can lookup the type of a variable in the context, we can produce its type.

```
inferType (Var x) ctx = Just ty when
  ty = lookupTy ctx x
```

Likewise, the case of the typing function for the **Type** term is straightforward.

```
inferType Type ctx = Just Type
```

The only stumbling block for the algorithm is the lambda rule. To type check a function, we need to type check its body when the context has been extended with the type of the argument. But, the type of the argument A is not annotated on the lambda. So where does it come from?

There is actually an easy fix to turn our current system into an algorithmic one. We just annotate lambdas with the types of the abstracted variables. But perhaps this is not what we want to do.

Look at our example code: the only types that we wrote were the types of definitions. It is good style to do that, and wherever we define a function, we can look at those types to know what the types of the argument should be. So, maybe if we change our point of view, we can get away without annotating lambdas with those argument types.

3.1 A bidirectional type system

Let's redefine the system using two judgments. The first one is similar to the judgement that we saw above, and we will call it type *inference*. This judgement will be paired with (and will depend on) a second judgement, called type *checking*, that takes advantage of known type information, such as the annotations on top-level definitions.

We will express these judgements using the following notation and implement them in Haskell using the following mutually-recursive functions. Furthermore, to keep track of which rule is in which judgement, rules that have inference as a conclusion will start with I- and rules that have checking as a conclusion will start with C-.

$\boxed{\Gamma \vdash a \Rightarrow A}$ `inferType` in context Γ, infer that term a has type A

$\boxed{\Gamma \vdash a \Leftarrow A}$ `checkType` in context Γ, check that term a has type A

Let's go back to some of our existing rules. For variables, we can just change the colon to an inference arrow. The context tells us the type to infer.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \text{ I\_VAR}$$

On the other hand, we should check lambda expressions against a known type. If that type is provided, we can propagate it to the body of the lambda expression. We also know that we want A to be a **Type**.

$$\frac{\Gamma, x : A \vdash a \Leftarrow B \quad \Gamma \vdash A \Leftarrow \mathbf{Type}}{\Gamma \vdash \lambda x.a \Leftarrow (x : A) \to B} \text{ C\_LAMBDA}$$

Applications can be in inference mode (in fact, checking mode doesn't help). Here, we must infer the type of the function, but once we have that type, we may use it to check the type of the argument.

$$\frac{\Gamma \vdash a \Rightarrow (x : A) \to B \quad \Gamma \vdash b \Leftarrow A}{\Gamma \vdash a\ b \Rightarrow B\{b/x\}} \text{ I\_APP}$$

For types, it is apparent what their type is, so we will just continue to infer that.

$$\frac{\Gamma \vdash A \Leftarrow \mathbf{Type} \quad \Gamma, x : A \vdash B \Leftarrow \mathbf{Type}}{\Gamma \vdash (x : A) \to B \Rightarrow \mathbf{Type}} \text{ I\_PI}$$

$$\frac{}{\Gamma \vdash \mathbf{Type} \Rightarrow \mathbf{Type}} \text{ I\_TYPE}$$

Notice that this system is incomplete. There are inference rules for every form of expression except for lambda. On the other hand, only lambda expressions can be checked against types. We can make the checking judgement more applicable by including the following rule

$$\frac{\Gamma \vdash a \Rightarrow A}{\Gamma \vdash a \Leftarrow A} \text{ C\_INFER}$$

that allows us to use inference whenever a checking rule doesn't apply.

Now, let's think about the reverse problem a bit. There are programs that the checking system won't admit but would have been acceptable by our first system. What do they look like?

Well, they involve applications of explicit lambda terms:

$$\frac{\vdash \lambda x.x : \mathbf{Bool} \to \mathbf{Bool} \qquad \vdash \mathit{true} : \mathbf{Bool}}{\vdash (\lambda x.x)\ \mathbf{True} : \mathbf{Bool}} \text{ T-APP}$$

This term doesn't type check in the bidirectional system because application requires the function to have an inferable type, but lambdas don't. However, there is not that much need to write such terms. We can always replace them with something equivalent by doing a β-reduction of the application (in this case, just replace the term with **True**).

In fact, the bidirectional type system has the property that it only checks terms in *normal* form, i.e. those that do not contain any β-reductions. If we would like to add non-normal forms to our language, we can add a typing rule for type annotations:

$$\frac{\Gamma \vdash a \Leftarrow A}{\Gamma \vdash (a : A) \Rightarrow A} \text{ I\_ANNOT}$$

Type annotations allow us to supply known type information anywhere within a term.

$$\frac{\vdash (\lambda x.x : \mathbf{Bool} \to \mathbf{Bool}) \Rightarrow \mathbf{Bool} \to \mathbf{Bool} \qquad \vdash \mathit{true} \Leftarrow \mathbf{Bool}}{\vdash (\lambda x.x : \mathbf{Bool} \to \mathbf{Bool})\ \mathbf{True} \Rightarrow \mathbf{Bool}} \text{ I-APP}$$

The nice thing about the bidirectional system is that it reduces the number of annotations that are necessary in programs that we want to write. As we will see, checking mode will be even more important as we add more terms to the language.

