Implementing Dependent Types in `pi-forall`

Stephanie Weirich

July 7, 2023

1 Overview and Goals

These lecture notes describe the design of a minimal 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 July 2023 and derived from earlier lectures from summer schools during 2022, 2014 and 2013.

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

Furthermore, these 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` on branch `2023`. As you study these notes, I encourage you to download this repository, read through its source code, and experiment with it. Installation instructions are available with the repository.

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. This implementation itself is available in separate versions, each found in separate subdirectories of the repository.

Key feature	`pi-forall` subdirectory	Section
Core system	`version1`	Sections 3, 4, and 5
Equality	`version2`	Sections 6 and 7
Irrelevance	`version3`	Section 8
Datatypes	`full`	Sections 2 and 9

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

These implementations build on each other (each is an extension of the previous version) 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. The directory `main` is the source of all of these implementations and contains the markup needed to generate the four versions.

- Section 2 starts with some examples of the `full pi-forall` language so that you can see how the pieces fit together.

- Section 3 presents the mathematical specification of the core `pi-forall` type system including its concrete syntax (as found in `pi-forall` source files), abstract syntax (represented with a Haskell datatype), and core typing rules (written using standard mathematical notation). This initial type system is simple and declarative. In other words, it *specifies* what terms should type check, but cannot be directly implemented.

- Section 4 reformulates the typing rules so that they are *syntax-directed* and correspond to a type-checking algorithm. The key idea of this section is to recast the typing rules as a *bidirectional type system*.

- Section 5 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 4. This section also shows how the `Unbound` library can assist with the implementation of variable binding and automatically derive operations for capture-avoiding substitution and alpha-equivalence. Finally, the section 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 6 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 7 shows how to extend `pi-forall` with a simple form of dependent pattern matching. We look at this feature in two different ways: First how to reflect the gain in information about boolean values in each branch of an `if` expression. Second, how to reflect the equality represented via propositional equality.

- Section 8 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 9 introduces arbitrary datatypes and generalizes dependent pattern matching. This section combines the features introduced in the previous sections (Sections 6, 7, and 8) into a single unified language feature.

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

For a broader view, section 10 includes a discussion of alternative tutorials and describes how this one differs from other approaches. The key differences are summarized below:

- 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 recursion.
- 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.
- For simplicity, `pi-forall` does not attempt to infer missing 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, these operations can be automatically derived. A more common approach is to use de Bruijn indices.
- Recent work on cubical type theory, higher-inductive types, and univalence is not covered here. Indeed, the rules for dependent pattern matching that we present are incompatible with such extensions.

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 judgment holds.
2. Exposure to the design choices of dependently-typed languages. In this respect, the 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, Section 10 includes pointers to related work if you want to go deeper.

3. Experience with the Haskell programming language. I think Haskell is 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.[1] Building these notes requires Ott [SNO$^+$07], a tool specifically tailored for typesetting type systems and mathematical specifications of programming languages.

2 A taste of `pi-forall`

Before diving into the design of the `pi-forall` language, we will start with a few motivating examples for dependent types. This way, you will get to learn a bit about the syntax of `pi-forall` and anticipate the definitions that are to come.

One example that dependently-typed languages are really good at is tracking the length of lists.

In `pi-forall`, we can define a datatype for length-indexed lists (often called vectors) as follows.[2]

```
data Vec (A : Type) (n : Nat) : Type where
  Nil of [n = Zero]
  Cons of [m:Nat] (A) (Vec A m) [n = Succ m]
```

The type `Vec A n` has two parameters: `A` the type of elements in the list and `n`, a natural number indicating the length of the list. This datatype also has two constructors: `Nil` and `Cons`, corresponding to empty and non-empty lists. The `Cons` constructor has three arguments, a number `m`, the element at the head of the list (of type `A`), and the tail of the list (of type `Vec A m`). Both constructors also include *constraints* that must be satisfied when they are used. The `Nil` constructor constrains the length parameter to be `Zero` and the `Cons` constructor constrains it to be one more than the length of the tail of the list.

In the definition of `Cons`, the `m` parameter is the length of the tail of the list. This parameter is written in square brackets to indicate that it should be *irrelevant*: this value is for type checking only and functions should not use it.

For example, we can use `pi-forall` to check that the list below contains exactly three boolean values.

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

[2] In the `full` implementation, you can find these definitions in the file `pi/Vec.pi`.

```
v3 : Vec Bool 3
v3 = Cons [2] True (Cons [1] False (Cons [0] False Nil))
```

Above, `pi-forall` never infers arguments, so we must always supply the length of the tail whenever we use `Cons`.

We can also map over length-indexed lists.

```
map : [A:Type] → [B:Type] → [n:Nat] → (A → B) → Vec A n → Vec B n
map = λ[A][B][n] f v.
  case v of
    Nil → Nil
    Cons [m] x xs → Cons [m] (f x) (map[A][B][m] f xs)
```

As `pi-forall` doesn't infer arguments, the map function needs to take `A`, `B` and `n` explicitly at the beginning of the function definition, and they need to be provided as arguments to the recursive call.

The type checker helps us avoid errors in the definition of `map`. Like most functional languages, if we had forgotten to call `f` and instead included `x` in the `Cons`, then `map` would produce a vector containing elements of type `A` instead of `B`. The type checker would flag this as an error. Similarly, if we had replaces the `Cons` branch with `Nil` or had forgotten to `Cons f x` to the result of the recursive call in this branch, the resulting vector would be the wrong length. The type checker would catch this error too!

The `pi-forall` language does not include the sophisticated type inference algorithms found in many other languages, that can figure out some of the arguments provided to a function automatically. As a result, when we call map, we also need to supply the appropriate parameters for the function and the vector that we are giving to `map`. For example, to map the boolean `not` function, of type `Bool → Bool`, we need to provide these two `Bool`s along with the length of the vector.

```
v4 : Vec Bool 3
v4 = map [Bool][Bool][3] not v3
```

Because we are statically tracking the length of vectors, we can write a safe indexing operation. We index into the vector using an index where the type bounds its range. Below, the `Fin` type has a parameter that states what length of vector it is appropriate for.

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

For example, the type `Fin 3` has exactly three values:

```
x1 : Fin 3
x1 = Succ [2] (Succ [1] (Zero [0]))

x2 : Fin 3
x2 = Succ [2] (Zero [1])
```

```
x3 : Fin 3
x3 = Zero [2]
```

With the `Fin` type, we can safely index vectors. The following function is guaranteed to return a result because the index is within range.

```
nth : [a:Type] → [n:Nat] → Vec a n → Fin n → a
nth = λ[a][n] v f. case f of
   Zero [m] → case v of
         Cons [m'] x xs → x
   Succ [m] f' → case v of
         Cons [m'] x xs → nth [a][m] xs f'
```

Note how, in the case analysis above, neither `case` requires a branch for the `Nil` constructor. The `pi-forall` type checker can verify that the `Cons` case is exhaustive and that the `Nil` case would be inaccessible. In these cases, the type checker notes that the patterns for the missing branches don't make sense: constructing the `Nil` pattern would require satisfying the constraint that `Zero = Succ m`, which is impossible.

You may have noticed that some of the arguments to functions and constructors in the examples above are enclosed in square brackets. These arguments are *irrelevant* in `pi-forall`. What this means is that the compiler can erase these arguments before running the program and can ignore these arguments when comparing types for equality. (When arguments are marked in this way, the type checker must ensure that this erasure is sound and the argument is never actually used in a meaningful way.) Similarly, the constraints that are part of datatype definitions, such as `n = Succ m` above, are enclosed in square brackets because there is no runtime proof of the equality stored with the data constructor.

2.1 Homework: Check out `pi-forall`

The files in the directory `full/pi/` demonstrate the full power of the `pi-forall` language. Take a little time to familiarize yourself with some of these modules (such as `Fin.pi` and `Vec.pi`) and compare them to similar ones that you might have seen in Agda, Haskell, or elsewhere.

To run the type checker on these modules, make sure that you have first compiled the implementation using `stack build` at the terminal. Then, to run `pi-forall` on a source file, such as `Fin.pi`, you can issue the command `stack exec -- pi-forall Fin.pi` in the `full/pi` subdirectory. If the file type checks, you will see the contents of the file displayed in the terminal window. Otherwise, you will see an error message from the type checker.

For a more extensive series of examples, start with `pi/Lambda0.pi` for an interpreter for a small lambda calculus and then compare it with the implementations in `pi/Lambda1.pi` and `pi/Lambda2.pi`. These two versions index the expression type with the scope depth (a natural number) and the expression type to eliminate run-time scope and type errors from the execution of the interpreter.

2.1.1 Homework: Church and Scott encodings

The file `full/pi/NatChurch.pi` is a start at a Church encoding of natural numbers. Replace the `TRUSTME`s in this file so that it compiles. For example, one task in this file is to define a predecessor function. By replacing the `TRUSTME` with `Refl` below, you will be able to force the type checker to normalize both sides of the equality. The code will only type check if you have defined `pred` correctly (on this example).

```
test_pred : pred two = one
test_pred = TRUSTME -- replace with Refl
```

3 A Simple Core Language

Now let's turn to the design and implementation of the `pi-forall` language itself. We'll start with a small core, and then incrementally add features throughout the rest of this tutorial.

