Implementing Dependent Types in `pi-forall`

Stephanie Weirich

May 22, 2022

1 Goals and What to Expect

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

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

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

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

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

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

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

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

5. Templates and tools for writing about programming languages and type systems. The source files for these lecture notes are also available in the same repository as the demo implementation. Building these notes requires the Ott tool [**?**], specifically tailored for typesetting mathematical specifications of programming languages and type systems. You may wish to take a look at how this all works.

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.

- Section 2 presents the mathematical specification of the core type system including its abstract syntax (written in mathematics), concrete syntax (as found in `pi-forall` source files) and core typing rules.

- Our initial specification of the type system is declarative. It *specifies* what terms should type check, but cannot be directly implemented. Therefore, Section 3 reformulates the typing rules so that they are *syntax-directed* and specify a type checking algorithm. Ask yourself "How much information do we need to include in terms to make type checking algorithmic?"

- Section 4 introduces the core `pi-forall` implementation and walks through the Haskell implementation of the typing rules discussed in Section 3. Ask yourself "How can we represent the abstract syntax of the language, including variable binding?"

- Section 5 discusses the role of definitional equality in dependently typed languages. After motivating examples, it presents both a specification of when terms are equal and a semi-decidable algorithm that can be incorporated into the type checker. Ask yourself "How can we know when types (and terms) are equal?"

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

- Section 7 introduces the idea of tracking the *relevence* of arguments. Ask

yourself "How can we decide what parts of the term are irrelevant during computation? Can be ignored when checking for equality?"

- Section 8 generalizes pattern matching to arbitrary datatypes.
- The appendix includes the complete specification of the full `pi-forall` language.

`version1`	Sections 234
`version2`	Sections 5 and 6
`version3`	Section 7
`full`	Section 8

What do these notes not cover? The goal of this notes is to provide an introductory overview. As a result, there are many topics related to the implementation of dependent type theories that are not included in these notes.

- For simplicity, the language is not terminating. To implement something like Agda or Coq, where typechecking implies termination, requires additional structure, including universe levels and bounded iteration.
- Many implementations of dependent type theories use *normalization-by-evaluation* to decide equivalence at compile time. `pi-forall` uses an alternative equivalence algorithm based on weak-head-normalization using substitution. This approach is closer to lambda calculus theory, but can be less efficient in practice.
- For simplicity, `pi-forall` does not attempt to synthesize missing type 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 the implementation of variable binding, alpha-equivalence and substitution, which uses a locally nameless representation. As a result, most issues related to variable binding are avoided here.
- Recent work on cubical type theory, higher-inductive types and univalence is not covered here.

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

2 A Simple Core language with Type-in-Type

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

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

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

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

Figure 1: Typing rules for core system

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

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

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

2.1 When do expressions in this language type check?

We define the type system for this language using an inductively defined relation defined in Figure 1. This relation is between an expression a, its type A, and a typing context Γ.

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

The typing context is an ordered list of assumptions about the types of variables. We assume that the variables in this list are distinct from each other, so that there will always be at most one assumption about the type of any variable.

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

An initial set of typing rules: Variables and Functions Consider the first two rules, rule T-VAR and rule T-LAMBDA, shown in Figure 1. The first

rule states that the types of variables are recorded in the typing context. The premise $x : A \in \Gamma$ says that we can find the association between x and the type A in the list Γ.

The second rule, introduces variables into the context when we type check abstractions.

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

```
id :: forall a. a -> a
id y = y
```

and it's Haskell type is generic: we can apply this function to any type of argument. We can also write a polymorphic identity function in `pi-forall`, except that we will need to explicitly make this function polymorphic.

In `pi-forall`, this definition looks like below, where the 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.

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

The type of `id` has the general form $(x : A) \to B$, like the usual form of function type $A \to B$, except that we name the argument x so that it can be referred to in the body of the type B. In this example, the type argument x is used later as the type of y. We call types of this form Π types because they are usually written as $\Pi x : A.B$ in formalizations of dependent type theory.[1]

The fact that the type of x is **Type** means that x plays the role of a type variable, such as `a` in Haskell. 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.