A not so desirable property of this birdirectional system is that it is not closed under substitution. The types of variables are always inferred. This is particularly annoying in an application, rule when we replace a variable (inference mode) with another term that is correct only in checking mode. One solution to this problem is to work with *hereditary substitutions*, i.e. substitutions that preserve normal forms.

Alternatively, we can solve the problem through *elaboration*, the output of a type checker will be a term that works purely in inference mode.

4 Putting it all together in a Haskell implementation

Last time, we defined a bidirectional type system for a small core language. Today we'll start talking about what the implementation of this language might look like in Haskell.

First, an overview of the main files of the implementation.

```
Syntax.hs      - specification of the abstract syntax of the language (AST)
Parser.hs      - turn strings into AST
PrettyPrint.hs - displays AST in a (somewhat) readable form
Main.hs        - top-level routines (repl)

Environment.hs - defines the type checking monad
TypeCheck.hs   - implementation of the bidirectional type checker
```

4.0.1 Variable binding using the unbound library [Syntax.hs]

One difficulty with implementing the lambda calculus is the treatment of variable binding. Functions (λ-expressions) and their types (Π-types) *bind* variables. In the implementation of our type checker, we'll need to be able to determine whether two terms are *alpha-equivalent*, calculate the *free variables* of a term, and perform *capture-avoiding substitution.* When we work with a lambda expression, we will want to be sure that the binding variable is *fresh*, that is, distinct from all other variables in the program.

In today's code, we'll use the unbound library to get all of these operations for free. This library defines a type for variable names, called `Name`.

```
type TName = Name Term
```

This type is indexed by the type of AST that this is a name for. That way Unbound can make sure that substitutions make sense.

```
class Subst b a where
   subst  :: Name b -> b -> a -> a
```

The `subst` function in this class ensures that when we see `subst x a b`, which means "substitute a for x in b" (also written $b\{a/x\}$ above) that `a` is the right sort of thing to stick in for `x`. The Unbound library can automatically generate instances of the `Subst` class. Furthermore, although it seems like we only need to substitute within terms, we'll actually need to have substitution available at many types.

With names, we can define the syntax that corresponds to our language above, using the following datatype.

```
data Term =
     Type                                -- ^ universe
   | Var TName                           -- ^ variables
   | Lam (Bind TName, Embed Annot) Term)
                                         -- ^ abstraction
   | App Term Term                       -- ^ application
   | Pi (Bind (TName, Embed Term) Term) -- ^ function type
```

As you can see, variables are represented by names. The `Bind` type constructor declares the scope of the bound variables. Both `Lam` and `Pi` bind a single variable in a `Term`. The `Annot` type is an optional type annotation:

```
newtype Annot = Annot (Maybe Type) deriving Show
```

and, because the syntax is all shared, a `Type` is just another name for a `Term`. We'll use this name just for documentation.

```
type Type = Term
```

The fact that this annotation is optional means that we'll be able to use a single datatype for both the versions of the language (the one where lambdas are annotated and the one where they aren't). We'll start with an expression that has no annotations on lambdas, and elaborate it to one that does.

The definitions of the AST datatypes derive instances for two type classes from the `unbound-generics` library: `Unbound.Alpha` and `Unbound.Subst`.

Among other things, the `Alpha` class enables functions for alpha equivalence and free variable calculation, with the types shown below. Because unbound creates these instances for us, we don't have to worry about defining them.

```
aeq :: Alpha a => a -> a -> Bool
fv  :: Alpha a => a -> [Name a]
```

Creating an instance of the `Subst` type class requires telling unbound where the variables are (and no more):

```
instance Subst Term Term where
  isvar (Var x) = Just (SubstName x)
  isvar _ = Nothing
```

We also need to be able to substitute terms through annotations, but annotations don't contain free variables directly, they only have them within the terms inside them.

For more information about unbound, see the unbound-generics hackage page.

4.0.2 A TypeChecking monad [Environment.hs]

Recall that our plan is to write two mutually recursive functions for type checking of the following types:

```
inferType :: Term -> Ctx -> Maybe (Term,Type)

checkType :: Term -> Type -> Ctx -> Maybe Term
```

The inference function should take a term and a context and if it type checks, produce its type and its elaboration (where all annotations have been filled in). The checking function should take a term and a context and a type, and if that term has that type produce an elaborated version (where all of the annotations have been filled in.)