Consider this simple dependently-typed lambda calculus, shown below. What should it contain? At the bare minimum we start with the following five forms:

$$
\begin{array}{lcll}
a, b, A, B & ::= & x & \text{variables} \\
 & & \lambda x.a & \text{lambda expressions (anonymous functions)} \\
 & & a\ b & \text{function applications} \\
 & & (x\!:\!A) \to B & \text{dependent function type, aka } \Pi\text{-types} \\
 & & \mathbf{Type} & \text{the `type' of types}
\end{array}
$$

As in many dependently-typed languages, we have the *same* syntax for both expressions and types. For clarity, the convention is to use lowercase letters (a,b) for expressions and uppercase letters (A,B) for types.

Note that λ and Π above are *binding forms*. They bind the variable x in a and B respectively. In dependent function types $(x : A) \to B$, if x does not appear free in B, it is also convention to write the type as $A \to B$.

3.1 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.

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

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

$$\textsc{t-var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{t-lambda}\ \frac{\begin{array}{c}\Gamma, x{:}A \vdash a : B \\ \Gamma \vdash A : \mathbf{Type}\end{array}}{\Gamma \vdash \lambda x.a : (x{:}A) \to B} \qquad \textsc{t-app}\ \frac{\begin{array}{c}\Gamma \vdash a : (x{:}A) \to B \\ \Gamma \vdash b : A\end{array}}{\Gamma \vdash a\ b : B[b/x]}$$

$$\textsc{t-pi}\ \frac{\begin{array}{c}\Gamma \vdash A : \mathbf{Type} \\ \Gamma, x{:}A \vdash B : \mathbf{Type}\end{array}}{\Gamma \vdash (x{:}A) \to B : \mathbf{Type}} \qquad \textsc{t-type}\ \frac{}{\Gamma \vdash \mathbf{Type} : \mathbf{Type}}$$

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. We will do this in Section 5 using a freshness monad from the `Unbound` library.

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 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.

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

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.

Note that the rule rule T-LAMBDA makes subtle use of the fact that binding variables can be freely renamed to alpha-equivalent versions. In the conclusion of the rule, the x in the body of the lambda abstraction a, is a different binder from the x in the body of the function type B. Both of these terms can be modified so that these two x do not have to match. But, above the line, there is only a single x that appears free in both a and B! If we wanted to be explicit about what is going on, we could also write the rule with an explicit substitution:

$$\begin{array}{l}\text{T-ALTLAMBDA} \\ \dfrac{\begin{array}{c}\Gamma, x{:}A \vdash a : B[y/x] \\ \Gamma \vdash A : \mathbf{Type}\end{array}}{\Gamma \vdash \lambda x.a : (y{:}A) \rightarrow B}\end{array}$$

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

Most type theorists do not bother with this version and prefer the one that reuses the same binder. We will do the same here, and in other similar rules. However, we will also have to be careful about how we implement this rule in the implementation of our type checker!

More typing rules: Types Observe in the typing derivation above that the rule for typing λ-expressions has a 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.

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

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 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 the 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".

```
true : (A:Type) → A → A → A
```

[5] Note that this rule makes our language inconsistent as a logic, as it can encode a logical paradox. The `full` implementation includes a demonstration of this paradox in the file `Hurkens.pi`.

```
true = λx. λy. λz. y

false : (A:Type) → A → A → A
false = λx. λy. λz. z
```

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

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

Example: void type The type `(x:Type) → x`, called "void" is usually an *empty type* in polymorphic languages. Observe that this function takes a type `x:Type`, but never takes any expressions of type `x`. Thus, there is no way to return a value of any type without some sort of type case (not part of `pi-forall`).

However, because `pi-forall` does not enforce termination, we can define a value that has this type.

```
loop : (x:Type) → x
loop = λx . loop x
```

Typing invariant Overall, a property that we want for our type system is that if a term type checks, then its *type* will also type check. More formally, to state this property, we first need to say what it means to check all types in a context.

Definition 3.1 (Context type checks)**.** Define $\boxed{\vdash \Gamma}$ with the following rules.

$$\overset{\textsc{g-nil}}{\frac{}{\vdash \emptyset}} \qquad\qquad \overset{\textsc{g-cons}}{\frac{\Gamma \vdash A : \textbf{Type} \qquad \vdash \Gamma}{\vdash \Gamma, x : A}}$$

With this judgement, we can state the following property of our type system.

Lemma 3.1 (Regularity)**.** If $\Gamma \vdash a : A$ and $\vdash \Gamma$, then $\Gamma \vdash A : \textbf{Type}$

Making sure that this property holds for the type system is the motivation for the $\Gamma \vdash A : \textbf{Type}$ premise in rule T-LAMBDA.

4 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 have 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, which 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 us 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.

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 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 will 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.

4.1 A bidirectional type system

Let us redefine the system using two judgments. The first one is similar to the judgment that we saw above, and we will call it type *inference*.[7] This judgment will be paired with (and will depend on) a second judgment, called type *checking*, that takes advantage of known type information, such as the annotations on top-level definitions.

We express these judgments using the notation defined in Figure 3 and implement them in Haskell using the mutually-recursive functions `inferType` and

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

[7] The term *type inference* usually refers to much more sophisticated deduction of an expression's type, in the context of much less information encoded in the syntax of the language. We are 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)*

I-VAR

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

I-APP-SIMPLE

$$\frac{\begin{array}{c}\Gamma \vdash a \Rightarrow (x{:}A) \to B \\ \Gamma \vdash b \Leftarrow A\end{array}}{\Gamma \vdash a\ b \Rightarrow B[b/x]}$$

I-PI

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

I-TYPE

$$\frac{}{\Gamma \vdash \mathbf{Type} \Rightarrow \mathbf{Type}}$$

I-ANNOT

$$\frac{\begin{array}{c}\Gamma \vdash A \Leftarrow \mathbf{Type} \\ \Gamma \vdash a \Leftarrow A\end{array}}{\Gamma \vdash (a : A) \Rightarrow A}$$

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

C-LAMBDA

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

C-INFER-SIMPLE

$$\frac{\begin{array}{c}a \text{ is not a } \lambda\text{-expression} \\ \Gamma \vdash a \Rightarrow A\end{array}}{\Gamma \vdash a \Leftarrow A}$$

Figure 3: (Simple) Bidirectional type system

`checkType`. Furthermore, to keep track of which rule is in which judgment, rules that have inference as conclusion start with I- and rules that have checking as conclusion start with C-.

Let us 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. (If we assume that the type provided to the checking judgment has already been checked to be a good type, then we can skip checking 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 example, if map has type

```
map : (x : Type) → (y: Type) → (x → y) → List x → List y
```

then in an application, we can synthesize the type of the expression

```
map Nat Nat (λx.x)
```

as `List Nat → List Nat`. After the two type applications, we will end up checking the function argument against the type `Nat → Nat`.

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 judgment more applicable with rule C-INFER-SIMPLE that allows us to use inference whenever a checking rule doesn't apply. This rule only applies when the term is not a lambda expression.

Now, let us 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:

$$\frac{\emptyset \vdash \lambda x.x : \mathbf{Bool} \to \mathbf{Bool} \qquad \emptyset \vdash \mathbf{True} : \mathbf{Bool}}{\emptyset \vdash (\lambda x.x)\ \mathbf{True} : \mathbf{Bool}}\ \textsc{t-app}$$

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 the top level. For example, we can construct this derivation.

$$\frac{\emptyset \vdash (\lambda x.x : \mathbf{Bool} \to \mathbf{Bool}) \Rightarrow \mathbf{Bool} \to \mathbf{Bool} \qquad \emptyset \vdash \mathbf{True} \Leftarrow \mathbf{Bool}}{\emptyset \vdash (\lambda x.x : \mathbf{Bool} \to \mathbf{Bool})\ \mathbf{True} \Rightarrow \mathbf{Bool}}\ \textsc{i-app}$$

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[8]:

Lemma 4.1 (Correctness of the bidirectional type system). If $\Gamma \vdash a \Rightarrow A$ and $\vdash \Gamma$ then $\Gamma \vdash a : A$. If $\Gamma \vdash a \Leftarrow A$ and $\vdash \Gamma$ and $\Gamma \vdash A : \mathbf{Type}$ then $\Gamma \vdash a : A$.

[8] This result requires the addition of a typing rule for annotated terms $(a : A)$ to the original system.

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

This issue is not significant, and 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.

4.2 Homework: Add Booleans and Σ-types (Part I)

Some fairly standard typing rules for booleans are shown below.

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

T-BOOL

$$\frac{}{\Gamma \vdash \mathbf{Bool} : \mathbf{Type}}$$

T-TRUE

$$\frac{}{\Gamma \vdash \mathbf{True} : \mathbf{Bool}}$$

T-FALSE

$$\frac{}{\Gamma \vdash \mathbf{False} : \mathbf{Bool}}$$

T-IF-SIMPLE

$$\frac{\Gamma \vdash a : \mathbf{Bool} \qquad \Gamma \vdash b_1 : A \qquad \Gamma \vdash b_2 : A}{\Gamma \vdash \mathbf{if}\ a\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 : A}$$

Likewise, we can also extend the language with Σ-types, or products, where the type of the second component of the product can depend on the value of the first component.

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

T-SIGMA

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

T-PAIR

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

$$\begin{array}{l}\text{T-LETPAIR-WEAK}\\ \dfrac{\Gamma \vdash a : \{x{:}A_1 \mid A_2\} \qquad \Gamma, x{:}A_1, y{:}A_2 \vdash b : B \qquad \Gamma \vdash B : \mathbf{Type}}{\Gamma \vdash \mathbf{let}\,(x, y) = a\,\mathbf{in}\,b : B}\end{array}$$

We destruct Σ-types using pattern matching. The simple rule for pattern matching introduces variables into the context when type checking the body of the let expression. These variables are not allowed to appear free in the result type of the pattern match.