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

More typing rules: Types Observe in the typing derivation above that the rule for typing lambda-expressions has an additional precondition: we need to make sure that when we add assumptions $x : A$ to the context, that A really

[1] The terminology for these types is muddled: sometimes they are call dependent function

is a type. Without this precondition, the rules would allow us to derive this nonsensical type for the polymorphic identity function.

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

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 rather perplexing rule T-TYPE. Because the type of the polymorphic identity function starts with $(x : \mathbf{Type}) \to \ldots$ the rule T-PI rule means that **Type** must be a type for this Π-type to make sense. We declare this by fiat using the rule T-TYPE rule. [2]

More typing rules: Application 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 to substitute the argument 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', then we can apply 'id' to 'id'.

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

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

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

true : bool
true = λx. λy. λz. y

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

types and sometimes they are called dependent product types. We use the non Π notation to emphasize the connection to functions.

[2] Note that, this rule make our language inconsistent as a logic, as it can encode Girard's paradox. More about this in section 9.

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

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

Example: logical "and" We can also encode a logical "and" data structure using a Church encoding.

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

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

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

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

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

3 From typing rules to a typing algorithm

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

A type system is called *syntax-directed* if it is readily apparent how to turn a collection of typing rules into code. In other words, we would like to implement the following Haskell function, that when given a term and a typing context, produces the type of the term, if it exists.[3]

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

Let's look at our rules in Figure 1. Is this straightforward? For example, in the variable rule, as long as we can lookup the type of a variable in the context, we can produce its type. That means that, assuming that `lookupTy` has type `Ctx -> TName -> 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.

[3]NOTE: If you are looking at the `pi-forall` implementation, this is not the final type of

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

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

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

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. Furthremore, there is enough information there for type checking—wherever we define a function, we can look at those types to know what type its argument should have. So, by changing our point of view, we can get away without annotating lambdas with those argument types.

3.1 A bidirectional type system

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

We will express these judgements using the notation defined in Figure 2 and implement them in Haskell using the mutually-recursive functions `inferType` and `checkType`. Furthermore, to keep track of which rule is in which judgement, rules that have inference as a conclusion will start with I- and rules that have checking as a conclusion will start with C-.

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

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

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

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

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

this function.

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

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

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

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

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

Figure 2: (Simple) Bidirectional type system

applicable with rule C-INFER-SIMPLE that allows us to use inference whenever a checking rule doesn't apply.

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

Well, they involve applications of explicit lambda terms:

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

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

In fact, the bidirectional type system has the property that it only checks terms in *normal* form, i.e. those that do not contain any β-reductions.

3.2 Type annotations

To type check nonnormal forms in `pi-forall`, we add typing annotations, written $(a : A)$ and add rule I-ANNOT.

Type annotations allow us to supply known type information anywhere within a term. For example, we can construct this derivation.

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

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:

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

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

A not so desirable property of this bidirectional system is that it is not closed under substitution. The types of variables are always inferred. This is particularly annoying in the application rule when we replace a variable (inference mode) with a term that is correct in checking mode.

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

One solution to this problem is to add an annotation before substitution. In our implementation of `pi-forall`, we do not do so to reduce clutter. Alternatively, we can solve the problem through *elaboration*, the output of a type checker will be a term that works purely in inference mode.

4 Putting it all together in a Haskell implementation

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

First, an overview of the main files of the implementation. There are a few more source files, but these are the primary ones:

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

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

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

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

In our implementation, we use the `Unbound` library to get all of these operations for free. This library defines a type for variable names, called `Name`. This type is indexed by the AST that this is a name for. That way `Unbound` can make sure that we only substitute the right form of syntax for the right name, i.e. we can only replace a `TName` with a `Term`.

```
type TName = Unbound.Name Term
```

The `Unbound` library includes an overloaded function `subst x a b`, which means $b[a/x]$[4], i.e. substitute a for x in b. In general, we can apply a substitution to any sort of syntax; all substitution functions should match the following pattern.

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

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 `Unbound` library can automatically generate instances of the `Subst` class. Furthermore, although it seems like we only need to substitute within terms, we'll actually need to have substitution available at many types.