Well actually, we'll do something a bit different. We'll define a *type checking monad*, called `TcMonad` that will handle the plumbing for the typing context, and allow us to return more information than `Nothing` when a program doesn't type check.

```
inferType :: Term -> TcMonad (Term,Type)

checkType :: Term -> Type -> TcMonad Term
```

Those of you who have worked with Haskell before may be familiar with the MonadReader, and the MonadError, which our type checking monad will be instances of.

```
lookupTy :: TName -> TcMonad Term
extendCtx :: Decl -> TcMonad Term -> TcMonad Term


err  :: (Disp a) => a -> TcMonad b
warn :: (Disp a) => a -> TcMonad b
```

We'll also need this monad to be a freshness monad, to support working with binding structure, and throw in MonadIO for good measure.

4.0.3 Implementing the TypeChecking Algorithm [Typecheck.hs]

Now that we have the type checking monad available, we can start our implementation. For flexibility `inferType` and `checkType` will *both* be implemented by the same function:

```
inferType :: Term -> TcMonad (Term,Type)
inferType t = tcTerm t Nothing

checkType :: Term -> Type -> TcMonad (Term, Type)
checkType tm ty = tcTerm tm (Just ty)
```

The `tcTerm` function checks a term, producing an elaborated term where all of the type annotations have been filled in, and its type. The second argument is `Nothing` in inference mode and an expected type in checking mode.

```
tcTerm :: Term -> Maybe Type -> TcMonad (Term,Type)
```

The general structure of this function starts with a pattern match for the various syntactic forms in inference mode:

```
tcTerm (Var x) Nothing = ...

tcTerm Type Nothing = ...

tcTerm (Pi bnd) Nothing = ...

tcTerm (Lam bnd) Nothing = ... -- must have annotation

tcTerm (App t1 t2) Nothing = ...
```

Mixed in here, we also have a pattern for lambda expressions in checking mode:

```
tcTerm (Lam bnd) (Just (Pi bnd2)) = ...

tcTerm (Lam _) (Just nf) =  -- checking mode wrong type
   err [DS "Lambda expression has a function type, not", DD nf]
```

There are also several cases for practical reasons (annotations, source code positions, etc.) and a few cases for homework.

Finally, the last case covers all other forms of checking mode, by calling inference mode and making sure that the inferred type is equal to the checked type. This case is the implementation of rule C-INFER.

```
tcTerm tm (Just ty) = do
 (atm, ty') <- inferType tm
   unless (aeq ty' ty) $ err [DS "Types don't match", DD ty, DS "and", DD ty']
 return (atm, ty)
```

The function `aeq` merely ensures that the two types are alpha-equivalent. If they are, then it returns `()` to the monad, otherwise it throws an error.

4.0.4 Example

The file Lec1.pi contains the examples that we worked out in lecture last time. Let's try to type check it, after filling in the missing code in `TypeCheck.hs`.

4.0.5 Exercise (Type Theory & Haskell) - Add Booleans and Sigma types

Some fairly standard typing rules for booleans assert that Bool is a valid type:

$$\frac{}{\Gamma \vdash \mathbf{Bool} : \mathbf{Type}} \text{ T\_BOOL}$$

$$\frac{}{\Gamma \vdash \mathbf{True} : \mathbf{Bool}} \text{ T\_TRUE}$$

$$\frac{}{\Gamma \vdash \mathbf{False} : \mathbf{Bool}} \text{ T\_FALSE}$$

$$\frac{\begin{array}{c}\Gamma \vdash a : \mathbf{Bool} \\ \Gamma \vdash b_1 : A \\ \Gamma \vdash b_2 : A\end{array}}{\vdash \mathbf{if}\ a\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 : A} \text{ T\_IF}$$

Likewise, we can also extend the language with Sigma types.

$$\frac{\begin{array}{c}\Gamma \vdash A : \mathbf{Type} \\ \Gamma, x : A \vdash B : \mathbf{Type}\end{array}}{\Gamma \vdash \{x : A \mid B\} : \mathbf{Type}} \text{ T\_SIGMA}$$

A sigma type is a product where the type of the second component of the product can depend on the first component.

$$\frac{\begin{array}{c}\Gamma \vdash a : A \\ \Gamma \vdash b : B\{a/x\}\end{array}}{\Gamma \vdash (a, b) : \{x : A \mid B\}} \text{ T\_PAIR}$$

We destruct sigmas using pattern matching. A simple rule for pattern matching introduces variables into the context when pattern matching the sigma type. These variables are not allowed to appear free in the result type of the pattern match.

$$\frac{\begin{array}{l}\Gamma \vdash a : \{x : A_1 \mid A_2\} \\ \Gamma, x : A_1, y : A_2 \vdash b : B \\ \Gamma \vdash B : \mathbf{Type}\end{array}}{\Gamma \vdash \mathbf{let}\ (x, y) = a\ \mathbf{in}\ b : B} \text{ T\_LETPAIR\_WEAK}$$

This part of the homework has two parts:

1. First: rewrite the rules above in bidirectional style. Which rules should be inference rules? Which ones should be checking rules? If you are familiar with other systems, how do these rules compare?

