Implementing Dependent Types in `pi-forall`

Stephanie Weirich

May 26, 2022

1 Goals

These lecture notes describe the implementation of a small dependently-typed language called "`pi-forall`" and walk through the implementation of its type checker. They are based on lectures given at the *Oregon Programming Languages Summer School* during June 2022, and derived from earlier lectures from summer school lectures during 2014 and 2013. [1]

What do I expect from you, dear reader? These notes assume a familiarity with the basics of the lambda calculus, including its standard operations (alpha-equivalence, substitution, evaluation) and the basics of type systems (specification of type systems using inference rules). For background on this material, I recommend [Pie02, Har16].

Furthermore, these lectures and notes also refer to an implementation of a demo type checker and assume basic knowledge of the Haskell programming language. This implementation is available at `https://github.com/sweirich/pi-forall` and is written using the Haskell programming language. As you study these notes, I encourage you to download this code and experiment with it. Detailed installation instructions are available with the source code.

Key feature	`pi-forall` version	Section
Core system	`version1`	Sections 2, 3, and 4
Equality	`version2`	Sections 5 and 6
Irrelevance	`version3`	Section 7
Datatypes	`full`	Section 8

Figure 1: Connection between sections and `pi-forall` versions

What do these notes cover? These notes are broken into several sections that incrementally build up the design and implementation of a type checker for a dependently-typed programming language.

[1] NOTE: if the date on this document is before June 27, 2022, then these notes are still

If you are looking at the repository, you'll see that it includes several incrementally more expressive implementations in separate subdirectories. These implementations build on each other (each is an extension of the previous) and are summarized in Figure 1. As you read each chapter, refer to its corresponding implementation to see how the features described in that chapter can be implemented.

- Section 2 presents the mathematical specification of the core type system including its concrete syntax (as found in `pi-forall` source files), abstract syntax, and core typing rules (written using standard mathematical notation). This initial specification of the type system is simple and declarative. It *specifies* what terms should type check, but cannot be directly implemented.

- Section 3, then reformulates the typing rules so that they are *syntax-directed* and specify a type checking algorithm. The key idea of this section is to recast the typing rules as a *bidirectional type system* that specifies where type annotations are required.

- Section 4 introduces the core `pi-forall` implementation and walks through the type checker found in `version1`, the Haskell implementation of the typing rules discussed in Section 3. This section shows how the `Unbound` library can assist with variable binding and automatically derive operations for capture-avoiding substitution and alpha-equivalence. This section also describes a monadic structure for the type checker, and how it can help with the production of error messages, the primary purpose of a type checker.

- Section 5 discusses the role of definitional equality in dependently typed languages. After motivating examples, it presents both a specification of when terms are equal and a semi-decidable algorithm that can be incorporated into the type checker.

- An important feature of dependently-typed languages is the ability for run time tests to be reflected into the type system. Section 6 shows how to extend `pi-forall` with a simple form of dependent pattern matching.

- Section 7 introduces the idea of tracking the *relevance* of arguments. Relevance tracking enables parts of the program to be identified as "compile-time only" and thus erased before execution. It also identifies parts of terms that can be ignored when deciding equivalence.

- Finally, Section 8 introduces arbitrary datatypes and generalizes dependent pattern matching. This section combines the features introduced in the previous sections (Sections 5, 6, and 7) into a single unified language feature.

What do these notes not cover? The goal of this notes is to provide an introductory overview. As a result, there are many topics related to the implementation of dependent type theories and programming languages that are not included here. Furthermore, because these notes come with a reference implementation, they emphasize only a single point in the design space.

- For simplicity, the `pi-forall` language does not enforce termination through type checking. Implementing a proof system like Agda or Coq requires additional structure, including universe levels and bounded iteration.
- Many implementations of dependent type theories use *normalization-by-evaluation* to decide whether types are equivalent during type checking. `pi-forall` uses an alternative approach based on weak-head-normalization, defined using substitution. This approach is closer to λ-calculus theory, but can be less efficient in practice.
- Also for simplicity, `pi-forall` does not attempt to infer arguments, using unification or other means. As a result, example programs in `pi-forall` are significantly more verbose than in other languages.
- This implementation relies on the `Unbound` library for variable binding, alpha-equivalence and substitution. As a result, most issues related to this topic are avoided.
- Recent work on cubical type theory, higher-inductive types and univalence is not covered here.

Section 9 includes a discussion of related tutorials, including how this one differs, as well as pointers for where to go for more information.

What do I want you to get out of all of this?

1. An understanding of how to translate mathematical specifications of type systems and logics into code, i.e. how to represent the syntax of a programming language and how to implement a type checker. More generally, this involves techniques for turning a declarative specification of a system of judgments into an algorithm that determines whether the judgement holds.
2. Exposure to the design choices of dependently-typed languages. In this respect, my goal is breadth not depth. As a result, I will provide *simple* solutions to some of the problems that we face and sidestep other problems entirely. Because solutions are chosen for simplicity, the end of these notes includes pointers if you want to go deeper.
3. Experience with the Haskell programming language. I think Haskell is

in draft form. If you find any errors or typos, please send me a pull request at `https://github.com/sweirich/pi-forall`.

an awesome tool for this sort of work and I want to demonstrate how its features monads, generic programming) are well-suited for this task.

4. A tool that you can use as a basis for experimentation. When you design your language, how do you know what programs you can and cannot express? Having an implementation lets you work out (smallish) examples and will help convince you (and your reviewers) that you are developing something useful. Please use `pi-forall` as a starting point for your ideas.

5. Templates and tools for writing about type systems. The source files for these lecture notes are available in the same repository as the demo implementation and I encourage you to check them out.[2] Building these notes requires Ott [SNO+07], a tool specifically tailored for typesetting type systems and mathematical specifications of programming languages.

2 A Simple Core Language

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

a, b, A, B	$::=$	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

As in many dependently-typed languages, we have the *same* syntax for both 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 inductive relation shown in Figure 2. This relation is between an expression a, its type A, and a typing context Γ.

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

A typing context Γ is an ordered list of assumptions about variables and their types.

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

[2] See the `doc` subdirectory in `https://github.com/sweirich/pi-forall`.

$\boxed{\Gamma \vdash a : A}$ *(Core type system)*

$$\text{T-VAR}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \text{T-LAMBDA}\ \frac{\Gamma, x:A \vdash a : B \quad \Gamma \vdash A : \mathbf{Type}}{\Gamma \vdash \lambda x.a : (x:A) \to B} \qquad \text{T-APP}\ \frac{\Gamma \vdash a : (x:A) \to B \quad \Gamma \vdash b : A}{\Gamma \vdash a\ b : B[b/x]}$$

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

Figure 2: Typing rules for core system

We will assume that each of the variables in this list are distinct from each other, so that there will always be at most one assumption about any variable's type.[3]

An initial set of typing rules: Variables and Functions Consider the first two rules shown in Figure 2, rule T-VAR and rule T-LAMBDA. The first rule states that the types of variables are recorded in the typing context. The premise $x : A \in \Gamma$ requires us to find an association between x and the type A in the list Γ.

The second rule, for type checking functions, introduces a new variable into the context when we type check the body of a λ-expression. It also requires A, the type of the function's argument to be a valid type.

Example: Polymorphic identity functions Note that in rule T-LAMBDA, the parameter x is allowed to appear in the result type of the function, B. Why is this useful? Well, it gives us *parametric polymorphism* automatically. In Haskell, we write the identity function as follows, annotating it with a polymorphic type.

```
id :: forall a. a → a
id = λy → y
```

Because the type of `id` is generic we can apply this function to any type of argument.

We can also write a polymorphic identity function in `pi-forall`, as follows.

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

[3]On paper, this assumption is reasonable because we always extend the context with variables that come from binders, and we can always rename bound variables so that they differ from other variables in scope. Any implementation of the type system will need to somehow make sure that this invariant is maintained.

This definition is similar to Haskell, but with a few modifications. First, `pi-forall` uses a single colon for type annotations and a `.` for function binding instead of Haskell's quirky `::` and $\rightarrow$. Second, the `pi-forall` definition of `id` uses two lambdas: one for the "type" argument x and one for the "term" argument y. In `pi-forall`, there is no syntactic difference between these arguments: both are arguments to the identity function.