For your exercise, 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?

5 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 will 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         - runs the parser and type checker

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

5.1 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 will 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 (mostly) for free. This works because we use type constructors from this library in the definition of the abstract syntax of `pi-forall`, and those types 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
```

The `Unbound.Name` type is indexed by `Term`, the AST type that it is a name for, whose definition we will see shortly. 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 library includes an overloaded function `subst x a b`, which corresponds to the substitution we have been writing as $b[a/x]$ in our inference rules,[9] i.e., substitute a for x in b.

In general, we 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 the typing context. Or there may be other sorts of syntax that terms can be part of, and we want to be able to substitute for term variables through that syntax too. 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

  = -- | type of types `Type`
    TyType

  | -- | variables `x`
    Var TName

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

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

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

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

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

```
  ...
  deriving (Show, Generic)
```

As you can see, variables are represented by names. `Unbound`'s `Bind` type constructor declares the scope of the bound variables. Both `Lam` and `TyPi` bind a single variable in a `Term`. However, in a `TyPi` 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 will use this name in the source code just for documentation.

```
  type Type = Term
```

Among other things, the `Alpha` class enables functions for alpha equivalence and free variable calculation, with the types shown below. Usually `Unbound` can derive instances of this class for us so that we don't have to worry about defining these functions manually.

```
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 and source code positions for error messages. Therefore, our instance does so for these constructors of the `Term` data type and defers to the generic definition for the rest.

For example, if `xName` and `yName` are two different names, then `Unbound`'s implementation of α-equivalence can equate terms that differ only in using these names for binding.

```
-- 'λx → x'
idx :: Term
idx = Lam (Unbound.bind xName (Var xName))

-- 'λy → y'
idy :: Term
idy = Lam (Unbound.bind yName (Var yName))

-- >>> Unbound.aeq idx idy
-- True
```

Second, the `Subst` instance for `Term` requires telling `Unbound` where variables occur in the `Term`. This short function pattern matches to find the `Var` constructor, extracts its name, and returns it to `Unbound`. We don't need to provide any more code to `Unbound` in the implementation of substitution; the generic implementation suffices.

```
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`[10].

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

5.2 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. The checking function should take a term, a context, and a type, and determine whether the term has that type. However, note that the types above are not very informative in the case of failure: if a term does not type check, then the result is `Nothing` or `False`.

For that reason, we will do something a bit different. We will define a *type checking monad*, called `TcMonad`, that will handle the plumbing for the typing context, and more importantly, allow us to return more information when a program doesn't type check. Therefore our two functions will have these types that hide the context and rely on throwing errors to indicate type checking failure.

```
inferType :: Term -> TcMonad Type

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

Those of you who have worked with Haskell before may be familiar with the classes MonadReader, and the MonadError, which our type checking monad will be instances of. These two classes mean that that the `TcMonad` type can provide access to the type checking context and can throw errors. We will wrap this functionality up a bit with the following functions. The first two encapsulate the sort of context manipulation we need to do: look up the types of variables and add a new assumption about the type of a variable to the context (both `Sig` and `Decl` are defined in `Syntax.hs`).

```
-- | Find the type of a name specified in the context
-- throwing an error if the name doesn't exist
lookupTy :: TName -> TcMonad Sig

-- | Extend the context with a new binding
extendCtx :: Decl -> TcMonad Term -> TcMonad Term

-- | Throw an error at the corrent location, assembling the error message
-- from a list of displayable objects. Terminates type checking
err  :: (Disp a) => [a] -> TcMonad b

-- | Print a warning, but continue the current computation.
warn :: (Disp a) => a -> TcMonad ()
```

We will also need this monad to be a freshness monad, to support working with binding structure (i.e., a source of fresh names), and throw in MonadIO so that we can print warning messages.

5.3 Implementing the Type Checking Algorithm [Typecheck.hs]

Now that we have the type checking monad available, we can start our implementation.

The general structure of the inferType function starts with a pattern match for the various syntactic forms. There are also several cases for practical reasons (annotations, source code positions, etc.) and a few cases for homework. If the form of the term does not match any of the forms of the terms that we can synthesize types for, then the type checker produces an error.

```
inferType tm = case tm of
   (Var x) -> ...

   TyType -> ...

   (TyPi tyA bnd) -> ...

   (App t1 t2) -> ...

   (Ann tm ty) -> ...

   ...

   _ -> Env.err [DS "Must have a type for", DD tm]
```

For checking, the general form of the function also

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

```
checkType tm ty = case tm of
  (Lam bnd) -> case ty of
     TyPi tyA bnd2 = ...
        -- pass in the Pi type for the lambda expression
     _ ->
       Env.err [DS "Lambda expression has a function type, not ", DD ty]
```

Again there are several cases for practical reasons, plus 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.

```
 -- c-infer
 tm -> do
      ty' <- inferType tm
      unless (Unbound.aeq ty' ty) $
         Env.err [DS "Types don't match", DD ty, DS "and", DD ty'] -}
```

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

5.4 Exercise: Add Booleans and Σ-types (Part II)

The code in `version1/` includes abstract and concrete syntax for booleans and Σ-types. The `pi-forall` file `version1/test/Hw1.pi` contains examples of using these new forms. Your job is to get this file to compile by filling in the missing cases in `version1/src/TypeCheck.hs` based on the bidirectional rules that you worked out in the previous exercise.

6 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 will talk about algorithmic versions of both the equality relation and how to introduce it into the algorithmic type system. Finally, we will cover modifications to the Haskell implementation.

6.1 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.

Here's an example where we want a definition of equality that is more expressive than alpha-equivalence. Consider an encoding for simply-typed products:

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

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

```
prod : (p:Type) → (q:Type) → p → q → pair p q
prod = λp.λq. λx.λy. λc. λf. f x y
```

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

```
Type Error:
pi/Lec1.pi:56:22:
  Lambda expression should have a function type, not pair p q
  When checking the term
    \p. \q. \x. \y. \c. \f. f x y
  against the signature
    prod : (p : Type) -> (q : Type) -> p -> q -> pair p q
  In the expression
    \c. \f. f x y
```

The problem is that even though we want `pair 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 `pair` 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 `pair` and beta-reduce this application.

