Checking Dependent Types with Normalization by Evaluation: A Tutorial (Haskell Version)

David Thrane Christiansen

Draft of May 21, 2019

To implement dependent types, we need to be able to determine when two types are the same. In simple type systems, this process is a fairly straightforward structural equality check, but as the expressive power of a type system increases, this equality check becomes more difficult. In particular, when types can contain programs, we need to be able to run these programs and check whether their results are the same. *Normalization by evaluation* is one way of performing this sameness check, while *bidirectional type checking* is a technique that guides the invocation of the checks.

These notes are a collection of literate programs that demonstrate how to derive a normalization procedure from an evaluator or interpreter, and then how to use this normalization procedure to write a type checker for a small dependently typed language based on the language Pie[1] from *The Little Typer* (Friedman and Christiansen, 2018).

These notes are written assuming that you know the following:

- The basics of Haskell, including pattern matching, how to define datatypes and functions, and `do`-notation
- The basics of dependent types, equivalent to having worked through the first half of *The Little Typer* or the first part of the Idris tutorial[2]

Understanding some sections also requires familiarity with inference rules, but these sections can be safely skipped. The appendix to *The Little Typer* describes how to read inference rules.

In the interest of accessibility, the Haskell features used in this tutorial are kept to a minimal subset. Haskell experts will likely find many ways in which the code can be shortened or made more elegant.

[1] Available from `http://thelittletyper.com`
[2] `http://docs.idris-lang.org/en/latest/tutorial/`

Contents

1 Evaluating Untyped λ-Calculus

Let's start with an evaluator for the untyped λ-calculus. Writing an evaluator requires the following steps:

- Identify the *values* that are to be the result of evaluation
- Figure out which expressions become values immediately, and which require computation
- Implement datatypes for the values and helper functions for computation

In this case, for the untyped λ-calculus, the only values available are *closures*, and computation occurs when a closure is applied to another value.

1.1 Syntax

In this section, the following representation of the untyped λ-calculus is used:

```
data Expr
  = Var Name
  | Lambda Name Expr
  | App Expr Expr
  deriving Show
```

This corresponds closely to the typical presentation as a grammar:

$$\begin{array}{rcl} e & ::= & x \\ & | & \lambda x.e \\ & | & (e\ e) \end{array}$$

with the usual conventions regarding associativity and precedence of lambda and application, namely that the body of a λ extends as far as possible to the right and that function application associates to the left. For instance, $\lambda f.f\ f\ f$ corresponds to $\lambda f.((f\ f)\ f)$, not $((\lambda f.f)\ f)\ f$ or $\lambda f.(f\ (f\ f))$.

1.2 Values and Runtime Environments

A closure packages an expression that has not yet been evaluated with the runtime environment in which the expression was created. Here, closures always represent expressions with a variable that will be instantiated with a value, so these closures additionally keep track of a name.

```
data Value
  = VClosure (Env Value) Name Expr
    deriving Show
```

Runtime environments provide the values for each variable. By convention, runtime environments are referred to with the Greek letter "ρ," which can be written “rho” and is pronounced like “row.” In this implementation, environments are association lists, containing pairs of variable names and values. Earlier values override later values in the list, and the initial environment is empty. The type of environments is polymorphic over the contained values so that it can be used for other value representations later.

```
newtype Name = Name String
  deriving (Show, Eq)
newtype Env val = Env [(Name, val)]
  deriving Show
initEnv :: Env val
initEnv = Env []
```

It is possible to map a function over an Env, so it is a Functor.

```
instance Functor Env where
  fmap f (Env xs) =
    Env (map (λ(x, v) → (x, f v)) xs)
```

Looking up a value in an environment might fail, so the function that performs the lookup might return a message instead of a value.

```
newtype Message = Message String
  deriving Show
failure :: String → Either Message a
failure msg = Left (Message msg)
lookupVar :: Env val → Name → Either Message val
lookupVar (Env [])           (Name x) =
  failure ("Not found: " ++ x)
lookupVar (Env ((y, v) : env')) x
  | y == x    = Right v
  | otherwise = lookupVar (Env env') x
```

Because *lookupVar* finds the *first* value for the name in the environment, extending an environment with a new value should attach that new value at the front.

```
extend :: Env val → Name → val → Env val
extend (Env env) x v = Env ((x, v) : env)
```

1.3 The Evaluator

The evaluator in figure 1 consists of two procedures: *eval* evaluates an expression in a run-time environment that provides values for its free variables, and *doApply* is responsible for applying the value of a function to the value of its argument.

```
eval :: Env Value → Expr → Either Message Value
eval env (Var x)             =
  lookupVar env x
eval env (Lambda x body)     =
  Right (VClosure env x body)
eval env (App rator rand)    =
  do fun ← eval env rator
     arg ← eval env rand
     doApply fun arg

doApply :: Value → Value → Either Message Value
doApply (VClosure env x body) arg =
  eval (extend env x arg) body
```

Figure 1: The evaluator

Evaluating a variable looks up the variable's name in the environment, evaluating a function produces a closure using the current environment, and evaluating an application delegates to *doApply* after evaluating the function and its argument. *doApply* evaluates the function's body in its original environment, which is extended with the argument to reflect the fact that the argument's value is now known. The names *rator* and *rand* are short for "operator" and "operand." These names go back to Landin (1964).

1.4 Programs with Definitions

Real programs aren't single expressions; typically, programmers write code in meaningful parts. To support this way of working, we can add definitions to the little untyped programming language. Definitions provide the initial contents of the environment, which means that adding a definition can be done by first evaluating the body of the definition, and then adding that result to a list of definitions.

```
runProgram :: [(Name, Expr)] → Expr → Either Message Value
runProgram defs expr =
  do env ← addDefs initEnv defs
     eval env expr

addDefs :: Env Value → [(Name, Expr)] → Either Message (Env Value)
addDefs env []             = Right env
addDefs env ((x, e) : defs) =
  do v ← eval env e
     addDefs (extend env x v) defs
```

This approach to definitions means that there is no mutual recursion, because new definitions are only in scope in the rest of the program. Additionally, because the only values are closures, the return values of programs are not very interesting to look at. Section 3 demonstrates how to recover readable code from these closure values.