Finally, the `pi-forall` type of `id` uses the dependent function type, with general form $(x : A) \rightarrow B$. This form is like Haskell's usual function type $A \rightarrow B$, except that we can name the argument x so that it can be referred to in the body of the type B. In `id`, dependency allows the the type argument x to be used later as the type of y. We call types of this form Π-types. (Often dependent function types are written as $\Pi x : A.B$ in formalizations of dependent type theory.[4])

The fact that the type of x is **Type** means that x plays the role of a type variable, such as `a` in the Haskell type. Because we don't have a syntactic distinction between types and terms, in `pi-forall` we say that "types" are anything of type **Type** and "terms" are things of type A where A has type **Type**.

We can use the typing rules to construct a typing derivation for the identity function as follows.

$$\dfrac{\dfrac{\dfrac{y : x \in (x : \mathbf{Type}, y : x)}{x : \mathbf{Type}, y : x \vdash y : x} \qquad \dfrac{x : \mathbf{Type} \in (x : \mathbf{Type})}{x : \mathbf{Type} \vdash x : \mathbf{Type}}}{x : \mathbf{Type} \vdash \lambda y.y : (y : x) \rightarrow x} \qquad \vdash \mathbf{Type} : \mathbf{Type}}{\vdash \lambda x.\lambda y.y : (x : \mathbf{Type}) \rightarrow (y : x) \rightarrow x}$$

This derivation uses rule T-LAMBDA to type check the two λ-expressions, before using the variable rule to ensure that both the body of the function x and its type y are well-typed. Finally, this rule also uses rule T-TYPE to show that **Type** itself is a valid type, see below.

More typing rules: Types Observe in the typing derivation above that the rule for typing lambda-expressions has second precondition: we need to make sure that when we add assumptions $x : A$ to the context, that A really is a type. Without this precondition, the rules would allow us to derive this nonsensical type for the polymorphic identity function.

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

This precondition means that we need some rules that conclude that types are actually types. For example, the type of a function is itself a type, so we will declare it so with rule T-PI. This rule also ensures that the domain and

[4]The terminology for these types is muddled: sometimes they are call dependent function types and sometimes they are called dependent product types. We use the non Π-notation to emphasize the connection to functions.

range of the function are also types.

Likewise, for polymorphism we need the somewhat perplexing rule T-TYPE, that declares, by fiat, that **Type** is a valid **Type**.[5]

More typing rules: Applications The application rule, rule T-APP, requires that the type of the argument matches the domain type of the function. However, note that because the body of the function type B could have x free in it, we need also need to substitute the argument b for x in the result.

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`, producing an expression of type $(y : ((x : \mathbf{Type}) \to (y : x) \to x)) \to ((x : \mathbf{Type}) \to (y : x) \to x)$. This function can take `id` as an argument.

```
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, called a Church 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" We can also encode a logical "and" data structure using a Church encoding.

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

[5]Note that, this rule make our language inconsistent as a logic, as it can encode Girard's

```
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

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

A set of typing rules, or *type system*, is called *syntax-directed* if it is readily apparent how to interpret that collection of typing rules as code. In other words, for some type systems, we can directly translate them to some Haskell function.

For this type system, we would like to implement the following Haskell function, that when given a term and a typing context, represented as a list of pairs of variables and their types, produces the type of the term, if it exists.[6]

```
type Ctx = [(Var,Type)]

inferType :: Term → Ctx → Maybe Type
```

Let's look at our rules in Figure 2. Is the definition of this function straightforward? For example, in the variable rule, as long as we can look up the type of a variable in the context, we can produce its type. That means that, assuming that there is some function `lookupTy`, with type `Ctx → Var → Maybe Type`, this rule corresponds to the following case of the `inferType` function. So far so good!

```
inferType (Var x) ctx = lookupTy ctx x
```

Likewise, the case of the typing function for the **Type** term is also straightforward. When we see the term **Type**, we know immediately that it is its own type.

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

The only stumbling block for the algorithm is rule T-LAMBDA. To type check a function, we need to type check its body when the context has been extended

paradox. More about this in section 9.

[6]Note: If you are looking at the `pi-forall` implementation, note that this is not the final type of `inferType`.

with the type of the argument. But, like Haskell, the type of the argument A is not annotated on the function in `pi-forall`. So where does it come from?

There is actually an easy fix to turn our current system into an algorithmic one. We could just annotate λ-expressions with the types of the abstracted variables. But, to be ready for future extension, we'll do something else.

Look at our example `pi-forall` code above: the only types that we wrote were the types of definitions. It is good style to do that. Furthermore, there is enough information there for type checking—wherever we define a function, we can look at those types to know what type its argument should have. So, by changing 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*.[7] 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 express these judgements using the notation defined in Figure 3 and implement them in Haskell using the mutually-recursive functions `inferType` and `checkType`. Furthermore, to keep track of which rule is in which judgement, rules that have inference as a conclusion start with I- and rules that have checking as a conclusion start with C-.

Let's compare these rules with our original typing rules. For rule I-VAR, we need to only change the colon to an inference arrow. The context tells us the type to infer.

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

The rule for applications, rule I-APP-SIMPLE, is in inference mode. Here, we first infer the type of the function, but once we have that type, we may use it to check the type of the argument. This mode change means that $\lambda-$expressions that are arguments to other functions (like `map`) do not need any annotation.

For types, it is apparent their type is **Type**, so rules I-PI and I-TYPE just continue to infer that.

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 with rule C-INFER-SIMPLE 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

[7]The term *type inference* usually refers to much more sophisticated deduction of an expressions type, in the context of much less information encoded in the syntax of the language. We're not doing anything difficult here, just noting that we can read the judgment with A as an output.

$\boxed{\Gamma \vdash a \Rightarrow A}$ *(in context Γ, infer that term a has type A)*

$$\text{I-VAR}\ \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \text{I-APP-SIMPLE}\ \frac{\Gamma \vdash a \Rightarrow (x:A) \to B \qquad \Gamma \vdash b \Leftarrow A}{\Gamma \vdash a\ b \Rightarrow B[b/x]} \qquad \text{I-PI}\ \frac{\Gamma \vdash A \Leftarrow \mathbf{Type} \qquad \Gamma, x:A \vdash B \Leftarrow \mathbf{Type}}{\Gamma \vdash (x:A) \to B \Rightarrow \mathbf{Type}}$$

$$\text{I-TYPE}\ \frac{}{\Gamma \vdash \mathbf{Type} \Rightarrow \mathbf{Type}} \qquad \text{I-ANNOT}\ \frac{\Gamma \vdash a \Leftarrow A}{\Gamma \vdash (a : A) \Rightarrow A}$$

$\boxed{\Gamma \vdash a \Leftarrow A}$ *(in context Γ, check that term a has type A)*

$$\text{C-LAMBDA}\ \frac{\Gamma, x:A \vdash a \Leftarrow B \qquad \Gamma \vdash A \Leftarrow \mathbf{Type}}{\Gamma \vdash \lambda x.a \Leftarrow (x:A) \to B} \qquad \text{C-INFER-SIMPLE}\ \frac{\Gamma \vdash a \Rightarrow A}{\Gamma \vdash a \Leftarrow A}$$

Figure 3: (Simple) Bidirectional type system

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:

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

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.

Type annotations To type check nonnormal forms in `pi-forall`, we also add typing annotations as a new form of expression to `pi-forall`, written $(a : A)$, and add rule I-ANNOT to the type system.

Type annotations allow us to supply known type information anywhere, not just at top level. For example, we can construct this derivation.

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

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.

Furthermore, we want to convince ourselves that the bidirectional system checks the same terms as the original type system. This means that we want to prove a property like this one:

Lemma 3.1 (Correctness of the bidirectional type system)**.** If $\Gamma \vdash a \Rightarrow A$ then $\Gamma \vdash a : A$. If $\Gamma \vdash a \Leftarrow A$ then $\Gamma \vdash a : A$.

On the other hand, the reverse property is not true. Even if there exists some typing derivation $\Gamma \vdash a : A$ for some term a, we may not be able to infer or check that term using the algorithm. However, all is not lost: there will always be some term a' that differs from a only in the addition of typing annotations that can be inferred or checked instead.

One issue with this bidirectional system is that it is not closed under substitution. What this means is that given some term b with a free variable, $\Gamma, x : A \vdash b \Leftarrow B$, and another term a with the same type $\Gamma \vdash a \Leftarrow A$ of that variable, we *do not* have a derivation $\Gamma \vdash b[a/x] \Leftarrow A$. The reason is that types of variables are always inferred, but the term a, may need the type A to be supplied to the type checker. This issue is particularly annoying in rule I-APP when we replace a variable (inference mode) with a term that was validated in checking mode.