6.2 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 n A`. 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 (Succ n) A → Vec n A
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 1 Bool
v1 = Cons [0] True Nil
```

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. Take a look at the `append` function, that takes a vec of length `m`, vector of length n, and returns a vector of length `plus m n`:

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

In that case, to get `head Bool 1 (append v1 v1)` to type check, we need to show that the type `Vec (Succ 1) Bool` is equal to the type `Vec (plus 1 1) Bool`. If our definition of type equality is *alpha-equivalence*, then this equality

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

$$\textsc{e-beta}\quad \frac{}{\Gamma \vdash (\lambda x.a)\ b = a[b/x]} \qquad \textsc{e-refl}\quad \frac{}{\Gamma \vdash A = A} \qquad \textsc{e-sym}\quad \frac{\Gamma \vdash A = B}{\Gamma \vdash B = A} \qquad \textsc{e-trans}\quad \frac{\begin{array}{c}\Gamma \vdash A_1 = A_2\\ \Gamma \vdash A_2 = A_3\end{array}}{\Gamma \vdash A_1 = A_3}$$

$$\textsc{e-pi}\quad \frac{\begin{array}{c}\Gamma \vdash A_1 = A_2\\ \Gamma, x\!:\!A_1 \vdash B_1 = B_2\end{array}}{\Gamma \vdash (x\!:\!A_1) \to B_1 = (x\!:\!A_2) \to B_2} \qquad \textsc{e-lam}\quad \frac{\Gamma, x\!:\!A_1 \vdash b_1 = b_2}{\Gamma \vdash \lambda x.b_1 = \lambda x.b_2}$$

$$\textsc{e-app}\quad \frac{\begin{array}{c}\Gamma \vdash a_1 = a_2\\ \Gamma \vdash b_1 = b_2\end{array}}{\Gamma \vdash a_1\ b_1 = a_2\ b_2} \qquad \textsc{e-lift}\quad \frac{\begin{array}{c}\Gamma, x\!:\!A \vdash b : B\\ \Gamma \vdash a_1 = a_2\end{array}}{\Gamma \vdash b[a_1/x] = b[a_2/x]} \qquad \textsc{e-annot}\quad \frac{\Gamma \vdash a_1 = a_2}{\Gamma \vdash (a_1 : A) = a_2}$$

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.

6.3 Defining equality

The main idea is that we will:

- establish a new judgment 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

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

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

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 each other 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.)

How do we know our definition is good? We want to make sure that whatever rules that we add to the system, we define a relation that is *consistent.* We want to equate as many terms/types as possible, but we don't want to be be able to derive an equality between **Type** and some function type (e.g. $(x : \mathbf{Type}) \to \mathbf{Type}$), or an equality between **True** and **False**. To be absolutely confident about our definition, we want a proof that such equalties are not derivable in our system.

6.4 Using definitional equality in the algorithm

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

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

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 need 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.

  $$\textsc{c-infer}\quad \frac{a \text{ is not a } \lambda\text{-expression} \qquad \Gamma \vdash a \Rightarrow A \qquad \Gamma \vdash A \Leftrightarrow B}{\Gamma \vdash a \Leftarrow B}$$

  In this case, the types A and B are available to the type checker, and 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 6.5 below.)

$\boxed{\textbf{whnf}\ a \rightsquigarrow \mathit{nf}}$ *(Weak head normal form reduction)*

WHNF-LAM-BETA
$$\frac{\begin{array}{c}\textbf{whnf}\ a \rightsquigarrow (\lambda x.a') \\ \textbf{whnf}\ (a'[b/x]) \rightsquigarrow \mathit{nf}\end{array}}{\textbf{whnf}\ a\ b \rightsquigarrow \mathit{nf}}$$

WHNF-ANNOT
$$\frac{\textbf{whnf}\ a \rightsquigarrow \mathit{nf}}{\textbf{whnf}\ (a : A) \rightsquigarrow \mathit{nf}}$$

WHNF-TYPE
$$\frac{}{\textbf{whnf}\ \textbf{Type} \rightsquigarrow \textbf{Type}}$$

WHNF-LAM
$$\frac{}{\textbf{whnf}\ \lambda x.a \rightsquigarrow \lambda x.a}$$

WHNF-PI
$$\frac{}{\textbf{whnf}\ (x\!:\!A) \to B \rightsquigarrow (x\!:\!A) \to B}$$

WHNF-VAR
$$\frac{}{\textbf{whnf}\ x \rightsquigarrow x}$$

WHNF-APP
$$\frac{\textbf{whnf}\ a \rightsquigarrow \mathit{ne}}{\textbf{whnf}\ a\ b \rightsquigarrow \mathit{ne}\ b}$$

Figure 5: Relation computing the weak head normal form of a term

- 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 would like our algorithm for type equality to be able to figure this out for us.

  I-APP
  $$\frac{\Gamma \vdash a \Rightarrow A \qquad \textbf{whnf}\ A \rightsquigarrow (x\!:\!A_1) \to B \qquad \Gamma \vdash b \Leftarrow A_1}{\Gamma \vdash a\ b \Rightarrow B[b/x]}$$

  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.

- Whenever we call `checkType` we should call it with a term that has already been reduced to normal form. This will allow rule C-LAMBDA to match against a literal function type.

  C-WHNF
  $$\frac{\begin{array}{c}A \text{ is not in weak-head normal form} \\ \Gamma \vdash a \Leftarrow \mathit{nf} \qquad \textbf{whnf}\ A \rightsquigarrow \mathit{nf}\end{array}}{\Gamma \vdash a \Leftarrow A}$$

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

$$\boxed{\textbf{whnf}\ a \rightsquigarrow \mathit{nf}}$$

describes when a term *a* reduces to some result *nf* in *weak head normal form (whnf)*. For closed terms, these rules correspond to a big-step evaluation relation and produce a value (i.e. abstraction or type form). For open terms, the rules could also produce terms that are a sequence of applications headed by a variable. In this case, we call the term a *neutral* term, and use the metavariable *ne* to refer to it.

$$\begin{array}{lccl}
\textit{value} & v & ::= & \lambda x.a\ \textbf{Type} \mid (x\!:\!A) \to B \\
\textit{neutral terms} & ne & ::= & x \mid ne\ a \\
\textit{weak-head normal form} & nf & ::= & v \mid ne
\end{array}$$

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 anymore.

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.

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 `checkType` to make sure that the annotated type matches the inferred type.
- `whnf` is called in the `App` case of `inferType` to ensure that the function has some sort of function type.
- `whnf` is called at the beginning of `checkType` to make sure that we are using the head form of the type in checking mode.

$\boxed{\Gamma \vdash A \Leftrightarrow B}$ *(Algorithmic equality)*

EQ-REFL
$$\frac{}{\Gamma \vdash a \Leftrightarrow a}$$

EQ-WHNF
$$\frac{\begin{array}{c}\mathbf{whnf}\ a \leadsto nf_1 \\ \mathbf{whnf}\ b \leadsto nf_2 \\ \Gamma \vdash nf_1 \Leftrightarrow nf_2\end{array}}{\Gamma \vdash a \Leftrightarrow b}$$

EQ-ABS
$$\frac{\Gamma \vdash a_1 \Leftrightarrow a_2}{\Gamma \vdash \lambda x.a_1 \Leftrightarrow \lambda x.a_2}$$

EQ-PI
$$\frac{\begin{array}{c}\Gamma \vdash A_1 \Leftrightarrow A_2 \\ \Gamma \vdash B_1 \Leftrightarrow B_2\end{array}}{\Gamma \vdash (x\!:\!A_1) \to B_1 \Leftrightarrow (x\!:\!A_2) \to B_2}$$

EQ-APP
$$\frac{\begin{array}{c}\Gamma \vdash ne_1 \Leftrightarrow ne_2 \\ \Gamma \vdash b_1 \Leftrightarrow b_2\end{array}}{\Gamma \vdash ne_1\ b_1 \Leftrightarrow ne_2\ b_2}$$

Figure 6: Algorithmic equality for core `pi-forall`

6.5 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 a *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 find out that terms are equivalent without fully reducing them. For example, if we observe that the two terms are already α-equivalent, then we already know that they are equal—no need to reduce them first. Because reduction can be expensive (or even nonterminating) we would like to only reduce as much as we need to find out whether the terms can be identified.

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

```
equate t1 t2 = do
   if (Unbound.aeq t1 t2) 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 for alpha-equivalence. If they are, we do not need to do anything else. Instead, the function returns 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.

This algorithm is summarized in Figure 6.

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

- We can extend this algorithm to 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 equality is not decidable in `pi-forall`. There will always be terms that could cause `equate` to loop forever. However, the algorithm is sound. If it says that two terms are equal, then there is some derivation using the rules of Figure 4 that the terms are equal.

7 Dependent pattern matching and propositional equality

One of the powerful features of dependently-typed languages is the idea of "learning by testing". In other words, the information gained in pattern matching can be reflected into the type system, allowing it to reason in a flow-sensitive manner. We will demonstrate how this works in this section.

7.1 More expressive rules for if and Σ-type elimination

Consider our elimination rule for if from your homework assignment:

$$\frac{\Gamma \vdash a : \mathbf{Bool} \qquad \Gamma \vdash b_1 : A \qquad \Gamma \vdash b_2 : A}{\Gamma \vdash \mathbf{if}\, a\, \mathbf{then}\, b_1\, \mathbf{else}\, b_2 : A}\ \textsc{t-if-simple}$$

One solution to the homework assignment is to add the following two rules to the algorithmic system, one in inference mode and one in checking mode.

$$\frac{\Gamma \vdash a \Leftarrow \mathbf{Bool} \qquad \Gamma \vdash b_1 \Leftarrow A \qquad \Gamma \vdash b_2 \Leftarrow A}{\Gamma \vdash \mathbf{if}\, a\, \mathbf{then}\, b_1\, \mathbf{else}\, b_2 \Leftarrow A}\ \textsc{c-if-simple}$$

$$\frac{\begin{array}{c}\Gamma \vdash a \Leftarrow \mathbf{Bool}\\ \Gamma \vdash b_1 \Rightarrow A_1 \qquad \Gamma \vdash b_2 \Rightarrow A_2 \qquad \Gamma \vdash A_1 \Leftrightarrow A_2\end{array}}{\Gamma \vdash \mathbf{if}\, a\, \mathbf{then}\, b_1\, \mathbf{else}\, b_2 \Rightarrow A_1}\ \textsc{i-if-simple}$$

In both of these rules, we should check that the condition has type bool (when using the checking judgement here, the algorithm will infer the type of the condition and then compare the inferred type against bool using definitional equality.) However, there is no unique best way to handle the branches of the if expression. Some examples work better using the first rule and some examples require the second rule. For example, we can only type check the example below using the first (checking) rule:

$$\emptyset \vdash \lambda y.\mathbf{if}\, y\, \mathbf{then}\, (\lambda x.x)\, \mathbf{else}\, (\lambda x.not\ \ x) \Leftarrow \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool}$$

and we can only type check this next example using the second (inference) rule:

$$x : \mathbf{Bool}, y : \mathbf{Bool} \to \mathbf{Bool} \vdash (\mathbf{if}\, x\, \mathbf{then}\, y\, \mathbf{else}\, not\ )\ true\ \Rightarrow \mathbf{Bool}$$

However, note that we can strengthen the specification of the type checking rule for `if` by making the result type `A` depend on whether the condition is true or false when type checking each branch. In this way, our type system is flow-sensitive: we do different things while type checking based on how control flows in a `pi-forall` program.

$$\frac{\begin{array}{c}\Gamma \vdash a : \mathbf{Bool} \qquad \Gamma \vdash b_1 : A[\mathbf{True}/x]\\ \Gamma \vdash b_2 : A[\mathbf{False}/x] \qquad \Gamma, x : \mathbf{Bool} \vdash A : \mathbf{Type}\end{array}}{\Gamma \vdash \mathbf{if}\, a\, \mathbf{then}\, b_1\, \mathbf{else}\, b_2 : A[a/x]}\ \textsc{t-if-full}$$