1.5 Example: Church Numerals

The *Church numerals* are an encoding of the natural numbers as programs in the untyped λ-calculus. The basic idea is that a number n is represented by a function that takes another function and some starting value, and applies the function n times to the starting value. This means that zero is represented by $\lambda f.\lambda x.x$, and the function that adds one to its argument is represented by $\lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$. Addition of j and k is represented by $\lambda j.\lambda k.\lambda f.\lambda x.j\ f\ (k\ f\ x)$. Figure 2 defines the Church numerals in a form suitable for *addDefs*.

These can be used to test the evaluator:

```
test :: Either Message Value
test =
  runProgram churchDefs
    (App (App (Var (Name "+"))
              (toChurch 2))
         (toChurch 3))
```

Unfortunately, it is somewhat difficult to see whether the test was successful. Evaluating `test` in `ghci` yields the output in figure 3, which is not very easy to read, even after omitting the details of some closures in the environment.

2 Interlude: Fresh Names

Normalization requires generating fresh names to avoid conflicting variable names. There are many strategies for generating fresh names. Here, the overall approach is meant to be simple and to generate names that make sense to humans. The technique is to track the used, and thus unavailable, names. When a fresh name is needed, take some starting name and append tick marks until it is not included in the used names.

```
freshen :: [Name] → Name → Name
freshen used x
  | elem x used = freshen used (nextName x)
  | otherwise   = x

nextName :: Name → Name
nextName (Name x) = Name (x ++ "'")
```

```
churchDefs :: [(Name, Expr)]
churchDefs =
  [ (Name "zero",
      Lambda (Name "f")
        (Lambda (Name "x")
          (Var (Name "x"))))
  , (Name "add1",
      Lambda (Name "n")
        (Lambda (Name "f")
          (Lambda (Name "x")
            (App (Var (Name "f"))
                 (App (App (Var (Name "n"))
                           (Var (Name "f")))
                      (Var (Name "x")))))))
  , (Name "+",
      Lambda (Name "j")
        (Lambda (Name "k")
          (Lambda (Name "f")
            (Lambda (Name "x")
              (App (App (Var (Name "j")) (Var (Name "f")))
                   (App (App (Var (Name "k")) (Var (Name "f")))
                        (Var (Name "x"))))))))
  ]
toChurch :: Integer → Expr
toChurch n
  | n ⩽ 0      = Var (Name "zero")
  | otherwise = App (Var (Name "add1")) (toChurch (n − 1))
```

Figure 2: Church numerals

```
Right
 (VClosure
  (Env [ (Name "k",
          VClosure
            (Env [...])
            (Name "f")
            (Lambda (Name "x") ...))
       , (Name "j",
          VClosure
            (Env [...])
            (Name "f")
            (Lambda (Name "x") ...))
       , (Name "add1",
          VClosure
            (Env [...])
            (Name "n")
            (Lambda (Name "f") ...))
       , (Name "zero",
          VClosure
            (Env [])
            (Name "f")
            (Lambda (Name "x")
              (Var (Name "x"))))])
  (Name "f")
  (Lambda (Name "x")
    (App (App (Var (Name "j")) (Var (Name "f")))
         (App (App (Var (Name "k")) (Var (Name "f")))
              (Var (Name "x"))))))
```

Figure 3: The value of $2 + 3$ in Church numerals

3 Normalizing Untyped λ-Calculus

Expressions in the λ-calculus are not defined only by the grammar of expressions. There is also an equational theory that tells us when two expressions mean the same thing. The first rule is that consistently renaming bound variables doesn't change the meaning of an expression, a property referred to as *α-equivalence*. The second rule is that applying a λ-expression to an argument is equal to the result of the application, a rule called *β*. Expressions equated by zero or more α and β steps are called *αβ-equivalent*. It is important to remember that both rules are *equations*, which means that they can be applied anywhere in an expression and that they can be read both from left to right and from right to left.

Expressed in mathematical notation, the β rule relies on *substitution*, which is consistently replacing bound occurrences of a variable with some other expression in such a way as to not capture any variables. The operation of replacing x with e_2 in e_1 is written $e_1[e_2/x]$. Think of it as dividing by x, thus removing each occurrence, and then putting an e_2 into each empty space.

$$(\lambda x.e_1)\ e_2 \equiv e_1[e_2/x]$$

When we have a collection of equations over syntax, the syntax can be seen as divided into various "buckets," where each expression in a bucket is αβ-equivalent to all the others in its bucket. One way to check whether two expressions are in the same bucket is to assign each bucket a representative expression and provide a way to find the bucket representative for any given expression. Then, if two expressions are in the same bucket, they will have the same representative. This canonical representative is referred to as a *normal form* for the collection of expressions that are equal to each other–that is, the expressions in the same "bucket."

Here, we adopt the convention that normal forms are those that contain no reducible expressions, or *redexes*, which is to say that there are no λ-expressions directly applied to an argument. Because α-equivalence is easier to check than β-equivalence, most people consider normal forms with respect to the β-rule only, and then use α-equivalence when comparing β-normal forms.

Reading the β rule from left to right, we see that it is always possible to replace a redex with the result of the substitution. Another way to view β-normal forms is as expressions in which all redexes have been replaced.

In the untyped λ-calculus, not every expression has a normal form. For instance, the redex $(\lambda f.f\ \ f)\ (\lambda g.g\ \ g)$ computes via β to $(\lambda g.g\ \ g)\ (\lambda g.g\ \ g)$, which is α-equivalent to the original expression. Once a type system is added, it will become possible to rule out these infinite loops such that *every* well-typed expression will have a normal form.

3.1 Finding Normal Forms

One way to find the normal form of an expression would be to repeatedly traverse it, performing all possible β-reductions. However, this is extremely inefficient–first, a redex must be located, and then a new expression constructed by applying the β rule. Then, the context around the former redex must be reconstructed to point at the newly-constructed expression. Doing this for each and every redex is not particularly efficient, and the resulting code is typically not pleasant to read.

Alternatively, the environment-based evaluator from section 1 can be modified to do normalization by adding a second step that reads values back into their syntax. It is much more efficient to re-use our evaluator, adding cases to handle reductions in the bodies of λ-expressions, because the expression does not need to be traversed as many times. Additionally, it is much easier to implement an evaluator with environments than it is to correctly implement substitution, which is surprisingly subtle. By carefully choosing the set of values to *only* represent expressions that do not contain redexes, we can be very confident that our normalization procedure actually does produce normal forms with respect to β-conversion.