As a result, our type system infers types, but the types that are inferred do not have enough annotations to be checked themselves. We can express the property that does hold about our system, using this lemma:

Lemma 3.2 (Regularity)**.** If $\Gamma \vdash a \Rightarrow A$ then $\Gamma \vdash A : \mathbf{Type}$.

This issue is not significant, but we could resolve it by adding an annotation before substitution. However, in our implementation of `pi-forall`, we do not do so to reduce clutter.

4 Putting it all together in a Haskell implementation

In the previous section, we defined a bidirectional type system for a small core language. Here, we'll start talking about what the implementation of this language looks like in Haskell.

First, an overview of the main files of the implementation. There are a few more source files in the repository (in the subdirectory `src/`), but these are the primary ones.

```
Syntax.hs      - specification of the 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
```

4.1 Variable binding using the `Unbound` library [Syntax.hs]

One difficulty with implementing any sort of 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 λ-expression, we will want to be sure that the binding variable is *fresh*, that is, distinct from all other variables in the program or in the typing context.

In the `pi-forall` implementation, we use the `Unbound` library to get all of these operations for free. This works because we will use types from this library in the definition of the abstract syntax of `pi-forall`, and those types will specify the binding structure of `pi-forall` expressions.

First, the `Unbound` library defines a type for variable names, called `Name` and we use this type to define `TName`, the type of term names in our AST.

```
-- | Type used for variable names in Terms
type TName = Unbound.Name Term
```

This type is indexed by `Term` the AST type that it is a name for. That way `Unbound` can make sure that we only substitute the right form of syntax for the right name, i.e. we can only replace a `TName` with a `Term`. The `Unbound` library includes an overloaded function `subst x a b`, which means $b[a/x]$[8], i.e. substitute a for x in b.

In general, we will want to apply a substitution to many different sorts of syntax. For example, we may want to substitute a term a for a term name x in all of the terms that appear in some typing context. Therefore, `Unbound`'s substitution function has a type with the following general pattern for `subst a x b`.

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

The `subst` function in this class ensures that when we see that `a` is the right sort of thing to stick in for `x`. The library can automatically generate instances of the `Subst` class.

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

```
data Term
```

[8] See Guy Steele's talk about the notation for substitution [Ste17]. This is the most common mathematical notation for this operation.

```
  = -- | type of types  `Type`
    Type

  | -- | variables  `x`
    Var TName

  | -- | abstractions  `\x. a`
    Lam (Unbound.Bind TName Term)

  | -- | applications `a b`
    App Term Arg

  | -- | function types   `(x : A) -> B`
    Pi Type (Unbound.Bind TName Type)

  | -- | Annotated terms `( a : A )`
    Ann Term Type

  ...
  deriving (Show, Generic)

-- | An argument to a function
data Arg = Arg {unArg :: Term}
  deriving (Show, Generic, Unbound.Alpha, Unbound.Subst Term)
```

As you can see, variables are represented by names. `Unbound`'s `Bind` type constructor declares the scope of the bound variables. Both `Lam` and `Pi` bind a single variable in a `Term`. However, in a `Pi` term, we also want to store the type A, the type of the bound variable x.

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
```

We also isolate function arguments as a special type `Arg`, which is isomorphic to `Term`. This distinction is not useful for the initial version of `pi-forall`, but is important for future extensions.

The definitions of the `Arg` datatype automatically derives 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
```

For `Term`, we do not use the default definitions of the `Alpha` and `Subst` classes. Instead, we need to provide a bit more information. First, we would

like the definition of alpha-equivalence of terms to ignore type annotations (as well as other practicalities, such as source code positions for error messages and explicit parentheses). Therefore our instance does so for these constructors of the `Term` data type and defers to the generic definition for the rest.

The `Subst` instance for `Term` requires telling `Unbound` where variables occur in the `Term`. This short function pattern matches to find the `Var` constructor, extract its name and return it to `Unbound`.

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

For more information about `Unbound`, see the documentation for `unbound-generics`[9].

4.2 A Type Checking 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 Type

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

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. Therefore our two functions will have this type: hiding the context and part

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

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

Those of you who have worked with Haskell before may be familiar with the monads 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
```

[9] `https://github.com/lambdageek/unbound-generics`

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

4.3 Implementing the Type Checking 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 Type
inferType t = tcTerm t Nothing

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

The `tcTerm` function checks a term, producing its type. The second argument is `Nothing` in inference mode and an expected type in checking mode.

```
tcTerm :: Term -> Maybe Type -> TcMonad 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 tyA bnd) Nothing = ...

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

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

```
tcTerm (Lam bnd) (Just (Pi tyA bnd2)) = ...    -- pass in the Pi type for the lambda express

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
 ty' <- inferType tm
 unless (Unbound.aeq ty' ty) $
   err [DS "Types don't match", DD ty, DS "and", DD ty']
 return ty
```

The function `aeq` ensures that the two types are alpha-equivalent. If they are not, it stops the computation and throws an error.

4.4 Example

The `pi-forall` source file `Lec1.pi` contains the examples that we worked out in Section 2. Let's try to type check it, after filling in the missing code in `TypeCheck.hs`.

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

Some fairly standard typing rules for booleans are shown below.

$\boxed{\Gamma \vdash a : A}$ *(Booleans)*

$$\textsc{T-BOOL}\ \frac{}{\Gamma \vdash \mathbf{Bool} : \mathbf{Type}} \qquad \textsc{T-TRUE}\ \frac{}{\Gamma \vdash \mathbf{True} : \mathbf{Bool}} \qquad \textsc{T-FALSE}\ \frac{}{\Gamma \vdash \mathbf{False} : \mathbf{Bool}}$$

$$\textsc{T-IF-WEAK}\ \frac{\Gamma \vdash a : \mathbf{Bool} \quad \Gamma \vdash b_1 : A \quad \Gamma \vdash b_2 : A}{\vdash \mathbf{if}\ a\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 : A}$$

Likewise, we can also extend the language with Σ-types.

$\boxed{\Gamma \vdash a : A}$ *(Sigma-types)*

$$\textsc{T-SIGMA}\ \frac{\Gamma \vdash A : \mathbf{Type} \quad \Gamma, x{:}A \vdash B : \mathbf{Type}}{\Gamma \vdash \{x{:}A \mid B\} : \mathbf{Type}} \qquad \textsc{T-PAIR}\ \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \{x{:}A \mid B\}}$$

$$\textsc{T-LETPAIR-WEAK}\ \frac{\Gamma \vdash a : \{x{:}A_1 \mid A_2\} \quad \Gamma, x{:}A_1, y{:}A_2 \vdash b : B \quad \Gamma \vdash B : \mathbf{Type}}{\Gamma \vdash \mathbf{let}\ (x, y) = a\ \mathbf{in}\ b : B}$$

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

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

Next, we are going to think about how adding a notion of definitional equality can make our language more expressive.

The addition of definitional equality follows several steps. First, we will see why we even want this feature in the language in the first place. Next, we will create a declarative specification of definitional equality and extend our typing relation with a new rule that enables it to be used. After that, we'll talk about algorithmic versions of both the equality relation and how to introduce it into the algorithmic type system. Finally, we'll cover modifications to the Haskell implementation.

5.1 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 merely alpha-equivalent, 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 = λp. λq. (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 = λp.λq. λx.λy. λc. λf. 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 type checker 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 this application.

5.2 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

$\boxed{\Gamma \vdash A = B}$ *(Definitional Equality)*

$$\text{E-BETA}\ \frac{}{\Gamma \vdash (\lambda x.a)\ b = a[b/x]} \qquad \text{E-REFL}\ \frac{}{\Gamma \vdash A = A} \qquad \text{E-SYM}\ \frac{\Gamma \vdash A = B}{\Gamma \vdash B = A} \qquad \text{E-TRANS}\ \frac{\Gamma \vdash A_1 = A_2 \quad \Gamma \vdash A_2 = A_3}{\Gamma \vdash A_1 = A_3}$$

$$\text{E-PI}\ \frac{\Gamma \vdash A_1 = A_2 \quad \Gamma, x:A_1 \vdash B_1 = B_2}{\Gamma \vdash (x:A_1) \to B_1 = (x:A_2) \to B_2} \qquad \text{E-LAM}\ \frac{\Gamma, x:A_1 \vdash b_1 = b_2}{\Gamma \vdash \lambda x.b_1 = \lambda x.b_2}$$

$$\text{E-APP}\ \frac{\Gamma \vdash a_1 = a_2 \quad \Gamma \vdash b_1 = b_2}{\Gamma \vdash a_1\ b_1 = a_2\ b_2} \qquad \text{E-LIFT}\ \frac{\Gamma, x:A \vdash b : B \quad \Gamma \vdash a_1 = a_2}{\Gamma \vdash b[a_1/x] = b[a_2/x]} \qquad \text{E-ANNOT}\ \frac{\Gamma \vdash a_1 = a_2}{\Gamma \vdash (a_1 : A) = a_2}$$

Figure 4: Definitional equality for core `pi-forall`

will not hold. We need to enrich our definition of equality so that it equates more terms.