For example, here is a simple program that requires this stronger rule to type check. The function `T` is an example of a *large elimination*: a function that destructs a boolean value to produce a type. Then in the type of `bar` below, each branch can return a different type of result, according to `T`.

```
-- 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 strong type checking rule is difficult to implement. In fact, it is not syntax-directed because A and x are not fixed by the syntax. Given $A[\textbf{True}/x]$ and $A[\textbf{False}/x]$ and $A[a/x]$ (or anything that they are definitionally equal to!) how can we figure out whether they correspond to each other? Neither inference nor checking mode works, because we don't ever see the type A with just the variable x.

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

$$\textsc{T-if}\quad \frac{\begin{array}{c}\Gamma \vdash x : \textbf{Bool} \\ \Gamma \vdash b_1 : A[\textbf{True}/x] \qquad \Gamma \vdash b_2 : A[\textbf{False}/x] \qquad \Gamma \vdash A : \textbf{Type}\end{array}}{\Gamma \vdash \textbf{if}\ x\ \textbf{then}\ b_1\ \textbf{else}\ b_2 : A}$$

And, in our bidirectional system, we will only allow refinement when we are in checking mode. That way, because a is just x, we can easily identify how the type should vary in each branch.

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

To implement this rule, we need only remember that x is **True** or **False** when checking the individual branches of the if expression. [11]

We can modify the elimination rule for Σ-types similarly. Here, we are not learning which branch was taken, but we are still learning something. If we get

[11] Here is an alternative version, for inference mode only, suggested during a prior summer school:

$$\textsc{I-if-alt}\quad \frac{\Gamma \vdash a \Leftarrow \textbf{Bool} \qquad \Gamma \vdash b_1 \Rightarrow B_1 \qquad \Gamma \vdash b_2 \Rightarrow B_2}{\Gamma \vdash \textbf{if}\ a\ \textbf{then}\ b_1\ \textbf{else}\ b_2 \Rightarrow \textbf{if}\ a\ \textbf{then}\ B_1\ \textbf{else}\ B_2}$$

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

$$\textsc{E-if-eta}\quad \frac{}{\Gamma \vdash \textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ b = b}$$

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.

to the body of the let expression, we know that the scrutinee terminated and produced some product value.

$$\frac{\Gamma \vdash z \Rightarrow \{x\!:\!A_1 \mid A_2\} \qquad \Gamma, x\!:\!A_1, y\!:\!A_2 \vdash b \Leftarrow B[(x,y)/z]}{\Gamma \vdash \textbf{let}\,(x,y) = z\,\textbf{in}\,b \Leftarrow B}\;\textsc{c-letpair}$$

This modification changes our definition of Σ-types from weak Σs to strong Σ-types. 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 Σ-types cannot define the second projection. The following code only type checks when the above rule is available.

```
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.)

7.1.1 Definitions and let expressions

Now let's consider let expressions, of the form $\textbf{let}\,x = a\,\textbf{in}\,b$. The simplest rule that we can use for this term is the following:

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x\!:\!A \vdash b : B \qquad \Gamma \vdash B : \textbf{Type}}{\Gamma \vdash \textbf{let}\,x = a\,\textbf{in}\,b : B}\;\textsc{t-let-simple}$$

Note the last premise: we need to make sure that the variable x does not appear in the type B because otherwise it would escape its scope. In the specificational version of the type system, this version of the let rule is derivable from function application: $\textbf{let}\,x = a\,\textbf{in}\,b$ is equivalent to $(\lambda x.b)\ a$. On the other hand, the bidirectional system won't type check this definition without an annotation, so we should add this expression form directly.

However, we can also develop a stronger rule, which statically tracks that, because this is a let-expression, we know that x is equal to its definition. This view leads to the following rule:

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x\!:\!A, x = a \vdash b : B \qquad \Gamma \vdash B : \textbf{Type}}{\Gamma \vdash \textbf{let}\,x = a\,\textbf{in}\,b : B}\;\textsc{t-let-def}$$

In this rule, we've added a definition to our context, i.e. $x = a$. These definitions are used during weak head normalization. If we get to a variable in the head position, and there is a definition for it in the context, we should look it up and keep normalizing.

Definitions in the context act like deferred substitutions. Indeed, rule T-LET-DEF doesn't really increase the expressiveness of the type system compared to rule T-LET-SIMPLE: if we had derivation that uses this rule, we can replace it with one that uses $b[a/x]$ instead of b in the body of the let expression. However, for programmers, if a is a large expression and important for type checking, it is more convenient for to write x instead.

How do we approach let expressions in the bidirectional system? As with if and letpair expressions, it makes sense to have both checking and inference rules. In checking mode, the rule is very similar to the specification. We extend the context appropriately and use the known type to check the body of the expression.

$$\textsc{c-let}\quad \frac{\Gamma \vdash a \Rightarrow A \qquad \Gamma, x\!:\!A, x = a \vdash b \Leftarrow B}{\Gamma \vdash \textbf{let}\, x = a\, \textbf{in}\, b \Leftarrow B}$$

However, in inference mode, we have to be careful that the variable x does not escape its scope. One option would be just to reject the program if this would happen. However, a more flexible approach is to substitute this variable away with its definition.[12]

$$\textsc{i-let}\quad \frac{\Gamma \vdash a \Rightarrow A \qquad \Gamma, x\!:\!A, x = a \vdash b \Rightarrow B}{\Gamma \vdash \textbf{let}\, x = a\, \textbf{in}\, b \Rightarrow B[a/x]}$$

Definitions and refinement With the addition of definitions, we can revise our refining rules for booleans and Σ-types. Instead of substituting in the result type, we can replace those rules with suspended definitions in the context.

$$\textsc{c-if-def}\quad \frac{\begin{array}{c}\Gamma \vdash x \Leftarrow \textbf{Bool}\\ \Gamma, x = \textbf{True} \vdash b_1 \Leftarrow A \qquad \Gamma, x = \textbf{False} \vdash b_2 \Leftarrow A\end{array}}{\Gamma \vdash \textbf{if}\, x\, \textbf{then}\, b_1\, \textbf{else}\, b_2 \Leftarrow A}$$

$$\textsc{c-letpair-def}\quad \frac{\Gamma \vdash z \Rightarrow \{x\!:\!A_1 \mid A_2\} \qquad \Gamma, x\!:\!A_1, y\!:\!B_2, z = (x, y) \vdash b \Leftarrow B}{\Gamma \vdash \textbf{let}\, (x, y) = z\, \textbf{in}\, b \Leftarrow B}$$

[12] This rule for let can be used to derive the more general form of refinement rules for booleans and other datatypes. In the general case, we can annotate an elimination form with a *motive*:

$$\textsc{i-if-motive}\quad \frac{\begin{array}{c}\Gamma \vdash a \Leftarrow \textbf{Bool}\\ \Gamma \vdash b_1 \Leftarrow A[\textbf{True}/x] \qquad \Gamma \vdash b_2 \Leftarrow A[\textbf{False}/x] \qquad \Gamma, x\!:\!\textbf{Bool} \vdash A \Leftarrow \textbf{Type}\end{array}}{\Gamma \vdash \textbf{if}\, a\, \textbf{then}\, b_1\, \textbf{else}\, b_2[x.A] \Rightarrow A[a/x]}$$

This annotation lets us use refinement, even when the condition is not a variable, and even in inference mode. However, we can also derive this rule via let:

$$\textbf{if}\, a\, \textbf{then}\, b_1\, \textbf{else}\, b_2[x.A] = \textbf{let}\, x = a\, \textbf{in}\, (\textbf{if}\, x\, \textbf{then}\, b_1\, \textbf{else}\, b_2 : A)$$

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

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

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

Figure 7: Propositional equality (Specificational rules)

7.2 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 will start by adding a propositional equality type to `pi-forall`.

The main idea of the equality type is that it converts a *judgment* that two types are equal into a *type* that evaluates to a special value form, called **refl**, only when two types are equal.[13] In other words, the new value form **refl** has type $a = b$ when these two types are definitionally equal to each other, as shown in rule T-REFL in Figure 7.

Sometimes, you might see rule T-REFL written as follows:

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

However, this rule is equivalent to the above version because types are equal up to congruence. If we know that $\emptyset \vdash a = b$, then we know that the type $a = a$ is equal to the type $a = b$. In a type system with conversion, it doesn't matter which of these equal types we choose for the conclusion of the rule.

However, in the bidirectional version of the rules, shown in Figure 8, it does make a difference. The algorithmic typing rule for **refl** requires checking mode so that we know which types we need to show equivalent. (See rule C-REFL.) Furthermore, this rule allows a and b to differ so that there are no restrictions on its use. Instead, to implement this rule, we need only compare a and b using our algorithmic equality, as we did in Section 6.

[13] Recall that all types are inhabited in `pi-forall`, so to know whether types are equal, we must make sure that we have an actual proof of equality.

$\boxed{\Gamma \vdash a \Leftarrow A}$ *(Checking)*

$$\textsc{c-refl}\quad \frac{\Gamma \vdash a \Leftrightarrow b}{\Gamma \vdash \mathbf{refl} \Leftarrow a = b}
\qquad
\textsc{c-subst-left}\quad \frac{\begin{array}{c}\Gamma \vdash y \Rightarrow B\\ \mathbf{whnf}\ B \rightsquigarrow (x = a_2)\\ \Gamma \vdash a \Leftarrow A[a_2/x][\mathbf{refl}/y]\end{array}}{\Gamma \vdash \mathbf{subst}\ a\ \mathbf{by}\ y \Leftarrow A}
\qquad
\textsc{c-subst-right}\quad \frac{\begin{array}{c}\Gamma \vdash y \Rightarrow B\\ \mathbf{whnf}\ B \rightsquigarrow (a_1 = x)\\ \Gamma \vdash a \Leftarrow A[a_1/x][\mathbf{refl}/y]\end{array}}{\Gamma \vdash \mathbf{subst}\ a\ \mathbf{by}\ y \Leftarrow A}$$

$$\textsc{c-contra}\quad \frac{\begin{array}{c}\Gamma \vdash a : A \qquad \mathbf{whnf}\ A \rightsquigarrow (a_1 = a_2)\\ \mathbf{whnf}\ a_1 \rightsquigarrow \mathbf{True} \qquad \mathbf{whnf}\ a_2 \rightsquigarrow \mathbf{False}\end{array}}{\Gamma \vdash \mathbf{contra}\ a \Leftarrow B}$$

$\boxed{\Gamma \vdash a \Rightarrow A}$ *(Inference)*

$$\textsc{i-eq}\quad \frac{\Gamma \vdash a \Rightarrow A \qquad \Gamma \vdash b \Rightarrow B \qquad \Gamma \vdash A \Leftrightarrow B}{\Gamma \vdash (a = b) \Rightarrow \mathbf{Type}}$$

Figure 8: Propositional equality (Bidirectional rules)

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 T-EQ. In the algorithm, rule I-EQ, we infer the types of each side of the equality type and then check to make sure that those types are equal.