When reducing under λ, there will also be variables that do not have a value in the environment. To handle these cases, we need values that represent *neutral* expressions. A neutral expression is an expression that we are not yet able to reduce to a value, because information such as the value of an argument to a function is not yet known. In this language, there are two neutral expressions: variables that do not yet have a value, and applications where the function position is neutral. Normal forms are either neutral, atoms, or λ-expressions with normal bodies.

The neutral expressions and normal forms can be captured in a grammar. They cannot represent redexes, because the rator in neutral applications can never be a λ-expression.

$$\begin{array}{rcl} \mathit{Neu} & ::= & x \\ & | & \mathit{Neu}\ \mathit{Norm} \\ \mathit{Norm} & ::= & \mathit{Neu} \\ & | & \texttt{'}a \\ & | & \lambda x.\mathit{Norm} \end{array}$$

The evaluator from section 1 can be extended to become a *normalizer*, a program that computes normal forms instead of values. This is done by extending the value datatype to contain neutral expressions, and arranging for function application to construct a larger neutral expression when neutrals are applied to arguments. In these datatypes, Value plays the role of *Norm*. The reading-back step will transform the values into the syntax of normal forms.

```
data Value
  = VClosure (Env Value) Name Expr
  | VNeutral Neutral
   deriving (Show)
data Neutral
  = NVar Name
  | NApp Neutral Value
  deriving (Show)
```

Perhaps surprisingly, *eval* remains unchanged. The additional case is handled in the updated *doApply*.

```
eval :: Env Value → Expr → Either Message Value
eval env (Var x)            =
  lookupVar env x
eval env (Lambda x body) =
  return (VClosure env x body)
eval env (App rator rand) =
  do fun ← eval env rator
     arg ← eval env rand
     doApply fun arg

doApply (VClosure env x body) arg =
  eval (extend env x arg) body
doApply (VNeutral neu)        arg =
  Right (VNeutral (NApp neu arg))
```

We can now work with values that are neutral, but every neutral value begins with a neutral variable, and those are not introduced anywhere. Additionally, our values are clearly not the normal forms of expressions. The second step in implementing a normalizer is to write a procedure to convert the values back into their representations as syntax–a process referred to as *reading back* or *quoting* values into syntax.

```
readBack :: [Name] → Value → Either Message Expr
readBack used (VNeutral (NVar x))         = Right (Var x)
readBack used (VNeutral (NApp fun arg)) =
  do rator ← readBack used (VNeutral fun)
     rand  ← readBack used arg
     Right (App rator rand)
readBack used fun@(VClosure _ x _) =
  do let x' = freshen used x
     bodyVal  ← doApply fun (VNeutral (NVar x'))
     bodyExpr ← readBack (x' : used) bodyVal
     Right (Lambda x' bodyExpr)
```

The combination of evaluation and reading back leads to normalization.

```
normalize :: Expr → Either Message Expr
normalize expr =
  do val ← eval initEnv expr
     readBack [] val
```

Definitions and programs work similarly, except it is now possible to show the normal form of the expression, rather than just its value. The names used in definitions are considered already used, so that references to them do not get captured.

```
runProgram :: [(Name, Expr)] → Expr → Either Message Expr
runProgram defs expr =
  do env ← addDefs initEnv defs
     val  ← eval env expr
     readBack (map fst defs) val
addDefs :: Env Value → [(Name, Expr)] → Either Message (Env Value)
addDefs env []              = Right env
addDefs env ((x, e) : defs) =
  do v ← eval env e
     addDefs (extend env x v) defs
```

The test from section 1.5 is much easier to interpret. Given the following definition,

```
test :: Either Message Expr
test =
  runProgram churchDefs
    (App (App (Var (Name "+"))
              (toChurch 2))
         (toChurch 3))
```

evaluating `test` in `ghci` yields:

```
Right
  (Lambda (Name "f")
    (Lambda (Name "x")
      (App  (Var (Name "f"))
            (App  (Var (Name "f"))
                  (App (Var (Name "f"))
                       (App  (Var (Name "f"))
                             (App  (Var (Name "f"))
                                   (Var (Name "x")))))))))
```

It is much easier to check that this is the Church numeral for five, namely

$$\lambda f.\lambda x.f\ (f\ (f\ (f\ (f\ x))))$$

than it was for the value in section 1.5.

4 Bidirectional Type Checking

Type systems specify the conditions under which particular expressions can have particular types, but there is no guarantee that we have an algorithm to efficiently check this. Sometimes, however, an algorithm can be read directly from the rules that define the type system. This typically occurs when at most one rule applies in any-given situation. Because the syntax of the program and the type determine which choice to take, this property is called being *syntax-directed*.

There are a number of ways in which type systems might not be syntax-directed. One way is that there might be insufficient information in the program to determine the type. For example, languages like ML in which all types can be inferred typically require a type checking algorithm that looks very little like the rules. Another alternative is to require that programs be annotated with types in enough locations to allow the type rules to follow the annotations. Another way in which type systems frequently fail to be syntax-directed is by having a complicated notion of type equality or subsumption, which makes it so that the type checker at each step could either transform a type or follow the program's syntax. These are not the only ways in which types might fail to be syntax-directed, but they occur frequently.

An example of a type system that is not syntax-directed is the simply-typed λ-calculus with natural numbers but without type annotations on functions:

$$\frac{}{\Gamma_1, x : t, \Gamma_2 \vdash x : t} \qquad \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \to t_2} \qquad \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2}$$

$$\frac{}{\Gamma \vdash \mathsf{zero} : \mathsf{Nat}} \qquad \frac{\Gamma \vdash n : \mathsf{Nat}}{\Gamma \vdash \mathsf{add1}\ n : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash n : \mathsf{Nat} \quad \Gamma \vdash b : t_b \quad \Gamma \vdash s : \mathsf{Nat} \to t_b \to t_b}{\Gamma \vdash \mathsf{rec}\ n\ b\ s : t_b}$$

If the rules are read as the source code of a program that checks an expression against a type, the rule for $\lambda x.e$ is not immediately translatable, because there is nowhere to get the type t_1 from. While it is traditional to write the argument type t_1 on the function itself, with a syntax such as $\lambda x : t_1.e$, this can make programs unpleasant to read, especially if the involved types are large and complex.

Bidirectional type checking is a general technique for making type systems syntax-directed that adds only a minimal annotation burden. Typically, only the top level of an expression or any explicit redexes need to be annotated. Additionally, bidirectional type checking provides guidance for the appropriate places to insert checks of type equality or subsumption.