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

```
data Term

  = -- | type of types  `Type`
    Type

  | -- | variables  `x`
    Var TName
```

[4] See Guy Steele's talk about the notation for substitution [Ste17]. This is the most common

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

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

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

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

  ...
  deriving (Show, Generic)

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

As you can see, variables are represented by names. `Unbound`'s `Bind` type constructor declares the scope of the bound variables. Both `Lam` and `Pi` bind a single variable in a `Term`.

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

```
type Type = Term
```

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

The definitions of the `Arg` datatype automatically derives instances for two type classes from the `unbound-generics` library: `Unbound.Alpha` and `Unbound.Subst`.

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

```
aeq :: Alpha a => a -> a -> Bool
```

For `Term`, we do not use the default definitions of the `Alpha` and `Subst` classes. Instead, we need to provide a bit more information. First, we would like the definition of alpha-equivalence of terms to ignore type annotations (as well as other practicalities, such as source code positions for error messages and explicit parentheses). Therefore our instance does so for these constructors of the `Term` data type and defers to the generic definition for the rest.

The `Subst` instance for `Term` requires telling `Unbound` where the variables:

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

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

4.0.2 A TypeChecking monad [Environment.hs]

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

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

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

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

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

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

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

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

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


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

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

---

mathematical notation for this operation.

4.0.3 Implementing the TypeChecking Algorithm [Typecheck.hs]

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

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

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

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

```
tcTerm :: Term -> Maybe Type -> TcMonad Type
```

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

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

tcTerm Type Nothing = ...

tcTerm (Pi bnd) Nothing = ...

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

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

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

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

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

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

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

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

4.0.4 Example

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

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

Some fairly standard typing rules for booleans are shown below.

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

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

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

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

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

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

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

A Sigma-type is a product where the type of the second component of the product can depend on the first component. We destruct Sigmas using pattern matching. A simple rule for pattern matching introduces variables into the context when pattern matching the sigma type. These variables are not allowed to appear free in the result type of the pattern match.

This part of the homework has two parts:

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

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

5 Equality in Dependently-Typed Languages

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

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

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

5.1 Motivating Example: Type level reduction

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

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

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

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

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

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

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

```
In the expression
   a p ((\x . \y . x))
```

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

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

5.2 Another example needing more expressive equality

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

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

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

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

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

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

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

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

5.3 Defining definitional equality

The main idea is that we will:

- establish a new judgement that defines when types are equal

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

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

$$\textsc{E-BETA}\ \frac{}{\Gamma \vdash (\lambda x.a)\ b = a[b/x]} \qquad \textsc{E-REFL}\ \frac{}{\Gamma \vdash A = A} \qquad \textsc{E-SYM}\ \frac{\Gamma \vdash A = B}{\Gamma \vdash B = A}$$

$$\textsc{E-TRANS}\ \frac{\Gamma \vdash A = B \quad \texttt{<<multiple parses>>}}{\texttt{<<multiple parses>>}} \qquad \textsc{E-PI}\ \frac{\Gamma \vdash A_1 = A_2 \quad \Gamma, x:A_1 \vdash B_1 = B_2}{\Gamma \vdash (x:A_1) \to B_1 = (x:A_2) \to B_2}$$

$$\textsc{E-LAM}\ \frac{\Gamma, x:A_1 \vdash b_1 = b_2}{\Gamma \vdash \lambda x.b_1 = \lambda x.b_2} \qquad \textsc{E-APP}\ \frac{\Gamma \vdash a_1 = a_2 \quad \Gamma \vdash b_1 = b_2}{\Gamma \vdash a_1\ b_1 = a_2\ b_2}$$

$$\textsc{E-SUBST}\ \frac{\Gamma \vdash a = a' \quad \Gamma \vdash b = b'}{\Gamma \vdash \textbf{subst}\ a\ \textbf{by}\ b = \textbf{subst}\ a'\ \textbf{by}\ b'}$$

Figure 3: Definitional equality for core `pi-forall`

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

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

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

What is a good definition of equality? We started with a very simple one: alpha-equivalence. But we can do better. Check out the rules in Figure 3.