The elimination rule for propositional equality is written $\mathbf{subst}\ a\ \mathbf{by}\ b$ 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.

$$\textsc{t-subst-simple}\quad \frac{\Gamma \vdash a : A[a_1/x] \qquad \Gamma \vdash b : a_1 = a_2}{\Gamma \vdash \mathbf{subst}\ a\ \mathbf{by}\ b : A[a_2/x]}$$

How can we implement this rule? For simplicity in these lecture notes, we will play the same trick that we did with booleans, requiring that one of the sides of the equality be a variable.[14]

$$\textsc{c-subst-left-simple}\quad \frac{\Gamma \vdash b \Rightarrow B \qquad \mathbf{whnf}\ B \rightsquigarrow (x = a_2) \qquad \Gamma \vdash a \Leftarrow A[a_2/x]}{\Gamma \vdash \mathbf{subst}\ a\ \mathbf{by}\ b \Leftarrow A}$$

[14] The `pi-forall` implementation implements a more expressive rule that uses *unification* to determine the substitution to apply to A. Unification means that if we have an equality type such as $(x, \mathbf{True}) = ((), y)$ then we can produce the substitution of $()$ for x and $\mathbf{True}$ for y.

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

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 in Figure 7, which observes that the only way to construct a proof of equality is with the term **refl**.[15]

As above, the rule C-SUBST-RIGHT, shown in Figure 8, (and the corresponding left rule) only applies when b is also a variable.

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 will use rule T-CONTRA, shown in Figure 7, that requires a proof that **True** equals **False**. The algorithmic version of this rule, rule C-CONTRA, must use weak-head normalization multiple times: first to find the equality type and then to show that the two sides of it have different head forms. This rule must be in checking mode because we can use `contra` to inhabit any type.

7.2.1 Homework (`pi-forall`: equality)

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

8 Irrelevance: the ∀ of `pi-forall`

Now, let us talk about irrelevance. 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 as part of the compilation pipeline. This erasure makes sense because 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 would like to erase types too. And erase proofs that are only there to make things type check. Coq does this by

[15] This version of subst is similar to an eliminator for propositional equality called J. However, our eliminator is very strong and can be used to derive "axiom" K or the uniqueness of identity proofs. The eliminator for propositional equality in Coq and Agda is not this expressive, leaving the type system open to extension with other proofs of propositional equality.

making a distinction between `Prop` and `Set`. Everything in `Set` stays around until run time, and is guaranteed not to depend on `Prop`.

We will take another approach.

In `pi-forall` we have a new kind of quantification, called "forall", that marks erasable arguments. We mark forall quantified arguments with square 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 will 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 parameter `y` is erasable, but it is not. And, when we check this code with `version3` of `pi-forall`, it is rejected with the following error message.

```
Type Error:
Lec3.pi:16:16:
  Cannot access irrelevant variables in this context
  When checking the term
    \[x] [y]. y
  against the signature
    id’ : [x : Type] -> [x] -> x
  In the expression
    y
```

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 that 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.

8.1 How do we track irrelevance in the type system?

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

The key idea is that we mark variable assumptions in the context with an ϵ annotation, which is either $+$ (relevant) or $-$ (irrelevant). and only relevant variables can be used in terms. ("Normal" variables, introduced by λ-expressions, will always be marked as relevant.) As a result, we revise the variable typing rule to require the relevant annotation. In the `version3` implementation, type signatures in the environment now include an `Epsilon` tag, and the function `checkStage` ensures that this tag is $+$ when the variable is used in the term.

$$\textsc{t-evar}\quad \frac{x :^{+} A \in \Gamma}{\Gamma \vdash x : A}$$

An irrelevant abstraction introduces its variable into the context tagged with the irrelevant annotation. Because of the $-$ tag, these variables are inaccessible in the body of the function.

$$\textsc{t-elambda}\quad \frac{\Gamma, x :^{-} A \vdash a : B \qquad \Gamma^{+} \vdash A : \textbf{Type}}{\Gamma \vdash \lambda[x].a : [x : A] \to B}$$

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.

$$\textsc{t-eapp}\quad \frac{\Gamma \vdash a : [x : A] \to B \qquad \Gamma^{+} \vdash b : A}{\Gamma \vdash a[b] : B[b/x]}$$

This operation on the context, called *resurrection*, converts all "$-$" tags on variables to be "$+$". It represents a shift in our perspective: because we have entered an irrelevant part of the term, 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.

$$\textsc{t-epi}\quad \frac{\Gamma \vdash A : \textbf{Type} \qquad \Gamma, x :^{+} A \vdash B : \textbf{Type}}{\Gamma \vdash [x : A] \to B : \textbf{Type}}$$

8.2 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.

However, there is another benefit to marking some arguments as irrelevant: we can ignore them during equivalence checking. When deciding whether two terms are equal, we don't need to look at their irrelevant components. In other words, we can add the following equality rule to our definitional equality in the case of applications with irrelevant arguments.

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

If you look at `version3` of the implementation, you can see where we use this idea in the definition of equality. Note in this version that arguments are now tagged with their relevance, and when we compare arguments that are tagged as irrelevant, we do nothing. In fact, what we are doing is actually *defining* equality over just the computationally relevant parts of the term instead of the entire term. And we have been doing this in a limited form from the beginning—recall that `version1` of `pi-forall` already ignores type annotations. Just as type annotations do not affect computation and can be ignored, the same reasoning holds for irrelevant arguments.

Why is compile-time irrelevance important?

- faster comparison: don't have to look at the whole term when comparing for equality
- more expressive: these parts of the term don't actually have to be equal

For example, consider the example below. The parameter p takes an irrelevant argument, so we know that it must be a constant function. 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
```

9 Datatypes and Indexed Datatypes

Finally, we will add a general implementation of datatypes (including potentially irrelevant 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 Haskell, 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 am giving you is not the complete story. Recall that we are not considering termination. That means that we can use datatypes merely by writing recursive functions; we do not have 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 here.

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

9.1 "Dirt simple" datatypes

Our first goal is simple. What do we need to get the simplest examples of non-recursive and regular recursive datatypes working? By this I mean datatypes that you might see in Haskell or ML, such as `Bool`, `Void` and `Nat`. Here are some examples of what we would like these types to look like in `pi-forall`. If you are looking at the implementation, you will find this code in `pi/Lec4.pi`.

9.1.1 Booleans

For example, one homework assignment was to implement booleans as a primitive language feature. We would like to replace this primitive feature with a datatype definition of a new type constructor (`Bool`), with two constructors `True` and `False`, defined by the following data type declaration.

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

In the homework assignment, we used `if` as the elimination form for boolean values. For example, with the primitive `if` expression, we could write this function.

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

For uniformity, we want to have a common elimination form for all datatypes, called `case`. This expression form has branches for each of the data constructors of the data type. (In `pi-forall`, we will steal Haskell's 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
```

9.1.2 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 {}
```

The `Void` type 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? For `Void` this is simple, as we can see from its declaration that it has no cases to cover. But, as our language becomes more sophisticated, we will need to revisit our implementation to make sure that it can correctly identify when a case expression is exhaustive.

9.1.3 Nat

Natural numbers include a data constructor with an argument, which we can write in the syntax of `pi-forall` as below. (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
```

However, to write `plus`, we need to use recursion:

```
plus : Nat → Nat → Nat
plus = λx y. case x of
   Zero → y
   Succ x' → Succ (plus x' y)
```

9.1.4 Dependently-typed data constructor args

Now, even in our “dirt simple” system, we will be able to encode some new structures that are beyond what is available using regular algebraic datatypes 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 become more so. For example, here is 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)
```

In this example, the type of `ImTrue` is

```
ImTrue : (b:Bool) → (b = True) → SillyBool
```

9.2 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 (i.e. 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)

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

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

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
   K4 of (x:A)(y:B) -- two args, the type B can mention x
   K5 of (x:A)[x = a] -- one arg, plus an equality constraint about that arg
```

In this setting, 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 [dB72] 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 to the syntax.[16]

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

-- | data constructor names
type DataConName = String

-- | Declarations stored in the context
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 TyConName [ConstructorDef]

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

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

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

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

9.3 Type checking uses of (simple) data constructors

In general, when we have a datatype declaration, that means that a new data type T of type **Type** will be added to the context. Furthermore, the context should record all of the data constructors for that type, K_i, as well as the telescope, written Δ_i, for each data constructor. For example for natural numbers, the new type T is **Nat**, there are two new data constructors, Zero and Succ, the former has an empty telescope, the latter has a telescope containing the declaration of a single relevant argument of type **Nat**. Alternatively, for the `SillyBool` example, there are again two new data constructors ImTrue and $ImFalse$. Each of these constructors takes a telescope with two entries, first the declaration of a name of type **Bool** and then a definition of that name as either **True** or **False**.