5.3 Defining equality

The main idea is that we will:

- establish a new judgement that defines when types are equal

$$\boxed{\Gamma \vdash A = B}$$

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

$$\text{T-CONV}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash A = B}{\Gamma \vdash a : B}$$

- 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 have implicitly started with a very simple one: identify terms up to α-equivalence. But we can do better.

The rules in Figure 4 define what it means for two terms to be equivalent, by stating the properties that we want to be true of this relation. Rule E-BETA ensures that our relation *contains beta-equivalence*. Terms that evaluate

to eachother should be equal. Rules E-REFL, E-SYM, and E-TRANS makes sure that this relation is an *equivalence relation*. Furthermore, we want this relation to be a *congruence relation* (i.e. if subterms are equal, then larger terms are equal), as specified by rules rules E-PI, E-LAM, and E-APP. We also want to be sure that this relation has "functionality" (i.e. we can lift equalities). We declare so with rule E-LIFT. Finally, we want to ignore type annotations, so rule E-ANNOT allows the equality judgment to skip over them on the left (and with the help of rule E-SYM, on the right as well).

(Note: if we add booleans and $\Sigma-$types, we will also need corresponding β-equivalence rules and congruence rules for those constructs.)

5.4 Using definitional equality in the algorithm

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

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash A = B}{\Gamma \vdash a : B}\ \textsc{t-conv}$$

but this rule is not syntax-directed; it can be used at any place in a derivation so how do we know where to check types for equality? And which types should we check?

It turns out that in our bidirectional system, there are only a few places where type equality matters.

- We meed to check for equality when we switch from checking mode to inference mode in the algorithm in rule rule C-INFER. Here we need to ensure that the type that we infer is the same as the type that is passed to the checker.

  $$\frac{\Gamma \vdash a \Rightarrow A \qquad \Gamma \vdash A \Leftrightarrow B}{\Gamma \vdash a \Leftarrow B}\ \textsc{c-infer}$$

  In this case, the types A and B are available to the type checker, and the our equality algorithm must decide whether they are equal. We use $\Gamma \vdash A \Leftrightarrow B$ as the notation for an algorithmic version of our equality relation—the relation we defined in Figure 4 does not readily lead to an algorithm. (More on this algorithmic version in Section 5.5 below.)

- 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.

  $$\frac{\Gamma \vdash a \Rightarrow A \qquad \mathbf{whnf}\ A = (x : A_1) \to B \qquad \Gamma \vdash b \Leftarrow A_1}{\Gamma \vdash a\ b \Rightarrow B[b/x]}\ \textsc{i-app}$$

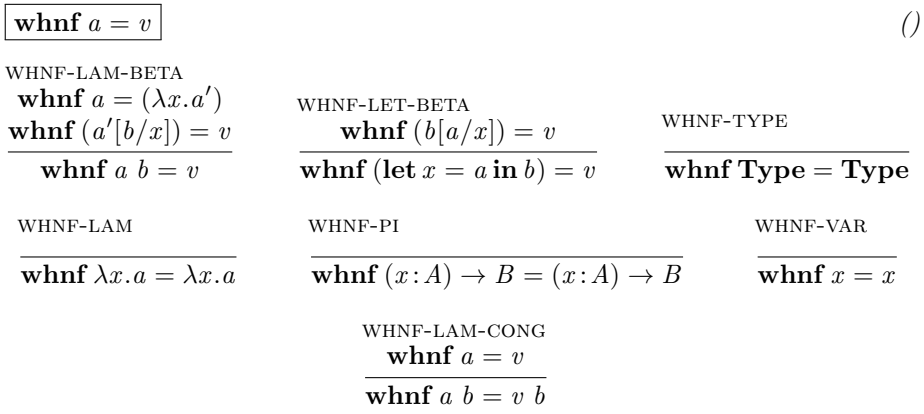

Figure 5: Weak-head normal form reduction

In this case, we are given a single type A and we need to know whether it is equivalent to some Π-type. Because Π-types are *head* forms, we can do this via reduction. This rule evaluates the type A to its head form, using the rules shown in Figure 5. If that form is a Π-type, then we can access its subcomponents.

Weak-head normal form The following judgement (shown in Figure 5)

$$\boxed{\textbf{whnf}\ a = v}$$

describes when a term a reduces to some result v in *weak-head normal form (whnf)*. For closed terms, these rules correspond to a big-step evaluation relation and produce a value. For open terms, the rules produce a term that is either headed by a variable or by some head form (i.e. λ or Π).

For example, the term

```
(λx.x) (λx.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, the terms

```
λy. (λx.x) (λx.x)
```

and

```
(y:Type) → (λx.x) Bool
```

are in weak head normal form. Furthermore, the term `x y` is also in weak head normal form, because, even though we don't know what the head form is, we cannot reduce the term any more.

Because evaluation may not terminate, this relation is semi-decidable. However, it is syntax-directed so we can readily express it as a Haskell function.

Using algorithmic equality In the `pi-forall` implementation, the function that corresponds to $\Gamma \vdash a \Leftrightarrow b$ has type

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

This function ensures that the two provided types are equal, or throws a type error if they are not.

Additionally, the `pi-forall` includes the function

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

that checks the given type to see if the given is equal to some Π-type of the form $(x:A_1) \to A_2$, and if so returns `x`, `A1` and `A2`.

This function is defined in terms of a helper function that implements the rules shown in Figure 5.

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

In `version2` of 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.

5.5 Implementing definitional equality (see Equal.hs)

The rules for $\Gamma \vdash a = b$ *specify* when terms should be equal, but they are not an algorithm. But how do we implement its algorithmic analogue $\Gamma \vdash a \Leftrightarrow b$?

There are several ways to do so.

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  -- full reduction, not just weak-head reduction
   nf2 <- reduce t2
   Unbound.aeq nf1 nf2
```

However, we can do better. Sometimes we can equate the terms without completely reducing them. Because reduction can be expensive (or even non-terminating) we'd like to only reduce as much as necessary.

Therefore, the implementation of `equate` has the following form.

```
equate t1 t2 = do
   if (Unbound.aeq t1 t1) then return () else do
```

```
nf1 <- whnf t1  -- reduce only to 'weak head normal form'
nf2 <- whnf t2
case (nf1,nf2) of
  (Lam bnd1, Lam bnd2) -> do
      -- ignore variable names, but use the same
      -- fresh name for both lambda bodies
      (_, b1, _, b2) <- Unbound.unbind2Plus bnd1 bnd2
       equate b1 b2
  (App a1 a2, App b1 b2) -> do
      equate a1 b1
      equateArg a2 b2

  ... -- all other matching head forms

  -- | head forms don't match throw an error
  (_,_) -> Env.err ...
```

Above, we first check whether the terms are already alpha-equivalent. If they are, we do not need to do anything else, we can return immediately (i.e. without throwing an error, which is what the function does when it decides that the terms are not equal). The next step is to reduce both terms to their weak-head normal forms. If two terms have different head forms, then we know that they must be different terms so we can throw an error. If they have the same head forms, then we can call the function recursively on analogous subcomponents until the function either terminates or finds a mismatch.

Why do we use weak-head reduction vs. full reduction?

- We can implement deferred substitutions for variables. Note that when comparing terms we need to have their 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, if we unfold recursive definitions inside functions before they are applied. In contrast, the definition based on `whnf` only unfolds recursive definitions when they stand in the way of equivalence, so avoids some infinite loops in the type checker.

  Note that we don't have a complete treatment of equality. There will always be terms that can cause `equate` to loop forever.