2. In Haskell, later: The code in `version1/` includes abstract and concrete syntax for booleans and sigma types. The pi-forall file `version1/test/Hw1.pi` contains examples of using these new forms. However, to get this file to compile, you'll need to fill in the missing cases in `version1/src/TypeCheck.hs`.

5 Equality in Dependently-Typed Languages

You may have noticed in the previous lecture that there was something missing. Most of the examples that we did could have also been written in System F (or something similar)!

Today we are going to think about how type equality can make our language more expressive. We will do this in two steps: adding both definitional and propositional equality to the language.

5.1 Definitional equality

5.1.1 Motivating Example: Type level reduction

In full dependently-typed languages (and in full pi-forall) we can see the need for definitional equality. We want to equate types that are not just *syntactically* equal, so that more expressions type check.

We saw yesterday an example where we wanted a definition of equality that was more expressive than alpha-equivalence. Recall our encoding for the logical `and` proposition:

```
and : Type -> Type -> Type
and = \p. \q. (c: Type) -> (p -> q -> c) -> c
```

Unfortunately, our definition of `conj` still doesn't type check:

```
conj : (p:Type) -> (q:Type) -> p -> q -> and p q
conj = \p.\q. \x.\y. \c. \f. f x y
```

Running this example with `version1` of the type checker produces the following error:

```
Checking module "Lec1"
Type Error:
../test/Lec1.pi:34:22:
    Function a should have a function type. Instead has type and p q
    When checking the term
       \p . \q . \a . a p ((\x . \y . x))
    against the signature
       (p : Type) -> (q : Type) -> (and p q) -> p
    In the expression
       a p ((\x . \y . x))
```

The problem is that even though we want `and p q` to be equal to the type `(c: Type) -> (p -> q -> c) -> c` the typechecker does not treat these types as equal.

Note that the type checker already records in the environment that `and` is defined as `\p.\q. (c: Type) -> (p -> q -> c) -> c`. We'd like the type checker to look up this definition when it sees the variable `and` and beta-reduce th this application.

5.1.2 Another example needing more expressive equality

As another example, in the full language, we might have a type of length indexed vectors, where vectors containing values of type `A` with length `n` can be given the type `Vec A n`. In this language we may have a safe head operation, that allows us to access the first element of the vector, as long as it is nonzero.

```
head : (A : Nat) -> (n : Nat) -> Vec A (succ n) -> Vec A n
head = ...
```

However, to call this function, we need to be able to show that the length of the argument vector is equal to `succ n` for some n. This is ok if we know the length of the vector outright

```
v1 : Vec Bool (succ 0)
v1 = VCons True VNil
```

So the application `head Bool 0 v1` will type check. (Note that pi-forall cannot infer the types `A` and `n`.)

However, if we construct the vector, its length may not be a literal natural number:

```
append : (n : Nat) -> (m : Nat) -> Vec A m -> Vec A n -> Vec A (plus m n)
append = ...
```

In that case, to get `head Bool 1 (append v1 v1)` to type check, we need to show that the type `Vec Bool (succ 1)` is equal to the type `Vec Bool (plus 1 1)`. If our definition of type equality is *alpha-equivalence*, then this equality will not hold. We need to enrich our definition of equality so that it equates more terms.

5.1.3 Defining definitional equality

The main idea is that we will:

- establish a new judgement that defines when types are equal

  ```
  G |- A = B
  ```

- add the following rule to our type system so that it works "up-to" our defined notion of type equivalence

  ```
  G |- a : A    G |- A = B
  ------------------------- conv
          G |- a : B
  ```

- Figure out how to revise the *algorithmic* version of our type system so that it supports the above rule.

What is a good definition of equality? We started with a very simple one: alpha-equivalence. But we can do better:

We'd like to make sure that our relation *contains beta-equivalence*:

```
-------------------------- beta
G |- (\x.a)b = a {b / x}
```

(with similar rules for if/sigmas if we have them.)

Is an *equivalence relation*:

```
----------  refl
G |- A = A

G |- A = B
-----------  sym
G |- B = A

G |- A = B    G |- B = C
------------------------ trans
G |- A = C
```

and a *congruence relation* (i.e. if subterms are equal, then larger terms are equal):

```
G |- A1 = A2       G,x:A1 |- B1 = B2
------------------------------------ pi
G |- (x:A1) -> B1 = (x:A2) -> B2


G,x:A1 |- b1 = b2
```

```
 ------------------- lam
 G |- \x.b1 = \x.b2


 G |- a1 = a2    G |- b1 b2
 -------------------------- app
 G |- a1 b1 = a2 b2

[similar rules for if and sigmas]
```

that has "functionality" (i.e. we can lift equalities over `b`):

```
G, x : A |- b : B    G |- a1 == a2
---------------------------------
G |- b{a1 / x} = b{a2 / x}
```

5.1.4 Using definitional equality in the algorithm

We would like to consider our type system as having the following rule:

```
 G |- a : A    G |- A = B
 ----------------------- conv
 G |- a : B
```

But that rule is not syntax directed. Where do we need to add equality preconditions in our bidirectional system? It turns out that there are only a few places.