[16] Some parts of this implementation have been elided compared to `full`. We are building up to that version slowly.

The information about datatype declarations stored in the context is used to check terms that are the applications of data constructors. For simplicity, `pi-forall` requires that data constructors must be fully applied to all of their arguments.

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

$$\textsc{i-dcon-simple}\quad \frac{\begin{array}{c} K : \Delta \rightarrow T \;\in \Gamma \\ \Gamma \vdash \overline{a} \Leftarrow \Delta \end{array}}{\Gamma \vdash K\,\overline{a} \Rightarrow T}$$

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

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

$$\textsc{tele-sig}\quad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash \overline{a} \Leftarrow \Delta[a/x]}{\Gamma \vdash a\,\overline{a} \Leftarrow x : A, \Delta}$$

Furthermore, we also substitute when the telescope contains definitions.

$$\textsc{tele-def}\quad \frac{\Gamma \vdash \overline{a} \Leftarrow \Delta[a/x]}{\Gamma \vdash \overline{a} \Leftarrow x = a, \Delta}$$

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.

$$\textsc{tele-nil}\quad \frac{}{\Gamma \vdash \Leftarrow}$$

$\Gamma \vdash \overline{a} \Leftarrow \Delta$	`tcArgTele`	Type check a list of arguments against a telescope
$\Gamma \vdash pat : A \Rightarrow \Delta$	`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 9: Functions for checking datatypes and case expressions

Figure 9 lists the relevant judgments for type checking the use of datatypes and the corresponding functions in the `pi-forall` implementation. The $\Gamma \vdash \overline{a} \Leftarrow \Delta$ judgment, described above, is implemented by the `tcArgTele` function in `TypeCheck.hs`. This function has the following type in the implementation:

(For reasons that we explain below, we have a special type `Arg` for the arguments to the data constructor.)

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

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

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

This substitution function propagates the substitution through the telescope.

9.4 Pattern matching with dirt simple datatypes

In your homework assignment, we used a special `if` term to eliminate boolean types. Now, we would like to replace that with the more general `case` expression, of the form $\mathbf{case}\ a\ \mathbf{of}\ \{\ \overline{pat_i \to a_i}^{\,i}\ \}$ that works with any form of datatype. What should the typing rule for that sort of expression look like? There is a lot of work to do.

$$\textsc{c-case-simple}\quad \dfrac{\begin{array}{c}\mathbf{whnf}\ A \rightsquigarrow T \qquad \begin{array}{c}\Gamma \vdash a \Rightarrow A\\ \overline{\Gamma \vdash pat_i : T \Rightarrow \Delta_i}^{\,i}\end{array} \qquad \overline{\vdash a \sim pat_i \Rightarrow \Delta'_i}^{\,i}\\ \overline{\Gamma, \Delta_i, \Delta'_i \vdash a_i : A}^{\,i} \qquad \mathbf{branches\,exhaustive}\end{array}}{\Gamma \vdash \mathbf{case}\ a\ \mathbf{of}\ \{\ \overline{pat_i \to a_i}^{\,i}\ \} \Leftarrow A}$$

Mathematically, this rule first 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 pat_i and uses the context to calculate a telescope Δ_i containing declarations of all of the variables bound in that pattern. That telescope is added to the context to type check each branch a_i against the type of the entire expression A. Furthermore, when checking the type of each branch, we can refine that type because we know that the scrutinee is equal to the pattern. To reflect this information during type checking, we also construct the telescope Δ'_i.

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 (`ensureTyCon`)
3. For each case alternative $pi \to a_i$:
   - 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 pi (`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`)

9.5 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 `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 Σ-types as a datatype, instead of making them primitive.

```
data TySigma (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
   K1 of D1
   ...
   Kn of Dn
```

That means that when we check an occurrence of a type constructor, we need to make sure that its actual arguments match up with the parameters in the telescope. For this step, we can reuse the same argument checking judgment as we used for data constructor applications.

I-TCON

$$\frac{T : \Delta \to \textbf{Type} \in \Gamma \qquad \Gamma \vdash \overline{a} \Leftarrow \Delta}{\Gamma \vdash T\,\overline{a} \Rightarrow \textbf{Type}}$$

Now, to type check data constructor applications for parameterized types, we need to modify the typing rule for data constructors. This rule below separates the type constructor telescope (Δ_1) from the data constructor telescope Δ_2 when looking up the data constructor from the context. When type checking the arguments of the data constructor $\overline{a}$, we first substitute the arguments of the type constructor $\overline{b}$ into the data constructor telescope.

C-DCON

$$\frac{K : \Delta_1 \to \Delta_2 \to T \;\in \Gamma \qquad \Gamma \vdash \overline{a} \Leftarrow \Delta_2[\overline{b}/\Delta_1]}{\Gamma \vdash K\,\overline{a} \Leftarrow T\,\overline{b}}$$

For example, if we are trying to check the expression `Just True`, with expected type `Maybe Bool`, we will 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 will use to check the argument `True`.

In `TypeCheck.hs`, the function

```
substTele :: [Decl] -> [Arg] -> [Decl] -> TcMonad [Decl]
```

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. For example, if we see `True`, we would like to infer that it has type `Bool` without any annotation.

9.6 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 the `nat` argument an *index* instead of a *parameter* because it varies in the result type of each data constructor. It is not used uniformly in each case.

In `pi-forall`, we implement indices by explicitly *constraining* parameters. These constraints are just expressed as equalities written in square brackets. In other words, we 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. 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`).

- 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.

9.6.1 Homework: 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 one number (called Zero), Fin 2 includes two 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.

9.7 Irrelevance and datatypes

Datatypes may wish to mark some of their arguments as irrelevant, so that these components of the data structure can be erased at run time. For example, the Vec type in Section 2 marked the length of the tail of the vector as irrelevant in the telescope for the Cons constructor.

The mechanisms that we have set up so far extend naturally to irrelevant data constructor arguments. Note however: we can only erase *data* constructor arguments; pi-forall does not allow marking arguments to *type* constructors as irrelevant. This is not a significant limitation as we almost always want these arguments to actually be relevant—we don't want to equate Vec 0 Int with Vec 1 Bool. However, these type parameters are never relevant when they are used with data constructors—we don't even include them in the abstract syntax.

9.7.1 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. Where do we need to pass the length of a list around at run time?

10 Where to go next?

10.1 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].

  This tutorial implements the "LambdaPi" language and is accompanied by a Haskell implementation available online [17]. The tutorial starts with an implementation of the simply-typed lambda calculus, using a locally nameless representation and bidirectional type checking, and then extends to core type system with type-in-type and uses normalization-by-evaluation for equivalence checking. It also adds natural numbers and vectors along with their eliminators, but does not include a general form of (inductive) datatypes.

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

  The goal of Augustsson's tutorial, inspired by an earlier version of LambdaPi, is to be the simplest implementation possible. Therefore, he uses a string representation of variables and binding and weak-head normalization for the implementation of equality. His version is a Haskell implementation of Barendregt's lambda cube [Bar93].

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

  Coquand et al. describe another Haskell implementation of dependent types, called "Mini-TT", which shows how to implement (in about 400 lines of code) a small dependently-typed language that includes Σ-types, void, unit, and sums. The language does not include propositional equality or indexed datatypes, nor does it enforce termination. The implementation uses strings to represent variable binding, which are converted to de Bruijn indices during normalization-by-evaluation.

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

  Bauer explores several different implementations of dependent type theory in a series of blog posts using the OCaml language. The first version uses a named representation (i.e., strings) and generates fresh names during substitution and alpha-equality. This version uses full normalization to implement definitional equality. Then, in the second version, Bauer revises the implementation to use normalization by evaluation. In the third version, he revises again to use de Bruijn indices, explicit substitutions, and switches back to weak-head normalization.

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

[17] `https://www.andres-loeh.de/LambdaPi/`

In a recent blog post tutorial, Rompf uses Javascript to implement a core dependent type theory using higher-order abstract syntax.

- Andras Kovacs, *Elaboration Zoo* [Kov]

  Kovacs' GitHub repository includes several different implementations of evaluation and type checking for a minimal dependently typed languages. These implementations are accompanied by video recordings of a seminar that goes over this code. Notably, this tutorial includes implicit argument synthesis via (higher-order) unification.

10.2 Related work for `pi-forall`

Section 3: Core Dependent Types Because they lack a rule for conversion, the typing rules for the core dependent type system presented in Section 3 do not correspond to any existing type system. However, jumping ahead, the core language with the inclusion of conversion (rule T-CONV) has been well-studied.

As a foundation of logic, the inclusion of rule rule T-TYPE makes the system inconsistent. Indeed, Martin-Löf's original formulation of type theory [ML71] included this rule until the resulting paradox was demonstrated by Girard. This inconsistency was resolved in later versions of the systems [ML75] and in the Calculus of Constructions [CH86] by stratifying **Type**: first into two levels and then into an infinite hierarchy of universes [Luo90].

The inconsistency of this system as a logic does not prevent its use as a programming language. Cardelli [Car86] showed that this language was type sound and explored applications for programming.

Section 4: Bidirectional type checking The expressiveness of dependently-typed programming languages put them out of reach of the Hindley Milner type inference algorithm found in Haskell and ML. As a result, type checkers and elaborators for these languages must abandon complete type inference and require users to annotate their code.

Bidirectional type systems provide a platform for propagating type annotations through judgments, and notably, support a syntax-directed specification of where to apply the conversion rule during type checking. As a result, such type systems are frequently used as the basis for type checking, with elaboration for implicit arguments layered on top.