6 Dependent pattern matching and propositional equality

6.1 Refining rules for if and Σ-type elimination

Consider our elimination rules for if:

$$\text{T-IF-WEAK}\quad \frac{\Gamma \vdash a : \mathbf{Bool} \qquad \Gamma \vdash b_1 : A \qquad \Gamma \vdash b_2 : A}{\vdash \mathbf{if}\ a\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 : A}$$

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

$$\text{T-IF-FULL}\quad \frac{\Gamma \vdash a : \mathbf{Bool} \qquad \Gamma \vdash b_1 : A[\mathbf{True}/x] \qquad \Gamma \vdash b_2 : A[\mathbf{False}/x]}{\Gamma \vdash \mathbf{if}\ a\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 : A[a/x]}$$

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

```
-- function from booleans to types
T : Bool → Type
T = λb. if b then Unit else Bool

-- returns Unit when the argument is True
bar : (b : Bool) → T b
bar = λb . if b then () else True
```

It turns out that this rule is difficult to implement. It is not syntax-directed because A and x are not fixed by the syntax. Given $A[\mathbf{True}/x]$ and $A[\mathbf{False}/x]$ and $A[a/x]$ (or anything that they are definitionally equal to!) how can we figure out whether they correspond to each other?

So, we'll not be so ambitious in `pi-forall`. We'll only allow this refinement when the scrutinee is a variable, deferring to the weaker, non-refining typing rule for if in all other cases.

$$\text{T-IF}\quad \frac{\Gamma \vdash x : \mathbf{Bool} \qquad \Gamma \vdash b_1 : A[\mathbf{True}/x] \qquad \Gamma \vdash b_2 : A[\mathbf{False}/x]}{\Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 : A}$$

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

$$\text{C-IF}\quad \frac{\Gamma \vdash x \Rightarrow \mathbf{Bool} \qquad \Gamma \vdash b_1 \Leftarrow A[\mathbf{True}/x] \qquad \Gamma \vdash b_2 \Leftarrow A[\mathbf{False}/x]}{\Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 \Leftarrow A}$$

To implement this rule, we need only remember that x is **True** or **False**

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

when checking the individual branches of the if expression. [10]

We can modify the elimination rule for Σ-types similarly.

C-LETPAIR

$$\frac{\Gamma \vdash z \Rightarrow \{x : A_1 \mid A_2\} \qquad \Gamma, x : A_1, y : B_2 \vdash b \Leftarrow B[(x,y)/z] \qquad \Gamma \vdash B \Leftarrow \mathbf{Type}}{\Gamma \vdash \mathbf{let}\,(x,y) = z\,\mathbf{in}\,b \Leftarrow B}$$

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

```
fst : (A:Type) → (B : A → Type) → (p : { x2 : A |  B x2 }) → A
fst = λA B p. let (x,y) = p in x
```

But, weak Sigmas cannot define the second projection. 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 = λA B p. let (x,y) = p in y
```

(Try this out using version2 of the implementation and the Lec2.pi input file.)

6.2 Propositional equality

You started proving things right away in Coq or Agda 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 a propositional equality 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 evaluates to a value only when two types are equal.[11] In other words, the new value form **refl** has type $a = b$, as shown in rule T-REFL.

I-IF-ALT

$$\frac{\Gamma \vdash a \Rightarrow \mathbf{Bool} \qquad \Gamma \vdash b_1 \Rightarrow B_1 \qquad \Gamma \vdash b_2 \Rightarrow B_2}{\Gamma \vdash \mathbf{if}\,a\,\mathbf{then}\,b_1\,\mathbf{else}\,b_2 \Rightarrow \mathbf{if}\,a\,\mathbf{then}\,B_1\,\mathbf{else}\,B_2}$$

It has a nice symmetry—if expressions are typed by if types. 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:

E-IF-ETA

$$\frac{}{\Gamma \vdash \mathbf{if}\,a\,\mathbf{then}\,b\,\mathbf{else}\,b = b}$$

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.

[11] Recall that all types are inhabited by an infinite loop in `pi-forall`.

$\boxed{\Gamma \vdash a : A}$ *(Typing)*

$$\frac{\Gamma \vdash a = b}{\Gamma \vdash \mathbf{refl} : a = b}\ \text{T-REFL} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a = b : \mathbf{Type}}\ \text{T-EQ}$$

$$\frac{\Gamma \vdash a : A[a_1/x][\mathbf{refl}/y] \qquad \Gamma \vdash b : a_1 = a_2}{\Gamma \vdash \mathbf{subst}\ a\ \mathbf{by}\ b : A[a_2/x][b/y]}\ \text{T-SUBST} \qquad \frac{\Gamma \vdash a : \mathbf{True} = \mathbf{False}}{\Gamma \vdash \mathbf{contra}\ a : A}\ \text{T-CONTRA}$$

Figure 6: Propositional equality

Sometimes, you might see the rule written as follows:

$$\frac{}{\Gamma \vdash \mathbf{refl} : a = a}\ \text{T-REFL-ALT}$$

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

An equality type is well-formed when the terms on both sides of the equality have the same type. In other words, when it implements *homogeneous* equality, as shown in rule rule T-EQ.

The elimination rule for propositional equality is written **subst** a **by** b in `pi-forall`. This term allows us to convert the type of one expression to another, as shown in rule rule T-SUBST-SIMPLE below. In this rule, we can change the type of some expression a by replacing an occurrence of some term a_1 with an equivalent type.

$$\frac{\Gamma \vdash a : A[a_1/x] \qquad \Gamma \vdash b : a_1 = a_2}{\Gamma \vdash \mathbf{subst}\ a\ \mathbf{by}\ b : A[a_2/x]}\ \text{T-SUBST-SIMPLE}$$

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.

$$\frac{\Gamma \vdash b \Rightarrow B \qquad \mathbf{whnf}\ B = (x = a_2) \qquad \Gamma \vdash a \Leftarrow A[a_2/x]}{\Gamma \vdash \mathbf{subst}\ a\ \mathbf{by}\ b \Leftarrow A}\ \text{C-SUBST-LEFT}$$

$$\frac{\Gamma \vdash b \Rightarrow B \qquad \mathbf{whnf}\ B = (a_1 = x) \qquad \Gamma \vdash a \Leftarrow A[a_1/x]}{\Gamma \vdash \mathbf{subst}\ a\ \mathbf{by}\ b \Leftarrow A}\ \text{C-SUBST-RIGHT}$$

Note that this 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 the elimination form for propositional equality with dependent pattern matching as we did for booleans. This corresponds to the elimination rule rule T-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`).

As above, this rule (and the corresponding left rule) only applies when b is also a variable.

$$\frac{\Gamma \vdash y \Rightarrow B \qquad \mathbf{whnf}\ B = (a_1 = x) \qquad \Gamma \vdash a \Leftarrow A[a_1/x][\mathbf{refl}/y]}{\Gamma \vdash \mathbf{subst}\ a\ \mathbf{by}\ y \Leftarrow A}\ \text{C-SUBST-RIGHT-REFL}$$

One last addition: `contra`. If we can somehow prove a false statement, 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 rule T-CONTRA, shown in Figure 6.

6.2.1 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.

6.2.2 Homework (`pi-forall`: equality)

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

7 Irrelevance: the ∀ of `pi-forall`

Now, let's talk about erasure. In dependently typed languages, some arguments are "ghost" or "specificational" and only there for proofs. For efficient executables, we don't want to have to "run" these arguments, nor do we want them taking up space in data structures.

Functional languages do this all the time: they erase *type annotations* and *type* arguments before running the code. This erasure makes sense because of parametric polymorphic functions are not allowed to depend on types. The behavior of map must be the same no matter whether it is operating on a list of integers or a list of booleans.

In a dependently-typed language we'd like to erase types too. And proofs that are only there to make things type check. Coq does this by making a distinction between `Prop` and `Set`. Everything in `Set` stays around until run time, and is guaranteed not to depend on `Prop`.

We'll take another approach.

In `pi-forall` we have new kind of quantification, called "forall", that marks erasable arguments. We mark forall quantified arguments with brackets. For example, we can mark the type argument of the polymorphic identity function as erasable.

```
id : [x:Type] → (y : x) → x
id = λ[x] y. y
```

When we apply such functions, we'll put the argument in brackets too, so we remember that `id` is not really using that type.