- Where we switch from checking mode to inference mode in the algorithm. Here we need to ensure that the type that we infer is the same as the type that is passed to the checker.

  ```
  G |- a => A    G |- A = B
  -------------------------- :: infer
  G |- a <= B
  ```

- In the rule for application, when we infer the type of the function we need to make sure that the function actually has a function type. But we don't really know what the domain and co-domain of the function should be. We'd like our algorithm for type equality to be able to figure this out for us.

  ```
  G |- a => A    A ?=> (x:A1) -> A2
  G |- b <= A1
  ---------------------------------- app
         G |- a b => A2 { b / x }
  ```

5.2 Using definitional equality

The rules above *specify* when terms should be equal, but they are not an algorithm. We actually need several different functions. First,

```
equate :: Term -> Term -> TcMonad ()
```

ensures that the two provided types are equal, or throws a type error if they are not. This function corresponds directly to our definition of type equality.

Second, we also need to be able to determine whether a given type is equal to some "head" form, without knowing exactly what that form is. For example, when *checking* lambda expressions, we need to know

that the provided type is of the form of a pi type (`(x:A) -> B`). Likewise, when inferring the type of an application, we need to know that the type inferred for the function is actually a pi type.

We can determine this in two ways. Most directly, the function

```
ensurePi :: Type -> TcMonad (TName, Type, Type)
```

checks the given type to see if it is equal to some pi type of the form `(x:A1) -> A2`, and if so returns `x`, `A1` and `A2`.
This function is defined in terms of a helper function:

```
whnf :: Term -> TcMonad Term
```

that reduces a type to its *weak-head normal form*. Such terms have done all of the reductions to the outermost lambda abstraction (or pi) but do not reduce subterms. In other words:

```
(\x.x) (\x.x)
```

is not in whnf, because there is more reduction to go to get to the head. On the other hand, even though there are still internal reductions possible:

```
\y. (\x.x) (\x.x)
```

and

```
(y:Type) -> (\x.x)Bool
```

are in weak head normal form. Likewise, the term `x y` is also in weak head normal form (if we don't have a definition available for `x`) because, even though we don't know what the head form is, we cannot reduce the term any more.

In `version2` of the the implementation, these functions are called in a few places: - `equate` is called at the end of `tcTerm` - `ensurePi` is called in the `App` case of `tcTerm` - `whnf` is called in `checkType`, before the call to `tcTerm` to make sure that we are using the head form in checking mode.

5.3 Implementing definitional equality (see Equal.hs)

There are several ways for implementing definitional equality, as stated via the rules above. The easiest one to explain is based on reduction—for `equate` to reduce the two arguments to some normal form and then compare those normal forms for equivalence.

One way to do this is with the following algorithm:

```
equate t1 t2 = do
   nf1 <- reduce t1
   nf2 <- reduce t2
   aeq nf1 nf2
```

However, we can do better. We'd like to only reduce as much as necessary. Sometimes we can equate the terms without completely reducing them.

```
equate t1 t2 = do
   when (aeq t1 t1) $ return ()
    nf1 <- whnf t1  -- reduce only to 'weak head normal form'
```

```
nf2 <- whnf t2
case (nf1,nf2) of
  (App a1 a2, App b1 b2) ->
      -- make sure subterms are equal
      equate a1 b1 >> equate a2 b2
  (Lam bnd1, Lam bnd2) -> do
      -- ignore variable name and typing annot (if present)
      (_, b1, _, b2) <- unbind2Plus bnd1 bnd2
       equate b1 b2
  (_,_) -> err ...
```

Therefore, we reuse our mechanism for reducing terms to weak-head normal form.
Why weak-head reduction vs. full reduction?

- We can implement deferred substitutions for variables. Note that when comparing terms we need to have the definitions available. That way we can compute that `(plus 3 1)` weak-head normalizes to 4, by looking up the definition of `plus` when needed. However, we don't want to substitute all variables through eagerly—not only does this make extra work, but error messages can be extremely long.
- Furthermore, we allow recursive definitions in pi-forall, so normalization may just fail completely. However, this definition based on wnhf only unfolds recursive definitions when necessary, and then only once, so avoids some infinite loops in the type checker.

  Note that we don't have a complete treatment of equality though. There will always be terms that can cause `equate` to loop forever. On the other hand, there will always be terms that are not equated because of conservativity in unfolding recursive definitions.