4.1 Types

The type system in this section is the simply-typed λ-calculus with natural numbers and primitive recursion. There are two ways of constructing types: Nat is a type, and $t_1 \to t_2$ is a type when t_1 is a type and t_2 is a type.

$$\begin{array}{rcl} t & ::= & \mathsf{Nat} \\ & | & t \to t \end{array}$$

$$
\begin{array}{rcl}
e & ::= & x \\
  & | & \lambda x.e \\
  & | & (e\ e) \\
  & | & \mathsf{zero} \\
  & | & \mathsf{add1}\ e \\
  & | & \mathsf{rec}[t]\ e\ e\ e \\
  & | & e \in t
\end{array}
$$

```
data Expr = Var Name
          | Lambda Name Expr
          | App Expr Expr
          | Zero
          | Add1 Expr
          | Rec Ty Expr Expr Expr
          | Ann Expr Ty
  deriving Show
```

Figure 4: Expressions in Bidirectional Simply-Typed λ-Calculus

This grammar translates directly to a Haskell datatype.

```
data Ty
  = TNat
  | TArr Ty Ty
  deriving (Eq, Show)
```

4.2 Checking Types

When writing a bidirectional type checker, the first step is to classify the expressions in the programming language into *introduction* and *elimination* forms. The introduction forms, also called *constructors*, allow members of a type to be created, while the eliminators expose the information inside of the constructors to computation. In this section, the constructor of the function type is λ and the constructors of Nat are zero and add1. The eliminators are function application and rec.

Under bidirectional type checking, the type system is split into two modes: in *checking* mode, an expression is analyzed against a known type to see if it fits, while in *synthesis* mode, a type is derived directly from an expression. Each expression for which a type can be synthesized can be checked against a given type by performing the synthesis and then comparing the synthesized type to the desired type. This is where subsumption or some other nontrivial type equality check can be inserted. Additionally, type annotations (here, written $e \in A$) allow an expression that can be checked to be used where synthesis is required. Usually, introduction forms have checking rules, while elimination forms admit synthesis. The complete syntax used in this section can be found in figure 4.

Computation steps (that is, redexes) arise when eliminators encounter constructors. Typically, most programs do not include explicit redexes, which is a key reason for the effectiveness of bidirectional type checking at reducing the burden of annotations–annotations are required only when a checkable expression (introduction form) is used in a synthesis position (typically a target of elimination). Because these are rare, annotations are typically required only at the outermost level surrounding an expression, and sometimes not even there.

For example, a function that returns its second argument would normally be written

$$(\lambda x.\lambda y.y) \in \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$

rather than

$$(\lambda x.\lambda y.((\lambda z.z) \in \mathsf{Nat} \to \mathsf{Nat})\ y) \in \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$

because there is no reason to include the redex in the program.

In mathematical notation, the bidirectional version of the type system splits the form of judgment $\Gamma \vdash e : t$ into two forms of judgment: $\Gamma \vdash e \Rightarrow t$, which means that type t can be synthesized from e, and $\Gamma \vdash e \Leftarrow t$, which means that it is possible to check that e has type t. The direction of arrow indicates the flow of type information. When checking, both the type and the expression are considered to be inputs to the algorithm, while in synthesis, only the expression is an input while the type is an output. One way to remember which form of judgment is which is to observe that the tip of $\Leftarrow$ looks a bit like the letter "C."

When reading bidirectional type systems, start below the line. If the conclusion is synthesis, then the rule applies when the expression matches. If the conclusion is checking, then the rule applies when both the expression and the type match the rule. In other words, below the line, inputs are matched while outputs are constructed. Inputs bind metavariables, while outputs refer to already-bound metavariables. Having discovered that a rule matches, next check the premises above the line from left to right and from top to bottom, in the order in which one would read English. In the premises, inputs construct data, so their metavariables must already be bound, while outputs are matched against data and can bind metavariables.

The first two rules dictate how the system changes between checking and synthesis. If there is a type annotation, then synthesis will succeed when the expression can be checked at the appropriate type. If we can synthesize a type, then we can check it by first synthesizing a type and then checking whether the synthesized type is equal to the desired type.

$$\frac{\Gamma \vdash e \Leftarrow t_1}{\Gamma \vdash e \in t_1 \Rightarrow t_1} \qquad \frac{\Gamma \vdash e \Rightarrow t_2 \quad t_2 = t_1}{\Gamma \vdash e \Leftarrow t_1}$$

A type can always be synthesized for a variable, by looking it up in Γ. For functions, we avoid having to invent a type for the bound variable by checking against any arrow type. Similarly, we avoid having to invent the argument type in an application by requiring that a type can be synthesized for the function being applied, and then checking the argument against the function's argument type.

$$\frac{}{\Gamma_1, x : A, \Gamma_2 \vdash x \Rightarrow A} \qquad \frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.e \Leftarrow A \to B}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \quad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}$$

In general, introduction forms require type checking, not synthesis, although it is possible to implement Nat using synthesis because its constructors are so simple. But lists, for example, do not have an explicit indication of the type of entries in the list, so checking them against a known list type allows that information to be extracted and propagated. Recursive type checking of arguments to constructors should be done in checking mode, because the type of the constructor will contain enough information to discover the types of the arguments. In this tutorial, all constructors' types are checked.

$$\frac{}{\Gamma \vdash \mathsf{zero} \Leftarrow \mathsf{Nat}} \qquad \frac{\Gamma \vdash n \Leftarrow \mathsf{Nat}}{\Gamma \vdash \mathsf{add1}\ n \Leftarrow \mathsf{Nat}}$$

Types for elimination forms are generally synthesized, rather than checked. The type of the target should be synthesized, while this should provide sufficient type information so that the types of the remaining arguments can be checked. This does require that immediate redexes have a type annotation around the target to mediate between checking and synthesis. However, because most actual programs do not contain explicit redexes, and because eliminations and variables all admit synthesis, the targets that occur in practice do not require annotations.

$$\frac{\Gamma \vdash n \Rightarrow \mathsf{Nat} \quad \Gamma \vdash b \Leftarrow A \quad \Gamma \vdash s \Leftarrow \mathsf{Nat} \to A \to A}{\Gamma \vdash \mathsf{rec}[A]\ n\ b\ s \Rightarrow A}$$

Expressed as a program, these rules become the functions *synth* and *check*. The typing context Γ can be represented using the Env type from section 1, because a typing context is a bit like a compile-time version of an environment. Instead of mapping names to values, the context maps names to types. To keep the distinction between contexts and values more clear in the code, it's useful to have a type synonym.

```
type Context = Env Ty
initCtx :: Context
initCtx = initEnv
```

Type synthesis is represented by *synth*, and type checking by *check*. Each returns Left on failure. On success, *synth* returns the type that was synthesized, while *check* returns only an indication that checking was successful.

```
synth :: Context → Expr → Either Message Ty
check :: Context → Expr → Ty → Either Message ()
```

4.2.1 Type Synthesis

Variables The first case in synthesis, the variable rule, can delegate to *lookupVar*.

$$\frac{}{\Gamma_1, x : t, \Gamma_2 \vdash x : t}$$

```
$synth\ ctx$ (Var $x$) =
  $lookupVar\ ctx\ x$
```

Function application Function application is implemented by first synthesizing a type for the function, and then checking the argument against the argument type.