Rule E-BETA ensures that our relation *contains beta-equivalence*. Rules E-REFL, E-SYM, and E-TRANS makes sure that this relation is an *equivalence relation*. Furthermore, this relation should be a *congruence relation* (i.e. if subterms are equal, then larger terms are equal), as verified by rules rules E-PI, E-LAM, and E-APP. Finally, we want to be sure that this relation has "functionality" (i.e. we can lift equalities). We declare so using rule E-SUBST.

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

5.4 Using definitional equality in the algorithm

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

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

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

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

  $$\text{C-INFER}\quad \frac{\Gamma \vdash a \Rightarrow A \qquad \Gamma \vdash A = B}{\Gamma \vdash a \Leftarrow B}$$

  In this case, our equality algorithm must, when given two terms, decide whether they are equal. In `pi-forall` we will use a semi-decision procedure based on reducing terms to normal forms. However, because reduction may not terminate, our equality checking function could diverge.

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

  $$\text{I-APP}\quad \frac{\Gamma \vdash a \Rightarrow A \qquad \mathbf{whnf}\ A = (x{:}A_1) \to B \qquad \Gamma \vdash b \Leftarrow A_1}{\Gamma \vdash a\ b \Rightarrow B[b/x]}$$

  In this case, we are given a single term 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. Just evaluate the type A to its head form, using the rules shown in Figure 4. If that form is a Π-type, then we can access its domain type.

  For closed terms, these rules correspond to a big-step evaluation relation. This relation is semi-decidable, so we can express it as a Haskell function that may diverge.

5.5 Algorithmic definitional equality

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

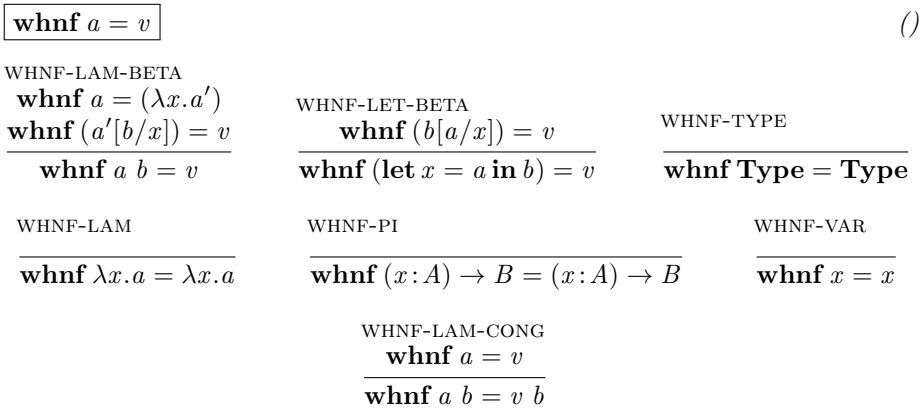

Figure 4: Weak-head normal form reduction

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

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

Second, we also need to be able to determine whether a given type is equal to some "head" form, without knowing exactly what that form is. For example, when *checking* lambda expressions, we need to know that the provided type is of the form of a Π-type $((x : A) \to B)$. Likewise, when inferring the type of an application, we need to know that the type inferred for the function is actually a Π-type.

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

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

checks the given type to see if it is equal to some pi type of the form $(x : A_1) \to A_2$, and if so returns `x`, `A1` and `A2`.
This function is defined in terms of a helper function, that implements the rules shown in Figure 4.

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

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

(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:

λy. (λx.x) (λx.x)

and

(y:Type) → (λx.x) Bool

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

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

5.6 Implementing definitional equality (see Equal.hs)

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

One way to do this is with the following algorithm:

```
equate t1 t2 = do
   nf1 <- reduce t1  -- full reduction, throughout term
   nf2 <- reduce t2
   Unbound.aeq nf1 nf2
```

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

```
equate t1 t2 = do
   if (Unbound.aeq t1 t1) then return () else do
    nf1 <- whnf t1  -- reduce only to 'weak head normal form'
    nf2 <- whnf t2
    case (nf1,nf2) of
      (App a1 a2, App b1 b2) ->
          -- make sure subterms are equal
          equate a1 b1 >> equate a2 b2
      (Lam bnd1, Lam bnd2) -> do
          -- ignore variable names, but use the name
          -- fresh name for both lambda bodies
          (_, b1, _, b2) <- unbind2Plus bnd1 bnd2
           equate b1 b2
      (_,_) -> err ...
```

Therefore, we reuse our mechanism for reducing terms to weak-head normal form.