```
t = id [Bool] True
```

However, we need to make sure that irrelevant arguments really are irrelevant. We wouldn't want to allow this definition:

```
id' : [x:Type] → [y:x] → x
id' = λ[x][y]. y
```

Here `id'` claims that its second argument is erasable, but it is not.

7.1 How do we rule this out in the type system?

We need to make sure that an irrelevant variable, like x above, is not "used" in the body of a λ-abstraction. How can we do so?

The approach we will take will seem a bit round about at first. But this approach is the most future-proof, for reasons that we discuss below.

The key idea is that we will mark variables in the context with an *epsilon* annotation, which is either $+$ or $-$. "Normal" variables, introduced by λ-expressions will always be marked as relevant ($+$). Only relevant variables can be used in terms and we revise the variable typing rule to require this annotation:

$$\frac{x :^{+} A \in \Gamma}{\Gamma \vdash x : A} \text{ T-EVAR}$$

An irrelevant abstraction introduces its variable into the context tagged with $-$. Because of this tag, this variable will be inaccessible in the body of the function.

$$\frac{\Gamma, x :^{-} A \vdash a : B \quad \Gamma^{-} \vdash A : \mathbf{Type}}{\Gamma \vdash \backslash[x].a : [x : A] \to B} \text{ T-ELAMBDA}$$

However, this variable should be available for use in *types* and other *irrelevant* parts of the term. To enable this use, we use a special context operation Γ^{-}, as in the second premise in rule T-ELAMBDA, and in the rule T-EAPP rule below.

$$\text{T-EAPP}\quad \frac{\Gamma \vdash a : [x : A] \to B \qquad \Gamma^{-} \vdash b : A}{\Gamma \vdash a[b] : B[b/x]}$$

This operation on the context, called *demotion*, converts all $-$ tags on variables to be $+$. It represents a shift in our perspective: the variables that were not visible before are now available after this context modification.

Finally, when checking irrelevant Π types, we mark the variable with $+$ when we add it to the context so that it may be used in the range of the Π type. Otherwise, we would not be able to verify the type of the polymorphic identity function above.

$$\text{T-EPI}\quad \frac{\Gamma \vdash A : \mathbf{Type} \qquad \Gamma, x :^{+} A \vdash B : \mathbf{Type}}{\Gamma \vdash [x : A] \to B : \mathbf{Type}}$$

7.2 Compile-time irrelevance

What does checking irrelevance buy us? Because we know that irrelevant arguments are not actually used by the function, we know that they can be *erased* prior to execution. We don't need this information to compute the answer, so why provide it?

An additional benefit is during equivalence checking. When deciding whether two terms are equal, we don't need to look at their irrelevant components.

$$\text{E-EAPP}\quad \frac{\Gamma \vdash a_1 = a_2}{\Gamma \vdash a_1[b_1] = a_2[b_2]}$$

We've been alluding to this the whole time, but now we'll come down to it. We're actually *defining* equality over just the computationally relevant parts of the term instead of the entire term. In `version3` of the implementation, note how the definition of `equate` ignores arguments that are tagged as 'irrelevant'. (And from the beginning, our system already ignores types that appear only in type annotations. Our justification for doing this is the same.)

Why is compile-time irrelevance important?

- faster comparison: don't have to look at the whole term when comparing for equality. Coq / Adga look at type annotations
- more expressive: don't have to *prove* that those parts are equal

For example, consider the example below. The function p has an irrelevant argument, so it must be a constant function. Another name for p might be a *phantom* type. Therefore it is sound to equate any two applications of p because we know that we will always get the same result.

```
irrelevance : (p : [i : Nat] → Type) → p [1] = p [2]
irrelevance = λp . Refl
```

However, note that casting is a relevant use of propositional equality. In the example below, `pi-forall` will prevent us from marking the argument `pf` as irrelevant.

```
proprel : [a : Type] → (pf : a = Bool) → (x : a) → Bool
proprel = λ[a] pf x .
  subst x by pf
```

The reason for this restriction is because the language includes non-termination. We don't know whether this proof is `Refl` or some infinite loop. So this argument must be evaluated to be sure that the two types are actually equal. If (somehow) we knew that the argument would always evaluate to `Refl` we could erase it.

8 Datatypes and Indexed Datatypes

Finally, we'll add datatypes with erasable arguments to `pi-forall`. The code to look at is the "complete" implementation in `full`.

Unfortunately, datatypes are both:

- Really important (you see them *everywhere* when working with languages like Coq, Agda, Idris, etc.)
- Really complicated (there are a *lot* of details). In general, datatypes subsume booleans, Σ-types, propositional equality, and can carry irrelevant arguments. So they combine all of the complexity of the previous sections in one construct.

Unlike the previous sections, where we could walk through all of the details of the specification of the type system, not to mention its implementation, we won't be able to do that here. There is just too much! The goal of this part is to give you enough information so that you can pick up the Haskell code and understand what is going on.

Even then, realize that the implementation that I'm giving you is not the complete story! Recall that we're not considering termination. That means that we can think about eliminating datatypes merely by writing recursive functions; without having to reason about whether those functions terminate. Coq, Agda and Idris include a lot of machinery for this termination analysis, and we won't cover any of it.

We'll work up the general specification of datatypes piece-by-piece, generalizing from features that we already know to more difficult cases. We'll start with "simple" datatypes, and then extend them with both parameters and indices.

8.1 "Dirt simple" datatypes

Our first goal is simple. What do we need to get the simplest examples of non-recursive and recursive datatypes working? By this I mean datatypes that you might see in Haskell or ML, such as `Bool`, `Void` and `Nat`.

8.1.1 Booleans

For example, one homework assignment was to implement booleans. Once we have booleans then we can

```
data Bool : Type where
  True
  False
```

In the homework assignment, we used `if` as the elimination form for boolean values.

```
not : Bool → Bool
not = λb . if b then False else True
```

For uniformity, we'll have a common elimination form for all datatypes, called `case` that has branches for all cases. (We'll steal Haskell syntax for case expressions, including layout.) For example, we might rewrite `not` with case like this:

```
not : Bool → Bool
not = λb .
  case b of
      True → False
      False → True
```

8.1.2 Void

The simplest datatype of all is one that has no constructors!

```
data Void : Type where {}
```

Because there are no constructors, the elimination form for values of this type doesn't need any cases!

```
false_elim : (A:Type) → Void → A
false_elim = λA v . case v of {}
```

Void brings up the issue of *exhaustiveness* in case analysis. Can we tell whether there are enough patterns so that all of the cases are covered? This is something that our implementation should be able to do.

8.1.3 Nat

Natural numbers include a data constructor with an argument. For simplicity in the parser, those parentheses must be there.

```
data Nat : Type where
  Zero
  Succ of (Nat)
```

In case analysis, we can give a name to that argument in the pattern.

```
is_zero : Nat → Bool
is_zero = λx . case x of
  Zero → True
  Succ n → False
```

8.1.4 Dependently-typed data constructor args

Now, even in our "dirt simple" system, we'll be able to encode some new structures, beyond what is available in functional programming languages like Haskell and ML. These structures won't be all that useful yet, but as we add parameters and indices to our datatypes, they will be. For example, here's an example of a datatype declaration where the data constructors have dependent types.

```
data SillyBool : Type where
  ImTrue of (b : Bool) (_ : b = True)
  ImFalse of (b: Bool)  (_ : b = False)
```

8.2 Implementing the type checker for "dirt simple" datatypes

Datatype declarations, such as `data Bool`, `data Void` or `data Nat` extend the context with new type constants (aka type constructors) and new data constructors. It is as if we had added a bunch of new typing rules to the type system, such as:

$\boxed{\Gamma \vdash a : A}$ *(Typing)*

$$\textsc{t-Nat}\ \frac{}{\Gamma \vdash \mathbf{Nat} : \mathbf{Type}} \qquad \textsc{t-Void}\ \frac{}{\Gamma \vdash \mathbf{Void} : \mathbf{Type}} \qquad \textsc{t-Zero}\ \frac{}{\Gamma \vdash \mathbf{Zero} : \mathbf{Nat}}$$

$$\textsc{t-Succ}\ \frac{\Gamma \vdash n : \mathbf{Nat}}{\Gamma \vdash \mathbf{Succ}\ n : \mathbf{Nat}} \qquad \textsc{t-ImTrue}\ \frac{\Gamma \vdash a_1 : \mathbf{Bool} \quad \Gamma \vdash a_2 : a_1 = \mathbf{True}}{\Gamma \vdash \mathbf{ImTrue}\ a_1\ a_2 : \mathbf{SillyBool}}$$

In the general form, a *simple* data type declaration includes a name and a list of data constructors.