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \quad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}$$

```
$synth\ ctx$ (App $rator\ rand$) =
  do $ty \leftarrow synth\ ctx\ rator$
     case $ty$ of
       TArr $argT\ retT$ →
         do $check\ ctx\ rand\ argT$
            Right $retT$
       $other$          →
         $failure$ ("Not a function type: " ++ $show\ other$)
```

Recursion Recursion over natural numbers is checked by first ensuring that the number being recurred over is a Nat, and then finding the appropriate types for the base and step.

$$\frac{\Gamma \vdash n \Rightarrow \mathsf{Nat} \quad \Gamma \vdash b \Leftarrow A \quad \Gamma \vdash s \Leftarrow \mathsf{Nat} \to A \to A}{\Gamma \vdash \mathsf{rec}[A]\ n\ b\ s \Rightarrow A}$$

```
$synth\ ctx$ (Rec $ty\ tgt\ base\ step$) =
  do $tgtT \leftarrow synth\ ctx\ tgt$
     case $tgtT$ of
       TNat →
         do $check\ ctx\ base\ tgtT$
            $check\ ctx\ step$ (TArr TNat (TArr $ty\ ty$))
            Right $ty$
       $other$ →
         $failure$ ("Not the type Nat: " ++ $show\ other$)
```

Annotations A type annotation is an opportunity to change modes, from synthesis to checking.

$$\frac{\Gamma \vdash e \Leftarrow t_1}{\Gamma \vdash e \in t_1 \Rightarrow t_1}$$

```
synth ctx (Ann e t) =
  do check ctx e t
     Right t
```

Failure Finally, if no other synthesis rule applied, then type checking fails with a friendly message.

```
synth _ other =
  failure ("Can't find a type for " ++ show other ++ ". " ++
           "Try adding a type annotation.")
```

4.2.2 Type Checking

Lambda To check a function, the expected type should be an arrow type.

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

```
check  ctx (Lambda x body) t =
  case t of
    TArr arg ret →
      check (extend ctx x arg) body ret
    other →
      failure ("Lambda requires a function type, but got " ++
               show other)
```

Zero zero can only be checked against the type Nat. It would certainly be possible to synthesize a type for zero, but there are constructors such as the empty list that require checking due to a lack of available type information.

$$\frac{}{\Gamma \vdash \mathsf{zero} \Leftarrow \mathsf{Nat}}$$

```
check ctx Zero t =
  case t of
    TNat → Right ()
    other →
      failure ("Zero should be a Nat, but was used where a " ++
               show other ++ " was expected")
```

Successor Similarly to zero, it would be possible to synthesize a type for add1, but not for all constructors.

$$\frac{\Gamma \vdash n \Leftarrow \mathsf{Nat}}{\Gamma \vdash \mathsf{add1}\ n \Leftarrow \mathsf{Nat}}$$

```
check  ctx (Add1 n) t =
  case t of
    TNat →
      check ctx n TNat
    other →
      failure ("Add1 should be a Nat, but was used where a " ++
              show other ++ " was expected")
```

Mode change Changing from checking to synthesis requires no annotation, but it does require a type equality check.

$$\frac{\Gamma \vdash e \Rightarrow t_2 \quad t_2 = t_1}{\Gamma \vdash e \Leftarrow t_1}$$

```
check ctx other t =
  do t' ← synth ctx other
     if t' == t
       then Right ()
       else  failure ("Expected " ++ show t ++
                      " but got " ++ show t')
```

4.3 Programs with Definitions

When running programs for their value, or finding the normal form of an expression in a program, definitions extend ρ, the run-time environment. Similarly, for type checking, definitions extend the context Γ.

To keep the implementation as simple as possible, there is no separate syntax for type annotations on definitions. Instead, the body of each definition is expected to support type synthesis, and Ann can be used to annotate checked forms such as functions.

```
addDefs :: Context → [(Name, Expr)] → Either Message Context
addDefs ctx []              = Right ctx
addDefs ctx ((x, e) : defs) =
  do t ← synth ctx e
     addDefs (extend ctx x t) defs
```

One reasonable test is to define two, three, and addition, and then type check both a partial and full application of addition.

```
test :: Either Message (Ty, Ty)
test =
  do ctx ← addDefs initCtx
              [(Name "two",
                 (Ann (Add1 (Add1 Zero))
                      TNat)),
               (Name "three",
                 (Ann (Add1 (Add1 (Add1 Zero)))
                      TNat)),
               (Name "+",
                 (Ann (Lambda (Name "n")
                        (Lambda (Name "k")
                          (Rec TNat (Var (Name "n"))
                            (Var (Name "k"))
                            (Lambda (Name "pred")
                              (Lambda (Name "almostSum")
                                (Add1 (Var (Name "almostSum"))))))))
                      (TArr TNat (TArr TNat TNat))))]
     t₁ ← synth ctx (App (Var (Name "+")) (Var (Name "three")))
     t₂ ← synth ctx (App (App (Var (Name "+")) (Var (Name "three")))
                         (Var (Name "two")))
     Right (t₁, t₂)
```

Evaluating `test` in `ghci` yields:

```
Right (TArr TNat TNat,TNat)
```

as expected.