Why weak-head reduction vs. full reduction?

- We can implement deferred substitutions for variables. Note that when comparing terms we need to have the definitions available. That way we can compute that `(plus 3 1)` weak-head normalizes to 4, by looking up the definition of `plus` when needed. However, we don't want to substitute all variables through eagerly—not only does this make extra work, but error messages can be extremely long.

- Furthermore, we allow recursive definitions in `pi-forall`, so normalization may just fail completely. However, this definition based on `whnf` only unfolds recursive definitions when necessary, and then only once, so avoids some infinite loops in the type checker.

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

6 Dependent pattern matching and propositional equality

6.1 Refining rules for if and Σ-type elimination

Consider our elimination rules for if:

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

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

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

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

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

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

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

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

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

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

$$\text{C-IF}\quad \frac{\Gamma \vdash x \Rightarrow \mathbf{Bool} \qquad \texttt{<<multiple parses>>} \qquad \texttt{<<multiple parses>>}}{\Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 \Leftarrow A}$$

To implement this rule, we need only remember that x is **True** or **False** when checking the individual branches of the if expression. [5]

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

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

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

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

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

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

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

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

$$\text{I-IF-ALT}\quad \frac{\Gamma \vdash a \Rightarrow \mathbf{Bool} \qquad \Gamma \vdash b_1 \Rightarrow B_1 \qquad \Gamma \vdash b_2 \Rightarrow B_2}{\Gamma \vdash \mathbf{if}\ a\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 \Rightarrow \mathbf{if}\ a\ \mathbf{then}\ B_1\ \mathbf{else}\ B_2}$$

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

$$\text{E-IF-ETA}\quad \frac{}{\Gamma \vdash \mathbf{if}\ a\ \mathbf{then}\ b\ \mathbf{else}\ b = b}$$

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

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

$$\textsc{t-refl}\ \frac{\Gamma \vdash a = b}{\Gamma \vdash \mathbf{refl} : a = b} \qquad \textsc{t-eq}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a = b : \mathbf{Type}}$$

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

Figure 5: Propositional equality

6.2 Propositional equality

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

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

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

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

The main idea of the equality type is that it converts a *judgement* that two types are equal into a *type* that evaluates to a value only when two types are equal.[6] In other words, the new value form **refl** has type $a = b$, as shown in rule T-REFL.

Sometimes, you might see therule written as follows:

$$\textsc{t-refl-alt}\ \frac{}{\Gamma \vdash \mathbf{refl} : a = a}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

One last addition: `contra`. If we can somehow prove a false statement, then we should be able to prove anything. A contradiction is a proposition between two terms that have different head forms. For now, we'll use rule T-CONTRA, shown in Figure 5.

6.2.1 Homework (`pi-forall`: more church encodings)

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

6.2.2 Homework (`pi-forall`: equality)

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

7 Irrelevance: the ∀ of `pi-forall`

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

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

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

We'll take another approach.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

7.2 Compile-time irrelevance

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

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

$$\textsc{e-eapp}\ \frac{\Gamma \vdash a_1 = a_2}{\Gamma \vdash a_1[b_1] = a_2[b_2]}$$

We've been alluding to this the whole time, but now we'll come down to it.

We're actually *defining* equality over just the computationally relevant parts of the term instead of the entire term. In `version3` of the implementatin, note how the definition of `equate` ignores arguments that are tagged as 'irrelevant'. (And from the beginning, our system already ignores types that appear only in type annotations. Our justification for doing this is the same.)

Why is compile-time irrelevance important?