6 Dependent pattern matching

6.0.1 Discussion of bi-directional rules for booleans and sigma types

```
---------------- Bool
G |- Bool <=> Type

---------------- true
G |- true <=> Bool

 ------------------- false
G |- false <=> Bool

 G |- a <= Bool
 G |- b <=> A
 G |- c <=> A
 --------------------------- if
 G |- if a then b else c <=> A

G |- A <= Type     G, x:A |- B <= Type
------------------------------------ sigma
G |- { x : A | B } <=> Type

G |- a <= A      G |- b <= B { a / x }
 ---------------------------------- pair
 G |- (a,b) <= { x : A | B }
```

```
G |- a => { x : A | B }
 G, x:A, y:B |- b <=> C
 G |- C <= Type
--------------------------------- weak-pcase
G |- pcase a of (x,y) -> b <=> C
```

6.0.2 Alternative rules for if and pcase

Consider our elimination rules for if:

```
 G |- a : Bool
 G |- b : A
 G |- c : A
 ---------------------------- if
 G |- if a then b else c : A
```

We can do better by making the type `A` depend on whether the scrutinee is true or false.

```
 G |- a : Bool
 G |- b : A { true/x }
 G |- c : A { false/x }
 ---------------------------- if
 G |- if a then b else c : A{a/x}
```

For example, here is a simple definition that requires this rule:

```
 -- function from booleans to types
 T : Bool -> Type
 T = \b. if b then One else Bool

-- returns unit when the argument is true
bar : (b : Bool) -> T b
bar = \b .if b then tt else True
```

It turns out that this rule is difficult to implement without annotating the expression with `x` and `A`. Given `A{true/x}`, `A{false/x}`, and `A{a/x}` (or anything that they are definitionally equal to!) how can we figure out whether they correspond to eachother?

So, we'll not be so ambitious. We'll only allow this refinement when the scrutinee is a variable.

```
 G |- x : Bool
 (G |- b : A) { true / x }
 (G |- c : A) { false / x }
 ---------------------------- if
 G |- if x then b else c : A
```

And, in going to our bidirectional system, we'll only allow refinement when we are in checking mode.

```
 G |- x => Bool
 G |- b <= A { true / x }
 G |- c <= A { false / x }
 ---------------------------- if
 G |- if x then b else c <= A
```

Then, we only have to remember that x is true / false when checking the individual branches of the if expression.

Here is an alternative version, for inference mode only, suggested during lecture:

```
 G |- a => Bool
 G |- b => B
 G |- c => C
 ------------------------------------------------ if
 G |- if a then b else c => if a then B else C
```

It has a nice symmetry—if expressions are typed by if. Note however, to make this rule work, we'll need a stronger definitional equivalence than we have. In particular, we'll want our definition of equivalence to support the following equality:

```
 ----------------------
 if a then b else b = b
```

That way, if the type of the two branches of the if does not actually depend on the boolean value, we can convert the `if` expression into a more useful type.

We can modify the rule for sigma types similarly.

```
G |- z => { x : A | B }
 G, x:A, y:B |- b <= C { (x,y) / z }
 G |- C <= Type
---------------------------------- pcase
G |- pcase z of (x,y) -> b <= C
```

This modification changes our definition of Sigma types from weak Sigmas to strong Sigmas. With either typing rule, we can define the first projection

```
fst : (A:Type) -> (B : A -> Type) -> (p : { x2 : A | B x2 }) -> A
 fst = \A B p. pcase p of (x,y) -> x
```

But, weak Sigmas cannot define the second projection using pcase. The following code only type checks using the above rule.

```
snd : (A:Type) -> (B : A -> Type) -> (p : { x2 : A | B x2 }) -> B (fst A B p)
snd = \A B p. pcase p of (x1,y) -> y
```

7 Propositional equality

You started proving things right away in Coq with an equality proposition. For example, in Coq, when you say

```
Theorem plus_O_n : forall n : nat, 0 + n  = n
```

You are using a built in type, `a = b` that represents the proposition that two terms are equal.

As a step towards more general indexed datatypes, we'll start by adding just this type to pi-forall.

The main idea of the equality type is that it converts a *judgement* that two types are equal into a *type* that is inhabited only when two types are equal. In other words, we can write the intro rule for this form as:

```
 G |- a = b
```

```
------------------- refl
G |- refl : a = b
```

Sometimes, you might see the rule written as follows:

```
------------------- refl'
G |- refl : a = a
```

However, this rule will turn out to be equivalent to the above version.