5 Typed Normalization by Evaluation

Types are more than just collections of programs. Types can also specify which of these programs are equivalent to each other. Writing $\Gamma \vdash e_1 \equiv e_2 : t$ means that the type t considers the expressions e_1 and e_2 to be equivalent, and presumes that both e_1 and e_2 have type t. In this section, we use the following rules:

$$\frac{\Gamma \vdash f : t_1 \to t_2 \quad x \notin \mathrm{FV}(f)}{\lambda x.f\ x \equiv f : t_1 \to t_2}(\eta) \qquad \frac{\Gamma, x : t_1 \vdash e_1 \equiv e_2 : t_2}{\Gamma \vdash \lambda x.e_1 \equiv \lambda x.e_2 : t_1 \to t_2}$$

$$\frac{\Gamma, x : t_1 \vdash e_2 : t_2 \quad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (\lambda x.e_2)\ e_1 \equiv [e_1/x]\, e_2 : t_2}$$

$$\frac{}{\Gamma \vdash \mathsf{zero} \equiv \mathsf{zero} : \mathsf{Nat}} \qquad \frac{\Gamma \vdash e_1 \equiv e_2 : \mathsf{Nat}}{\Gamma \vdash \mathsf{add1}\ e_1 \equiv \mathsf{add1}\ e_2 : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash b : t \quad \Gamma \vdash s : \mathsf{Nat} \to t \to t}{\Gamma \vdash \mathsf{rec}[t]\ \mathsf{zero}\ b\ s \equiv b : t}$$

$$\frac{\Gamma \vdash n : \mathsf{Nat} \quad \Gamma \vdash b : t \quad \Gamma \vdash s : \mathsf{Nat} \to t \to t}{\Gamma \vdash \mathsf{rec}[t]\ (\mathsf{add1}\ n)\ b\ s \equiv s\ n\ \ (\mathsf{rec}\,[t]\ \ n\ b\ s) : t}$$

Because types are an intrinsic part of equality, we are free to adopt rules at a type that are not necessarily given by the reduction of eliminators applied to constructors–some rules introduce constructors around eliminators, such as the η rule for functions above.

Later, when checking dependent types, the equality judgment will be checked by the computer. The more expressions that the computer can equate, the less tedious it is to use the resulting language. However, normal forms are more than just expressions that do not contain redexes. To be a normal form is to be *the* canonical representative of the equivalence class induced by the equality judgment, such that comparing normal forms for α-equivalence is sufficient to decide whether two expressions are equal.

One technique for solving this is extending normalization by evaluation to take types into account. Reductions still occur in the evaluator, as before, but the typed version of the read-back procedure takes the types into account to perform η-expansion.

5.1 Values for Typed NbE

The evaluator in this section contains additional values for the natural numbers. Unlike untyped normalization by evaluation, the typed version needs to keep track of the types of neutral expressions so that it can ensure that they are η-expanded when necessary. The constructor VNeutral is extended to track these types. This embedding of expressions into values is frequently referred to as *reflection*, and written with an up arrow $\uparrow$.

Similarly, the type of values is augmented with a type of saved normal forms called Normal, which contains values packaged up with their types. The process of recovering expressions in normal form from types and values is commonly referred to as *reification*, making a value into an expression, and it is frequently written with a down arrow $\downarrow$.

```
data Value
  = VZero
  | VAdd1 Value
  | VClosure (Env Value) Name Expr
  | VNeutral Ty Neutral
    deriving (Show)
data Neutral
  = NVar Name
  | NApp Neutral Normal
  | NRec Ty Neutral Normal Normal
    deriving (Show)
data Normal
  = Normal {normalType :: Ty,
                normalValue :: Value}
        deriving (Show)
```

5.2 The Evaluator

Just as in section 3, normalization consists of evaluation followed by reading back. Here, introduction and elimination rules for natural numbers are included.

The evaluator works in essentially the same way as the evaluator for untyped normalization. Constructor expressions become values, while eliminators delegate to helpers that either compute the right answer when the target is a value, or construct larger neutral terms when the target is neutral.

Because there is now a type checker for expressions, the evaluator need not return Either Message Value and can return Value. Any errors now represent bugs in the type checker rather than in the user's program, so the evaluator can just crash.

```
eval :: Env Value → Expr → Value
eval env (Var x)               =
  case lookupVar env x of
    Left msg →
      error ("Internal error: " ++
               show msg)
    Right v   → v
eval env (Lambda x body)       =
  VClosure env x body
eval env (App rator rand)      =
  doApply (eval env rator) (eval env rand)
eval env Zero                  = VZero
eval env (Add1 n)              = VAdd1 (eval env n)
eval env (Rec t tgt base step) =
  doRec t (eval env tgt) (eval env base) (eval env step)
eval env (Ann e t)             = eval env e
```

The implementations of eliminators are very similar to untyped normalization. Because equality is based on *types*, embedded normal expressions (such as the operand in an application) contain type annotations. This is because reading back neutral expressions must perform η-expansion. Likewise, the type annotation surrounding the neutral expressions is used to compute the appropriate type for each embedded normal form.

```
doApply :: Value → Value → Value
doApply (VClosure env x body)        arg =
  eval (extend env x arg) body
doApply (VNeutral (TArr t1 t2) neu) arg =
  VNeutral t2 (NApp neu (Normal t1 arg))
doRec :: Ty → Value → Value → Value → Value
doRec t VZero              base step = base
doRec t (VAdd1 n)          base step =
  doApply (doApply step n)
          (doRec t n base step)
doRec t (VNeutral TNat neu) base step =
  VNeutral t
    (NRec t neu
      (Normal t base)
      (Normal (TArr TNat (TArr t t)) step))
```

5.3 Typed Read-Back

In untyped normalization by evaluation, values were examined to determine how to read them back. In typed NbE, however, each type can specify its own notion of equality, and thus the syntax of its normal forms. Therefore, reading back is now recursive on the structure of the *type* rather than the structure of the value.

```
readBackNormal :: [Name] → Normal → Expr
readBackNormal used (Normal t v) = readBack used t v

readBack :: [Name] → Ty → Value → Expr
readBack used TNat       VZero        =
  Zero
readBack used TNat       (VAdd1 pred) =
  Add1 (readBack used TNat pred)
readBack used (TArr t1 t2) fun        =
  let  x    = freshen used (argName fun)
       xVal = VNeutral t1 (NVar x)
  in Lambda x
    (readBack used t2
      (doApply fun xVal))
  where
    argName (VClosure _ x _) = x
    argName _                = Name "x"
readBack used t1 (VNeutral t2 neu)    =
    – Note: checking t1 and t2 for equality here is a good way to find bugs,
    – but is not strictly necessary.
  if t1 == t2
    then readBackNeutral used neu
    else error "Internal error: mismatched types at readBackNeutral"

readBackNeutral :: [Name] → Neutral → Expr
readBackNeutral used (NVar x)           =
  Var x
readBackNeutral used (NApp rator arg)   =
  App (readBackNeutral used rator) (readBackNormal used arg)
```

```
readBackNeutral used (NRec t neu base step) =
  Rec t (readBackNeutral used neu)
    (readBackNormal used base)
    (readBackNormal used step)
```

5.4 Programs With Definitions

Just as in untyped languages, it is more convenient to use a system with definitions. Additionally, some of the benefits of bidirectional type checking do not become apparent until a language has top-level definitions, because the reduction in annotations is not as apparent.