```
data T : Type where
  K1                -- no arguments
  K2 of (A)         -- single arg of type A
  K3 of (x:A)       -- also single arg of type A, called x for fun
  K4 of (x:A)(y:B)  -- two args, the type of B can mention A
  K5 of (x:A)[x = a] -- one arg, plus constraint about that arg
```

In fact, each data constructor takes a special sort of list of arguments that we call a *telescope*. (The word telescope for this structure was coined by de Bruijn [**?**] to describe the scoping behavior of this structure. The scope of each variable overlaps all of the subsequent ones, nesting like an expandable telescope.)

We can represent this structure in our implementation by adding a new form of declaration (some parts have been elided compared to `full`, so we're building up to that version.)

```
-- | type constructor names
type TCName = String

-- | data constructor names
type DCName = String

data Decl =
    -- | Declaration for the type of a term ‘x : A‘
    TypeSig Sig
  | -- | The definition of a particular name ‘x = a‘
    Def TName Term
    ...

  | -- | Datatype definition  ‘data T = ...‘
    Data TCName [ConstructorDef]

-- | A Data constructor has a name and a telescope of arguments
data ConstructorDef = ConstructorDef DCName Telescope

-- | A telescope is a list of type declarations and definitions
newtype Telescope = Telescope [Decl]
```

For example, a declaration for the `Bool` type would be the following.

```
boolDecl :: Decl
boolDecl = Data "Bool" [ConstructorDef "False" [],
                        ConstructorDef "True"  [] ]
```

8.3 Checking (simple) data constructor applications

When we have a datatype declaration, that means that new data type T of type **Type** will be added to the context. Furthermore, the context should record all of the type constructors for that type, K_i, as well as the telescope, written Δ_i, for that data constructor. This information will be used to check terms that are the applications of data constructors. For simplicity, we'll assume that data constructors must be fully applied to all of their arguments.

$\Gamma \vdash \overline{a} \Leftarrow \Delta$	`tcArgTele`	Type check a list of arguments against a telescope
$\Gamma \vdash p : A \Rightarrow \Delta$	`declarePat`	Create telescope containing all of the variables from the pattern
$\vdash a \sim b \Rightarrow \Delta$	`unify`	Compare two terms to create a list of definitions
$\Delta[a/x]$	`doSubst`	Substitute through a telescope
$\Delta_1[\overline{a}/\Delta_2]$	`substTele`	Substitute a list of args for the variables declared in a telescope

Figure 7: Functions for checking datatypes and case expressions

So our typing rule looks a little like this. We have $\overline{a}$ as representing the list of arguments for the data constructor K.

$$\text{I-DCON-SIMPLE}\quad \frac{K : \Delta \to T \ \in \Gamma \qquad \Gamma \vdash \overline{a} \Leftarrow \Delta}{\Gamma \vdash K\,\overline{a} \Rightarrow T}$$

We need to check that list against the telescope for the constructor, using the judgment $\Gamma \vdash \overline{a} \Leftarrow \Delta$.

In this judgement, each argument must have the right type. Furthermore, because of dependency, we substitute that argument for the variable in the rest of the telescope.

$$\text{TELE-SIG}\quad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash \overline{a} \Leftarrow \Delta[a/x]}{\Gamma \vdash a\,\overline{a} \Leftarrow x : A, \Delta}$$

Furthermore, we also substitute when the telescope contains definitions.

$$\text{TELE-DEF}\quad \frac{\Gamma \vdash \overline{a} \Leftarrow \Delta[a/x]}{\Gamma \vdash \overline{a} \Leftarrow x = a, \Delta}$$

When we get to the end of the argument list (i.e. there are no more arguments) we should also get to the end of the telescope.

$$\text{TELE-NIL}\quad \frac{}{\Gamma \vdash \Leftarrow}$$

In `TypeCheck.hs`, the function `tcArgTele` essentially implements this judgement. (For reasons that we explain below, we have a special type `Arg` for the arguments to the data constructor.)

```
tcArgTele :: [Arg] -> Telescope -> TcMonad [Arg]
```

This function relies on the substitution function for telescopes, written $\Delta[a/x]$ in the rules above:

```
doSubst :: [(TName, Term)] -> [Assn] -> TcMonad [Assn]
```

8.4 Eliminating dirt simple datatypes

In your homework assignment, we used a special `if` to eliminate boolean types. Now, we'd like to replace that with the more general `case` expression, of the form **case** a **of** $\{\ \overline{p_i \rightarrow a_i}^{\,i}\ \}$ that works with any form of datatype. What should the typing rule for that sort of expression look like?

C-CASE-SIMPLE

$$\frac{\begin{array}{c}\Gamma \vdash a \Rightarrow A \\ \textbf{whnf}\ A = T \qquad \overline{\Gamma \vdash p_i : T \Rightarrow \Delta_i}^{\,i} \qquad \overline{\vdash a \sim p_i \Rightarrow \Delta_i'}^{\,i} \\ \overline{\Gamma, \Delta_i, \Delta_i' \vdash a_i : A}^{\,i} \qquad \textbf{branches exhaustive}\end{array}}{\Gamma \vdash \textbf{case}\ a\ \textbf{of}\ \{\ \overline{p_i \rightarrow a_i}^{\,i}\ \} \Leftarrow A}$$

Mathematically, this rule checks that the scrutinee has the type of some datatype T. Then, for each case of the pattern match, this rule looks at the pattern and its type to calculate a telescope containing declarations of all of the variables bound in that pattern. That telescope is added to the context to type check each branch. Furthermore, when checking the type of each branch, we can refine that type because we know that the scrutinee is equal to the pattern.

How do we implement this rule in our language? The general strategy is as follows:

1. Infer type of the scrutinee a (`inferType`)
2. Make sure that the inferred type is some type constructor T (`ensureTCon`)
3. For each case alternative $p_i \rightarrow a_i$:
   - Create the declarations Δ_i for the variables in the pattern (`declarePat`)
   - Create a list of definitions Δ_i' that follow from unifying the scrutinee a with the pattern p_i (`unify`)
   - Check the body of the case a_i in the extended context against the expected type A (`checkType`)
4. Make sure that the patterns in the cases are exhaustive (`exhausivityCheck`)

8.5 Datatypes with parameters

The first extension of the above scheme is for *parameterized datatypes*. For example, in pi-forall we can define the `Maybe` type with the following declaration. The type parameter for this datatype `A` can be referred to in any of the telescopes for the data constructors.

```
data Maybe (A : Type) : Type where
    Nothing
     Just of (A)
```

Because this is a dependently-typed language, the variables in the telescope can be referred to later in the telescope. For example, with parameters, we can implement Sigma types as a datatype, instead of making them primitive:

```
data Sigma (A: Type) (B : A → Type) : Type
    Prod of (x:A) (B)
```

The general form of datatype declaration with parameters includes a telescope for the type constructor, as well as a telescope for each of the data constructors.

```
data T D : Type where
   Ki of Di
```

That means that when we check an occurrence of a type constructor, we need to make sure that its actual arguments match up the parameters in the telescope. For this, we can use the argument checking judgement above.

$$\text{I-TCON}\quad \frac{T : \Delta \to \mathbf{Type} \in \Gamma \qquad \Gamma \vdash \overline{a} \Leftarrow \Delta}{\Gamma \vdash T\,\overline{a} \Rightarrow \mathbf{Type}}$$

We modify the typing rule for data constructors by marking the telescope for type constructor in the typing rule, and then substituting the actual arguments from the expected type:

$$\text{C-DCON}\quad \frac{K : \Delta_1 \to \Delta_2 \to T \ \in \Gamma \qquad \Gamma \vdash \overline{a} \Leftarrow \Delta_2[\overline{b}/\Delta_1]}{\Gamma \vdash K\,\overline{a} \Leftarrow T\,\overline{b}}$$

For example, if we are trying to check the expression `Just True`, with expected type `Maybe Bool`, we'll first see that `Maybe` requires the telescope `(A : Type)`. That means we need to substitute `Bool` for `A` in `(_ : A)`, the telescope for `Just`. That produces the telescope `(_ : Bool)`, which we'll use to check the argument `True`.

In `TypeCheck.hs`, the function

```
substTele :: Telescope -> [ Term ] -> Telescope -> TcMonad Telescope
```

implements this operation of substituting the actual data type arguments for the parameters.

Note that by checking the type of data constructor applications (instead of inferring them) we don't need to explicitly provide the parameters to the data constructor. The type system can figure them out from the provided type.

Also note that checking mode also enables *data constructor overloading*. In other words, we can have multiple datatypes that use the same data constructor. Having the type available allows us to disambiguate.

For added flexibility we can also add code to *infer* the types of data constructors when they are not actually parameterized (and when there is no ambiguity due to overloading).

8.6 Datatypes with indices

The final step is to index our datatypes with constraints on the parameters. Indexed types let us express inductively defined relations, such as `beautiful` from Software Foundations.