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

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

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

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

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

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

8 Datatypes and Indexed Datatypes

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

Unfortunately, datatypes are both:

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

Unlike the previous sections, where we could walk through all of the details

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

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

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

8.1 "Dirt simple" datatypes

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

8.1.1 Booleans

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

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

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

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

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

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

8.1.2 Void

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

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

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

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

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

8.1.3 Nat

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

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

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

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

8.1.4 Dependently-typed data constructor args

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

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

8.2 Specifying the type system with basic datatypes

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

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

(T-Nat?)(T-Void?)(T-Zero?)(T-Succ?)(T-ImTrue?)

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

```
data T : Type where
  K1       -- no arguments
  K2 of (A)    -- single arg of type A
  K3 of (x:A)  -- also single arg of type A, called x for fun
  K4 of (x:A)(y:B) -- two args, the type of B can mention A.
```

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

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

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

-- | data constructor names
type DCName = String

data Decl = ...
  | Data TCName [ConstructorDef]

-- | A Data constructor has a name and a telescope of arguments
data ConstructorDef = ConstructorDef DCName Telescope
      deriving (Show,Unbound.Alpha, Unbound.Subst Term)

newtype Telescope = Telescope [Assn]
  deriving (Show, Generic)
  deriving anyclass (Unbound.Alpha, Unbound.Subst Term)

data Assn
  = AssnSig Sig
  | AssnEq Term Term
  deriving (Show, Generic, Unbound.Alpha, Unbound.Subst Term)
```

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

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

8.3 Checking (simple) data constructor applications

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

So our typing rule looks a little like this. We have `as` as representing the list of arguments for the data constructor `Ki`.

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

```
Ki : Di -> T  in G
G |- as : Di
------------------------ simpl-constr
G |- Ki as : T
```

We need to check that list against the telescope for the constructor. Each argument must have the right type. Furthermore, because of dependency, we substitute that argument for the variable in the rest of the telescope.

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

```
G |- a : A       G |- as : D { a / x }
---------------------------------------- tele-arg
G |- a as : (x:A) D
```

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

```
---------- tele-empty
G |-  :
```

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

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

This function relies on the following substitution function for telescopes:

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

8.4 Eliminating dirt simple datatypes

In your homework assignment, we used if to eliminate boolean types. Here, we'd like to be more general, and have a `case` expression that works with any form of datatype. What should the typing rule for that sort of expression look like? Well, the pattern for each branch should match up the telescope for the corresponding data constructor.

```
G |- a : T
 Ki : Di -> T  in G
 G, Di |- ai : A
 G |- A : Type
 branches exhaustive
------------------------------------ case-simple
G |- case a of { Ki xsi -> ai } : A
```

Note that this version of case doesn't witness the equality between the scrutinee `a` and each of the patterns in the branches. To allow that, we can add a substiution to the result type of the case:

```
G |- a : T
 Ki : Di -> T  in G       dom(Di) = xsi
 G, Di |- ai : A { Ki xsi / x }
 G |- A : T -> Type
 branches exhaustive
------------------------------------------- case
G |- case a of { Ki xsi -> ai } : A { a / x}
```

How do we implement this rule in our language? The general for type checking a case expression `Case scrut alts` of type `ty` is as follows:

1. Infer type of the scrutinee `scrut`
2. Make sure that the inferred type is some type constructor (`ensureTCon`)
3. Make sure that the patterns in the case alts are exhaustive (`exhausivityCheck`)
4. For each case alternative:

- Create the declarations for the variables in the pattern (`declarePat`)
- Create defs that follow from equating the scrutinee `a` with the pattern (`equateWithPat`)
- Check the body of the case in the extended context against the expected type

8.5 Datatypes with parameters

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

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

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

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

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

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

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

```
  T : D -> Type in G
    G |- as : D
  --------------------   tcon
  G |- T as : Type
```

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

```
  Ki : D . Di -> T  in G
  G |- as : Di { bs / D }
    ------------------------ param-constr
    G |- Ki as : T bs
```

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

In `TypeCheck.hs`, the function

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

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

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

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

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

8.6 Datatypes with indices

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

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

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

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

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

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

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

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