A collection of definitions needs both the types and the values of the names, because they will be used both for type checking and evaluation. Handily, there is already a datatype which has both of these: Normal. Accordingly, it can be converted to either a context or an environment.

```
type Defs = Env Normal
noDefs :: Defs
noDefs = initEnv
defsToContext :: Defs → Context
defsToContext defs = fmap normalType defs
defsToEnv :: Defs → Env Value
defsToEnv defs = fmap normalValue defs


normWithDefs :: Defs → Expr → Either Message Normal
normWithDefs defs e =
  do t ← synth (defsToContext defs) e
     let v = eval (defsToEnv defs) e
     Right (Normal t v)


addDefs :: Defs → [(Name, Expr)] → Either Message Defs
addDefs defs [] =
  Right defs
addDefs defs ((x, e) : more) =
  do norm ← normWithDefs defs e
     addDefs (extend defs x norm) more
definedNames :: Defs → [Name]
definedNames (Env defs) = map fst defs
```

Reprising the definitions from section 4.3

```
normWithTestDefs :: Expr → Either Message Expr
normWithTestDefs e =
  do defs ← addDefs noDefs
```

```
                    [(Name "two",
                      (Ann (Add1 (Add1 Zero))
                           TNat)),
                     (Name "three",
                      (Ann (Add1 (Add1 (Add1 Zero)))
                           TNat)),
                     (Name "+",
                      (Ann (Lambda (Name "n")
                              (Lambda (Name "k")
                                (Rec TNat (Var (Name "n"))
                                  (Var (Name "k"))
                                  (Lambda (Name "pred")
                                    (Lambda (Name "almostSum")
                                      (Add1 (Var (Name "almostSum"))))))))
                           (TArr TNat (TArr TNat TNat))))]
        norm ← normWithDefs defs e
        Right (readBackNormal (definedNames defs) norm)
```

```
test1, test2, test3 :: Either Message Expr
test1 = normWithTestDefs (Var (Name "+"))
test2 = normWithTestDefs (App (Var (Name "+"))
                              (Var (Name "three")))
test3 = normWithTestDefs (App (App (Var (Name "+"))
                                   (Var (Name "three")))
                              (Var (Name "two")))
```

Evaluating `test1`, `test1`, and `test1` respectively yields

```
Right
  (Lambda (Name "n")
   (Lambda (Name "k")
    (Rec TNat (Var (Name "n"))
      (Var (Name "k"))
      (Lambda (Name "pred")
       (Lambda (Name "almostSum")
        (Add1 (Var (Name "almostSum"))))))))
```

```
Right
  (Lambda (Name "k")
   (Add1 (Add1 (Add1 (Var (Name "k"))))))
```

and

```
Right (Add1 (Add1 (Add1 (Add1 (Add1 Zero)))))
```

6 A Tiny Piece of Pie

Now it's time to put all the pieces together and write a type checker for a tiny dependently-typed language, called Tartlet. Tartlet is very much like the language Pie from *The Little Typer*, except it has fewer features and simpler rules. Tartlet contains functions, pairs, the natural numbers, atoms, and the unit and empty types. Also, the Tartlet type of types, $\mathcal{U}$, is a $\mathcal{U}$. This makes it inconsistent as a logic, but it is still safe as a programming language.

```
data Expr
  = Var Name                       – x
  | Pi Name Expr Expr              – (Π ((x A)) B)
  | Lambda Name Expr               – (λ (x) b)
  | App Expr Expr                  – (rator rand)
  | Sigma Name Expr Expr           – (Σ ((x A)) D)
  | Cons Expr Expr                 – (cons a d)
  | Car Expr                       – (car e)
  | Cdr Expr                       – (cdr d)
  | Nat                            – Nat
  | Zero                           – zero
  | Add1 Expr                      – (add1 e)
  | IndNat Expr Expr Expr Expr     – (ind-Nat tgt mot base step)
  | Equal Expr Expr Expr           – (= A from to)
  | Same                           – same
  | Replace Expr Expr Expr         – (replace tgt mot base)
  | Trivial                        – Trivial
  | Sole                           – sole
  | Absurd                         – Absurd
  | IndAbsurd Expr Expr            – (ind-Absurd tgt mot)
  | Atom                           – Atom
  | Tick String                    – 'a
  | U                              – 𝒰
  | The Expr Expr                  – (the t e)
  deriving (Eq, Show)
```

6.1 Program α-equivalence

While checking simple types could use the Eq instance for Ty to compare two types, α-equivalence is needed in a dependently-typed language because types are programs that can bind variables. Checking α-equivalence delegates immediately to a helper.

```
αEquiv :: Expr → Expr → Bool
αEquiv e₁ e₂ = αEquivHelper 0 [] e₁ [] e₂
```

The helper takes five arguments: the first is an integer, which is the number of variable bindings that have been crossed during the current traversal. The second argument and fourth arguments are representations of namespaces that map names to the depth at which they were bound. The third and fifth are expressions in which names might occur.

```
αEquivHelper :: Integer →
                [(Name, Integer)] → Expr →
                [(Name, Integer)] → Expr →
                Bool
```

Looking up the names in the namespaces will find Nothing when the name is free, or Just d when the name is bound at depth d. For the names to have a chance of being equivalent, either both must be bound or both must be free.

```
αEquivHelper i ns₁ (Var x) ns₂ (Var y) =
  case (lookup x ns₁, lookup y ns₂) of
    (Nothing, Nothing) → x == y
    (Just i,    Just j)    → i == j
    _                          → False
```

In the case of Pi, the argument type has the same number of bound variables as its surrounding context, while the return type has one more variable. This is represented by incrementing i as well as adding the variable for the recursive call.

```
αEquivHelper i ns₁ (Pi x a₁ r₁) ns₂ (Pi y a₂ r₂) =
  αEquivHelper i          ns₁           a₁ ns₂           a₂ &&
  αEquivHelper (i + 1) ((x, i) : ns₁) r₁ ((y, i) : ns₂) r₂
```