This *type* is well-formed when both sides have the same type. In other words, when it implements *homogeneous* equality.

```
G |- a : A    G |- b : A
------------------------ eq
G |- a = b : Type
```

The elimination rule for propositional equality allows us to convert the type of one expression to another.

```
 G |- a : A { a1 / x }   G |- b : a1 = a2
------------------------------------------------------ subst
 G |- subst a by b : A { a2 / x }
```

How can we implement this rule? For simplicity, we'll play the same trick that we did with booleans, requiring that one of the sides of the equality be a variable.

```
 G |- a <= A { a1 / x }    G |- b => x = a1
 ---------------------------------------- subst-left
 G |- subst a by b => A

G |- a <= A { a1 / x }    G |- b => a1 = x
 ---------------------------------------- subst-right
 G |- subst a by b => A
```

Note that our elimination form for equality is powerful. We can use it to show that propositional equality is symmetric and transitive.

```
sym : (A:Type) -> (x:A) -> (y:A) -> (x = y) -> y = x
trans : (A:Type) -> (x:A) -> (y:A) -> (z:A) -> (x = z) -> (z = y) -> (x = y)
```

Furthermore, we can also extend `subst`, the elimination form for propositional equality as we did for booleans. This corresponds to the following elimination rule for subst, that observes that the only way to construct a proof of equality is with the term `refl`. (This version of subst is very close to an eliminator for propositional equality called `J`).

```
 G |- a : A { a1 / x }{ refl / y }   G |- b : a1 = a2
------------------------------------------------------ subst
 G |- subst a by b : A { a2 / x }{ b / y }
```

As above, this rule (and the corresponding subst-right rule) only applies when `b` is also a variable.

```
G |- a <= A { a1 / x } { refl / y }    G |- y => x = a1
 ------------------------------------------------------ subst-left
 G |- subst a by y => A
```

One last addition: `contra`. If we can somehow prove a false, then we should be able to prove anything. A contradiction is a proposition between two terms that have different head forms. For now, we'll use:

```
 G |- p : True = False
--------------------- contra
 G |- contra p : A
```

7.0.1 Homework (pi-forall: more church encodings)

The file `version2/test/NatChurch.pi` is a start at a Church encoding of natural numbers. Replace the TRUSTMEs in this file so that it compiles.

7.0.2 Homework (pi-forall: equality)

Complete the file Hw2.pi. This file gives you practice with working with equality propositions in pi-forall.

A Full specification of the system

tname, x, y, z, f, g, n	variables			
mname, M	module names			
tm, a, b, A, B, u, v	::=			terms and types
	|	**Type**		sort
	|	x		variable
	|	$\lambda x.a$		function
	|	$a\ b$		function application
	|	$(x:A) \to B$		dependent function type
	|	$a\{b/x\}$	S	substitution
	|	$(a : A)$		type annotation
	|	(a)	S	parenthesis
	|	**TRUSTME**		A term that has any type
	|	**PRINTME**		Print the current context
	|	**let** $x = a$ **in** b		Name an expression
	|	**Unit**		unit type
	|	$()$		unit term
	|	**Bool**		boolean type
	|	**True**		boolean value true
	|	**False**		boolean value false
	|	**if** a **then** b_1 **else** b_2		conditional
	|	$\{x:A \mid B\}$		Σ-type (i.e. dependent products/dependent sums)
	|	(a, b)		product
	|	**let** $(x, y) = a$ **in** b		elimination form for pairs
	|	$a = b$		equality type
	|	**refl**		reflexivity proof
	|	**subst** a **by** b		equality type elimination
	|	**contra** a		false elimination
context, Γ	::=			contexts
	|			
	|	$\Gamma, x:A$		
	|	$x:A$		

$\boxed{a \rightsquigarrow b}$ single-step operational semantics

$$\frac{}{(\lambda x.a)\ b \rightsquigarrow a\{b/x\}}\ \textsc{AppAbs}$$

$$\frac{}{\mathbf{let}\, x = a\, \mathbf{in}\, b \rightsquigarrow b\{a/x\}}\ \textsc{Let}$$

$$\frac{}{\mathbf{if}\, \mathbf{True}\, \mathbf{then}\, b_1\, \mathbf{else}\, b_2 \rightsquigarrow b_1}\ \textsc{IfTrue}$$

$$\frac{}{\mathbf{if}\, \mathbf{False}\, \mathbf{then}\, b_1\, \mathbf{else}\, b_2 \rightsquigarrow b_1}\ \textsc{IfFalse}$$

$$\frac{}{\mathbf{let}\, (x, y) = (a_1, a_2)\, \mathbf{in}\, b \rightsquigarrow b\{a_1/x\}\{a_2/y\}}\ \textsc{LetPairProd}$$

$$\frac{a \rightsquigarrow a'}{a\ b \rightsquigarrow a'\ b}\ \textsc{App}$$

$$\frac{a \rightsquigarrow a'}{\mathbf{if}\, a\, \mathbf{then}\, b_1\, \mathbf{else}\, b_2 \rightsquigarrow \mathbf{if}\, a'\, \mathbf{then}\, b_1\, \mathbf{else}\, b_2}\ \textsc{If}$$

$$\frac{a \rightsquigarrow a'}{\mathbf{let}\, (x, y) = a\, \mathbf{in}\, b \rightsquigarrow \mathbf{let}\, (x, y) = a'\, \mathbf{in}\, b}\ \textsc{LetPair}$$