Pierce and Turner [PT00] popularized the use of bidirectional type systems, in the context of polymorphic languages with subtyping. The Haskell language uses bidirectional type checking layered on top of Hindley-Milner unification in order to support type inference for higher-rank polymorphism [PVWS07]. This type system extension allows functions to take polymorphic functions as arguments. Notably, the paper includes a well-documented reference implementation of the type system in the appendix.

David Christiansen's tutorial [Chr13] introduces the idea of bidirectional type checking as an independent topic. For another description of the role of bidirectional type systems in the context of dependent types, see Löh, McBride and Swierstra's tutorial [LMS10]. Unlike `pi-forall`, this tutorial separates the syntax of terms based whether they are inferable or checkable.

Finally, Dunfield and Krishnaswami's extensive survey paper [DK21] catalogs and organizes recent results in this area.

Section 6: Definitional equality Many implementations, such as Coq and Agda, use a procedure called *normalization-by-evaluation* to decide whether two terms are equivalent. In general, this algorithm is favored due to its speed (it can avoid interpretation overhead by relying on an evaluator) and because it can be extended with typed equalities that derive from η-equivalence. In Coq, this procedure is implemented by compilation to OCaml bytecode, through the procedure described in this paper [GL02]. More detailed information about NbE is available in Abel's habilitation thesis [Abe13]. A simple implementation of NbE as described in this document is available [18].

Sections 7 and 9: Dependently-typed pattern matching The implementation of indexed datatypes in `pi-forall` is inspired by the compilation of *Generalized Algebraic Datatypes* in Haskell [VPJSS11, VWPJ06]. The distinguishing feature of this formalization is that internally, datatype indices are represented by constrained parameters and so are an extension of the parameterized recursive sums of products found in functional programming languages.

In contrast, the theory of inductive datatypes found in Coq and Agda must ensure that they cannot be used to implement nonterminating functions. This means that there are several restrictions that must be applied: inductive datatypes must be strictly positive, their eliminators must be defined separately, and they must satisfy constraints based on universe-levels. For more information about how inductive definitions work in these settings, see the Coq [19] or Agda [20] manuals.

Section 8: Compile-time and run-time irrelevance The implementation of irrelevance in `pi-forall` is inspired by the Dependent Dependency Calculus (DDC) [CEIW22], a pure type system parameterized by a dependency lattice. The selection of the lattice determines the precise form of irrelevance described by the type system: does the system include run-time irrelevance only, compile-time irrelevance only, both sorts of irrelevance combined together (as in `pi-forall`), or both sorts and allow them to be distinguished.

Compile-time irrelevance only Pfenning [Pfe01] used a modal type system to identify arguments that can be ignored when checking equivalence. This system, extended by Abel and Scherer [AS12], became the basis for Agda's implementation of compile-time irrelevance.

[18] `https://github.com/jozefg/nbe-for-mltt`

[19] `https://coq.inria.fr/refman/language/core/inductive.html#theory-of-inductive-definitions`

[20] `https://agda.readthedocs.io/en/v2.6.2.2/language/data-types.html`

Run-time irrelevance only The Coq proof assistance distinguishes the `Prop` universe from the `Type` universe for several reasons. However, one feature of this distinction is that all terms in the `Prop` universe can be erased prior to execution, leading to faster evaluation of extracted code.

More recently, *quantitative type theory* [Atk18] unifies run-time irrelevance and linear type systems by restricting the number of times that variables may be used in a given context. This resulting system has been adopted for the implementations of irrelevance in the Agda and Idris [Bra21] languages.

Both run-time and compile-time irrelevance, but no distinction The Implicit Calculus of Constructions [Miq01] ensures that some arguments are treated irrelevantly (at both run time and compile time) by simply not including them in the syntax of terms. This work was extended to decidable type checking through the use of an erasure operation—terms may include irrelevant arguments, but these arguments must be erased prior to run time and equality checking[BB08]. This approach lead to Erasure Pure Type Systems (EPTS) [MLS08], and the treatment of irrelevance in Dependent Haskell [WVdAE17].

Acknowledgments

Thanks to "Team Weirich" from OPLSS 2022 (Kartik Singhal, Jonathan Chan, Shengyi Jiang and Thomas Binetruy), the Penn PLClub, and the attendees of OPLSS 2022, OPLSS 2014 and OPLSS 2013, for feedback and suggestions about these lecture notes. Some parts of the implementation This work was supported by the National Science Foundation under grants NSF 2006535, NSF 1703835 and NSF 0910786.

References

[Abe13] Andreas Abel. *Normalization by Evaluation: Dependent Types and Impredicativity.* Habilitation thesis, Ludwig-Maximilians-Universität München, 2013. URL: `http://www.cse.chalmers.se/~abela/habil.pdf`.

[AS12] Andreas Abel and Gabriel Scherer. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science*, 8(1), 2012. `doi:10.2168/lmcs-8(1:29)2012`.

[Atk18] Robert Atkey. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 56–65, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3209108.3209189`.

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

[Bar93] H. P. Barendregt. *Lambda Calculi with Types*, pages 117–309. Oxford University Press, Inc., USA, 1993.

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

[BB08] Bruno Barras and Bruno Bernardo. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In Roberto Amadio, editor, *Foundations of Software Science and Computational Structures*, pages 365–379, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-78499-9_26.

[Bra21] Edwin C. Brady. Idris 2: Quantitative Type Theory in Practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ECOOP.2021.9.

[Car86] Luca Cardelli. A Polymorphic λ Calculus with Type:Type. Technical Report 10, Digital Equipment Corporation, SRC, 1986. URL: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-10.pdf.

[CEIW22] Pritam Choudhury, Harley Eades III, and Stephanie Weirich. A Dependent Dependency Calculus. In Ilya Sergey, editor, *Programming Languages and Systems, ESOP 2022*, volume 13240 of *Lecture Notes in Computer Science*, pages 403–430, Cham, 2022. Springer International Publishing. URL: https://github.com/sweirich/graded-haskell, doi:10.1007/978-3-030-99336-8_15.

[CH86] T. Coquand and Gérard Huet. The calculus of constructions. Technical Report RR-0530, INRIA, 1986. URL: https://hal.inria.fr/inria-00076024.

[Chr13] David Raymond Christiansen. Bidirectional Typing Rules: A Tutorial. Tutorial, 2013. URL: http://davidchristiansen.dk/tutorials/bidirectional.pdf.

[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*, pages 139–164. Cambridge University Press, 2009. doi:10.1017/CBO9780511770524.007.

[dB72] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.

[DK21] Jana Dunfield and Neel Krishnaswami. Bidirectional Typing. *ACM Comput. Surv.*, 54(5), 2021. doi:10.1145/3450952.

[GL02] Benjamin Grégoire and Xavier Leroy. A Compiled Implementation of Strong Reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 235–246, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/581478.581501.

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

[Kov] Andras Kovacs. Elaboration Zoo. 2022. https://github.com/AndrasKovacs/elaboration-zoo/.

[LMS10] Andres Löh, Conor McBride, and Wouter Swierstra. A Tutorial Implementation of a Dependently Typed Lambda Calculus. *Fundam. Inform.*, 102:177–207, 2010. Available from http://www.andres-loeh.de/LambdaPi/. doi:10.3233/FI-2010-304.

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990. URL: http://hdl.handle.net/1842/12487.

[Miq01] Alexandre Miquel. The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications*, pages 344–359, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. doi:10.1007/3-540-45413-6_27.

[ML71] Per Martin-Löf. A Theory of Types. Preprint, Stockholm University, 1971. URL: https://raw.githubusercontent.com/michaelt/martin-lof/master/pdfs/martin-loef1971%20-%20A%20Theory%20of%20Types.pdf.

[ML75] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73, Proceedings of the Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975. doi:10.1016/S0049-237X(08)71945-1.

[MLS08] Nathan Mishra-Linger and Tim Sheard. Erasure and Polymorphism in Pure Type Systems. In *Proceedings of the Theory and Practice of Software, 11th International Conference on Foundations of Software Science and Computational Structures*, FOSSACS'08/ETAPS'08, pages 350–364, Berlin, Heidelberg, 2008. Springer-Verlag. `doi:10.1007/978-3-540-78499-9_25`.

[Pfe01] Frank Pfenning. Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, LICS '01, pages 221–230, Washington, DC, USA, 2001. IEEE Computer Society. URL: `https://www.cs.cmu.edu/~fp/papers/lics01.pdf`, `doi:10.1109/LICS.2001.932499`.

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

[PT00] Benjamin C. Pierce and David N. Turner. Local Type Inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000. `doi:10.1145/345099.345100`.

[PVWS07] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1), 2007. `doi:10.1017/S0956796806006034`.

[Rom20] Tiark Rompf. Implementing Dependent Types, 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, pages 1–12, New York, NY, USA, 2007. Association for Computing Machinery. `doi:10.1145/1291151.1291155`.

[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. `doi:10.1145/3018743.3018773`.

[VPJSS11] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X): Modular type inference with local assumptions. *Journal of Functional Programming*, 21:333–412, 2011. `doi:10.1017/S0956796811000098`.

[VWPJ06] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming (ICFP'06)*. ACM SIGPLAN, 2006. 2016 ACM SIGPLAN Most Influential ICFP Paper Award. URL: `https://www.microsoft.com/en-us/research/publication/simple-unification-based-type-inference-for-gadts/`, `doi:10.1145/1159803.1159811`.

[WVdAE17] Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.*, 1(ICFP):31:1–31:29, 2017. `doi:10.1145/3110275`.