```
G |- x = b      G |- as : D
------------------------------------- tele-constraint
G |- as : (x = b) D
```

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

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

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

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

8.6.1 Homework: Parameterized datatypes and proofs: logic

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

8.6.2 Homework: Indexed datatypes: finite numbers in `Fin1.pi`

The module `Fin1.pi` declares the type of numbers that are drawn from some bounded set. For example, the type `Fin 1` only includes 1 number (called Zero), `Fin 2` includes 2 numbers, etc. More generally, `Fin n` is the type of all natural numbers smaller than `n`, i.e. of all valid indices for lists of size `n`.

In Agda, we might declare these numbers as:

In pi-forall, this corresponding definition makes the constraints explicit:

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

The file Fin1.pi includes a number of definitions that use these types. However, there are some `TRUSTME`s. Replace these with the actual definitions.

8.7 Erasure and datatypes

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

Note: we can only erase *data* constructor arguments, not types that appear

as arguments to *type* constructors. (Parameters to type constructors must always be relevant, they determine the actual type.) On the other hand, datatype parameters are never relevant to data constructors—we don't even represent them in the abstract syntax.

8.7.1 Homework: Erasure and Indexed datatypes: finite numbers in `Fin1.pi`

Now take your code in `Fin1.pi` and see if you can mark some of the components of the `Fin` datatype as eraseable.

9 Where to go for more

Tutorials on the implmentation of dependent type systems

- A. Löh, C. McBride, W. Swierstra, *A tutorial implementation of a dependently typed lambda calculus*`http://www.andres-loeh.de/LambdaPi/` [LMS10].

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

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

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

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

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

- Lennart Augustsson [Aug07]

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

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

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

> "We allow definitions of non-terminating functions. This is essential if Mini-TT is going to be a core language for programming. Non-terminating functions are essential for interactive programs. Of course, it causes problems for type-checking to be terminating, so we assume that termination is checked in a separate phase."

Bidirectional type checking * Pierce and Turner

* Peyton Jones, Vytiniotis, Weirich, Shields. [Practical type inference for arbitrary-rank types](https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/putting.pdf)

* Christiansen, Tutorial on Bidirectional Typing. https://www.davidchristiansen.dk/tutorials/bidirectional.

* Dunfield and Krishnaswami, [Bidirectional Typing](https://www.cl.cam.ac.uk/ nk480/bidir-survey.pdf)

Normalization by evaluation * Andreas Abel's habilitation thesis.

* Daniel Gratzer, NBE for MLTT https://github.com/jozefg/nbe-for-mltt

Elaboration and type inference * Pierce and Turner (Local argument synthesis)

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

* Pientka

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

Topic areas: Representing lambda terms, Bidirectional Typing, Equality checking in Dependent Type Theories, Implicit argument inference (elaboration), Compile-time and runtime irrelevance.

* Augustsson, [Cayenne – a Language With Dependent Types](http://dl.acm.org/citation.cfm?id=289451)

* Cardelli, [A polymorphic lambda calculus with Type:Type](http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-10.pdf)

* Friedman and Christiansen, The Little Typer. [FC18]

"Pie" Implementation in Racket. Not really a tutorial implementation. Instead an introduction to dependent type theory based on a core language implemented in Racket. The Uses normalization by evaluation.

1. Type system does not enforce termination. That makes it convenient as a programming language, but less so as a logic. We do it this way because it sidesteps a number of issues in the implementation (universe levels, strict positivity in datatypes).

2. Implementation of equivalence checking based on substitution, not normalization by evaluation.

3. Details with variable binding hidden by the `Unbound` library. This library uses a locally nameless representation and generic programming to automatically derive implementations of substitution and alpha-equivalence.

Other tutorials describe how to represent variables using a locally nameless representation (McBride et al.) or de Bruijn representation (??) more directly.

9.1 References

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

References

[Aug07] Lennart Augustsson. Simpler, easier!, 2007.

[Bau12] Andrej Bauer. How to implement dependent type theory, November 2012. blog post.

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

[FC18] Daniel P. Friedman and David Thrane Christiansen. *The Little Typer*. MIT Press, Cambridge, MA, USA, September 2018.

[LMS10] Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundam. Inform.*, 102:177–207, 01 2010.

[Rom20] Tiark Rompf. Implementing dependent types, December 2020. blog post.

[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.

A Full specification of the system