$\boxed{a \rightsquigarrow^* b}$ multi-step

$$\frac{}{a \rightsquigarrow^* a}\ \textsc{Equal}$$

$$\frac{a \rightsquigarrow b \quad b \rightsquigarrow^* a'}{a \rightsquigarrow^* a'}\ \textsc{Step}$$

$\boxed{\Gamma \vdash a : A}$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{T\_VAR}$$

$$\frac{\Gamma, x{:}A \vdash a : B}{\Gamma \vdash \lambda x.a : (x{:}A) \to B}\ \text{T\_SLAMBDA}$$

$$\frac{\Gamma, x{:}A \vdash a : B \quad \Gamma \vdash a : \mathbf{Type}}{\Gamma \vdash \lambda x.a : (x{:}A) \to B}\ \text{T\_LAMBDA}$$

$$\frac{\Gamma \vdash A : \mathbf{Type} \quad \Gamma, x{:}A \vdash B : \mathbf{Type}}{\Gamma \vdash (x{:}A) \to B : \mathbf{Type}}\ \text{T\_PI}$$

$$\frac{}{\Gamma \vdash \mathbf{Type} : \mathbf{Type}}\ \text{T\_TYPE}$$

$$\frac{\Gamma \vdash a : (x{:}A) \to B \quad \Gamma \vdash b : A}{\Gamma \vdash a\ b : B\{b/x\}}\ \text{T\_APP}$$

$$\frac{}{\Gamma \vdash \mathbf{Bool} : \mathbf{Type}}\ \text{T\_BOOL}$$

$$\frac{}{\Gamma \vdash \mathbf{True} : \mathbf{Bool}}\ \text{T\_TRUE}$$

$$\frac{}{\Gamma \vdash \mathbf{False} : \mathbf{Bool}} \quad \text{T\_FALSE}$$

$$\frac{\begin{array}{c}\Gamma \vdash a : \mathbf{Bool} \\ \Gamma \vdash b_1 : A \\ \Gamma \vdash b_2 : A\end{array}}{\vdash \mathbf{if}\ a\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 : A} \quad \text{T\_IF}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : \mathbf{Type} \\ \Gamma, x{:}A \vdash B : \mathbf{Type}\end{array}}{\Gamma \vdash \{x{:}A \mid B\} : \mathbf{Type}} \quad \text{T\_SIGMA}$$

$$\frac{\begin{array}{c}\Gamma \vdash a : A \\ \Gamma \vdash b : B\{a/x\}\end{array}}{\Gamma \vdash (a, b) : \{x{:}A \mid B\}} \quad \text{T\_PAIR}$$

$$\frac{\begin{array}{l}\Gamma \vdash a : \{x{:}A_1 \mid A_2\} \\ \Gamma, x{:}A_1, y{:}A_2 \vdash b : B \\ \Gamma \vdash B : \mathbf{Type}\end{array}}{\Gamma \vdash \mathbf{let}\,(x, y) = a\,\mathbf{in}\,b : B} \quad \text{T\_LETPAIR\_WEAK}$$

$\boxed{\Gamma \vdash a \Rightarrow A}$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \quad \text{I\_VAR}$$

$$\frac{\Gamma \vdash a \Leftarrow A}{\Gamma \vdash (a : A) \Rightarrow A} \quad \text{I\_ANNOT}$$

$$\frac{\begin{array}{c}\Gamma \vdash a \Rightarrow (x{:}A) \to B \\ \Gamma \vdash b \Leftarrow A\end{array}}{\Gamma \vdash a\ b \Rightarrow B\{b/x\}} \quad \text{I\_APP}$$

$$\frac{\begin{array}{c}\Gamma \vdash A \Leftarrow \mathbf{Type} \\ \Gamma, x{:}A \vdash B \Leftarrow \mathbf{Type}\end{array}}{\Gamma \vdash (x{:}A) \to B \Rightarrow \mathbf{Type}} \quad \text{I\_PI}$$

$$\frac{}{\Gamma \vdash \mathbf{Type} \Rightarrow \mathbf{Type}} \quad \text{I\_TYPE}$$

$\boxed{\Gamma \vdash a \Leftarrow B}$

$$\frac{\begin{array}{c}\Gamma, x{:}A \vdash a \Leftarrow B \\ \Gamma \vdash A \Leftarrow \mathbf{Type}\end{array}}{\Gamma \vdash \lambda x.a \Leftarrow (x{:}A) \to B} \quad \text{C\_LAMBDA}$$

$$\frac{\Gamma \vdash a \Rightarrow A}{\Gamma \vdash a \Leftarrow A} \quad \text{C\_INFER}$$