```
Inductive beautiful : nat -> Prop :=
  b_0 : beautiful 0
| b_3 : beautiful 3
| b_5 : beautiful 5
| b_sum : forall n m, beautiful n -> beautiful m -> beautiful (n+m).
```

Even though `beautiful` has type `nat -> Prop`, we call `nat` this argument an index instead of a parameter because it is determined by each data constructor. It is not used uniformly in each case.

In `pi-forall`, we'll implement indices by explicitly *constraining* parameters. These constraints will just be expressed as equalities written in square brackets. In other words, we'll define `beautiful` this way:

```
data Beautiful (n : Nat) : Type where
  B0 of [n = 0]
  B3 of [n = 3]
  B5 of [n = 5]
  Bsum of (m1:Nat)(m2:Nat)(Beautiful m1)(Beautiful m2)[m = m1+m2]
```

Constraints can appear anywhere in the telescope of a data constructor. However, they are not arbitrary equality constraints—we want to consider them as deferred substitutions. So therefore, the term on the left must always be a variable.

These constraints interact with the type checker in a few places:

- When we use data constructors we need to be sure that the constraints are satisfied, by appealing to definitional equality when we are checking arguments against a telescope (in `tcArgTele`).

$$\frac{\Gamma \vdash \overline{a} \Leftarrow \Delta[a/x]}{\Gamma \vdash \overline{a} \Leftarrow x = a, \Delta}\ \text{TELE-DEF}$$

- When we substitute through telescopes (in `doSubst`), we may need to rewrite a constraint `x = b` if we substitute for `x`.
- When we add the pattern variables to the context in each alternative of a case expression, we need to also add the constraints as definitions (see `declarePats`).

For example, if we check an occurrence of `B3`, i.e.

```
threeIsBeautiful : Beautiful 3
threeIsBeautiful = B3
```

this requires substituting `3` for `n` in the telescope `[n = 3]`. That produces an empty telescope.

8.6.1 Homework: Parameterized datatypes and proofs: logic

Translate the definitions and proofs in Logic chapter of Software Foundations to `pi-forall`. See Logic.pi for a start.

8.6.2 Homework: Indexed datatypes: finite numbers in `FinHw.pi`

The module `FinHw.pi` declares the type of numbers that are drawn from some bounded set. For example, the type `Fin 1` only includes 1 number (called Zero), `Fin 2` includes 2 numbers, etc. More generally, `Fin n` is the type of all natural numbers smaller than `n`, i.e. of all valid indices for lists of size `n`.

```
data Fin (n : Nat) : Type where
  Zero of (m:Nat)[n = Succ m]
  Succ of (m:Nat)[n = Succ m] (Fin m)
```

The file FinHw.pi includes a number of definitions that use these types. Two of these definitions are marked with `TRUSTME`. Replace these expressions with appropriate definitions.

8.7 Erasure and datatypes

What about putting it in data structures? We should be able to define datatypes with "specificational arguments". For example, see Vec.pi.

Note: we can only erase *data* constructor arguments, not types that appear as arguments to *type* constructors. (Parameters to type constructors must always be relevant, they determine the actual type.) On the other hand, datatype parameters are never relevant to data constructors—we don't even represent them in the abstract syntax.

8.7.1 Homework: Erasure and Indexed datatypes: finite numbers in `FinHw.pi`

Now take your code in `FinHw.pi` and see if you can mark some of the components of the `Fin` datatype as erasable.

9 Where to go for more

9.1 Related work for `pi-forall`

Section 2: Core Dependent Types

- Barendregt, Lambda Calculi with Types
- Cardelli, A polymorphic lambda calculus with Type:Type `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-10.pdf`

- Augustsson, Cayenne – a Language With Dependent Types `http://dl.acm.org/citation.cfm?id=289451`

Section 3: Bidirectional type checking

- Pierce and Turner
- Peyton Jones, Vytiniotis, Weirich, Shields. Practical type inference for arbitrary-rank types `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/putting.pdf`
- Christiansen, Tutorial on Bidirectional Typing. `https://www.davidchristiansen.dk/tutorials/bidirectional.pdf`
- Dunfield and Krishnaswami, Bidirectional Typing `https://www.cl.cam.ac.uk/~nk480/bidir-survey.pdf`

Section 5: Normalization by evaluation

- Andreas Abel's habilitation thesis.
- Daniel Gratzer, NBE for MLTT `https://github.com/jozefg/nbe-for-mltt`

Section 6: Dependently-typed pattern matching

- Coq pattern matching: Coq User manual
- Agda pattern matching: Ulf Norell's dissertation
- Haskell GADTs: Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann, OutsideIn(X): Modular type inference with local assumptions

Section 7: Compile-time and runtime irrelevance * Pfenning, Abel and Scherer * ICC, Mishra-Linger and Sheard * McBride, Atkey * DDC

9.2 Other tutorials on the implementation of dependent type systems

- A. Löh, C. McBride, W. Swierstra, *A tutorial implementation of a dependently typed lambda calculus* [LMS10].

  "LambdaPi". Implementation in Haskell. Starts with simply-typed lambda calculus, uses locally nameless representation and bidirectional typing, extends to core type system with type-in-type, implements NBE, then adds natural numbers and vectors.

- Andrej Bauer, *How to implement dependent type theory* [Bau12]

Series of blog posts. Implementation in OCaml. Infinite hierarchy of type universes. Uses named representation and generates fresh names during substitution and alpha-equality. Uses full normalization to implement definitional equality. Then in second version revises to use NBE. The third version revises to use de Bruijn indices, explicit substitutions and then switches back to weak-head normalization.

- Tiark Rompf, *Implementing Dependent Types.* [Rom20].

  Uses Javascript. Implements a core dependent type theory using HOAS (tagless final) and Normalization by evaluation.

- Lennart Augustsson. *Simple, Easier!* [Aug07]

  Implemented in Haskell. Goal is simplest implementation. Uses string representation of variables and weak-head normalization for implementation of equality. Implementation of Barendregt's lambda cube.

- Coquand, Kinoshita, Nordstrom, Takeyama. *A simple type-theoretic language: Mini-TT* [CKNT09]

  "Mini-TT". Implemented in Haskell. Includes sigma types, void, unit and sums, but not indexed datatypes (or propositional equality). Uses NbE for conversion checking. Does not enforce termination.

- Andras Kovacs, Elaboration Zoo `https://github.com/AndrasKovacs/elaboration-zoo/`

- Brigitte Pientka `https://www.cs.mcgill.ca/~bpientka/papers/recon-jfp.pdf`

References

[Aug07] Lennart Augustsson. Simpler, easier!, 2007. Available from `https://augustss.blogspot.com/2007/10/simpler-easier-in-recent-paper-simply.html`.

[Bau12] Andrej Bauer. How to implement dependent type theory, November 2012. Available from `http://math.andrej.com/2012/11/08/how-to-implement-dependent-type-theory-i/`.

[CKNT09] Thierry Coquand, Yoshiki Kinoshita, Bengt Nordström, and Makoto Takeyama. A simple type-theoretic language: Mini-tt. In Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin, editors, *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, page 139–164. Cambridge University Press, 2009.

[Har16] Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, USA, 2nd edition, 2016.

[LMS10] Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundam. Inform.*, 102:177–207, 01 2010. Available from `http://www.andres-loeh.de/LambdaPi/`.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[Rom20] Tiark Rompf. Implementing dependent types, December 2020. Available from `https://tiarkrompf.github.io/notes/?/dependent-types/`.

[SNO+07] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, page 1–12, New York, NY, USA, 2007. Association for Computing Machinery.

[Ste17] Guy L. Steele. It's time for a new old language. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, page 1, New York, NY, USA, 2017. Association for Computing Machinery.