For Lambda, there is a bound variable in the body. The variables bound in each Lambda must be made equivalent by associating them with the same binding level, as in the return type of Pi.

```
αEquivHelper i ns₁ (Lambda x body₁) ns₂ (Lambda y body₂) =
  αEquivHelper (i + 1) ((x, i) : ns₁) body₁ ((y, i) : ns₂) body₂
```

Application binds no variables. Both the operator and operand positions must be α-equivalent.

```
αEquivHelper i ns₁ (App rator₁ rand₁) ns₂ (App rator₂ rand₂) =
  αEquivHelper i ns₁ rator₁ ns₂ rator₂ &&
  αEquivHelper i ns₁ rand₁  ns₂ rand₂
```

Sigma is checked for α-equivalence just like Pi, because its binding structure is identical.

```
αEquivHelper i ns₁ (Sigma x a₁ d₁) ns₂ (Sigma y a₂ d₂) =
  αEquivHelper i          ns₁           a₁ ns₂           a₂ &&
  αEquivHelper (i + 1) ((x, i) : ns₁) d₁ ((y, i) : ns₂) d₂
```

Pairs built with Cons are α-equivalent if their respective arguments are α-equivalent.

```
αEquivHelper i ns1 (Cons car1 cdr1) ns2 (Cons car2 cdr2) =
  αEquivHelper i ns1 car1 ns2 car2 &&
  αEquivHelper i ns1 cdr1 ns2 cdr2
```

Both Car and Cdr follow the usual rule that their α-equivalence is a consequence of their arguments' α-equivalence.

```
αEquivHelper i ns1 (Car pair1) ns2 (Car pair2) =
  αEquivHelper i ns1 pair1 ns2 pair2
αEquivHelper i ns1 (Cdr pair1) ns2 (Cdr pair2) =
  αEquivHelper i ns1 pair1 ns2 pair2
```

Because no more operators bind variables, the rest of the α-equivalence rules follow the same pattern of recurring on the subexpressions.

```
αEquivHelper _ _   Nat         _   Nat              = True
αEquivHelper _ _   Zero        _   Zero             = True
αEquivHelper i  ns1 (Add1 e1) ns2 (Add1 e2)         =
  αEquivHelper i ns1 e1       ns2 e2
αEquivHelper i  ns1 (IndNat tgt1 mot1 base1 step1)
                ns2 (IndNat tgt2 mot2 base2 step2) =
  αEquivHelper i ns1 tgt1   ns2 tgt2   &&
  αEquivHelper i ns1 mot1   ns2 mot2   &&
  αEquivHelper i ns1 base1  ns2 base2  &&
  αEquivHelper i ns1 step1  ns2 step2
αEquivHelper i  ns1 (Equal ty1 from1 to1)
                ns2 (Equal ty2 from2 to2)           =
  αEquivHelper i ns1 ty1    ns2 ty2    &&
  αEquivHelper i ns1 from1  ns2 from2  &&
  αEquivHelper i ns1 to1    ns2 to2
αEquivHelper _ _   Same        _   Same             = True
αEquivHelper i  ns1 (Replace tgt1 mot1 base1)
                ns2 (Replace tgt2 mot2 base2)       =
  αEquivHelper i ns1 tgt1   ns2 tgt2   &&
  αEquivHelper i ns1 mot1   ns2 mot2   &&
  αEquivHelper i ns1 base1  ns2 base2
αEquivHelper _ _   Trivial     _   Trivial          = True
αEquivHelper _ _   Sole        _   Sole             = True
αEquivHelper _ _   Absurd      _   Absurd           = True
αEquivHelper i  ns1 (IndAbsurd tgt1 mot1)
                ns2 (IndAbsurd tgt2 mot2)           =
  αEquivHelper i ns1 tgt1   ns2 tgt2   &&
  αEquivHelper i ns1 mot1   ns2 mot2
αEquivHelper _ _   Atom _ Atom                      = True
αEquivHelper _ _   U    _ U                         = True
```

There are two more special cases. In the case of concrete atoms, expressions built with tick marks are α-equivalent if the characters after the tick mark are the same.

```
αEquivHelper _ _ (Tick a1) _ (Tick a2) = a1 == a2
```

Additionally, the η-rule for Absurd states that every expression of type Absurd is the same as every other, just like the η-rule for Trivial. However, while η-expansion for Trivial is realized by reading every Trivial value back to the constructor Sole, there is no Absurd constructor. Tartlet implements the η-rule by reading back everything at type Absurd with a type annotation around it, marking its type, and then arranging for α-equivalence to ignore the contents of Absurd expressions.

```
αEquivHelper _ _    (The Absurd _) _    (The Absurd _) = True
αEquivHelper i ns1 (The t1 e1)      ns2 (The t2 e2)      =
  αEquivHelper i ns1 t1 ns2 t2 &&
  αEquivHelper i ns1 e1 ns2 e2
```

Finally, if no other rule matches, then the expressions are not α-equivalent.

```
αEquivHelper _ _ _ _ _ = False
```

6.2 Values and Normalization

6.2.1 The Values

Following the recipe for normalization by evaluation, we need to define value representations of each constructor and type constructor in the language.

```
data Value
  = VPi Ty Closure
  | VLambda Closure
  | VSigma Ty Closure
  | VPair Value Value
  | VNat
  | VZero
  | VAdd1 Value
  | VEq Ty Value Value
  | VSame
  | VTrivial
  | VSole
  | VAbsurd
  | VAtom
  | VTick String
  | VU
  | VNeutral Ty Neutral
  deriving Show
data Closure = Closure { closureEnv  :: Env
                       , closureName :: Name
                       , closureBody :: Expr
                       }
  deriving Show
```

During type checking, types are represented as values. For the sake of readability, a type synonym can help indicate when a value is expected to be a type.

```
type Ty = Value
```

Additionally, in order to write the evaluator, some of the usual machinery is necessary.

```
newtype Env = Env [(Name, Value)]
  deriving Show
extendEnv :: Env → Name → Value → Env
extendEnv (Env env) x v = Env ((x, v) : env)
evalVar :: Env → Name → Value
evalVar (Env [])           x = error ("Missing value for " ++ show x)
evalVar (Env ((y, v) : env)) x
  | x == y    = v
  | otherwise = evalVar (Env env) x
```

Just as in section 5, when neutral expressions are included as values, they have a type annotation. This is because the process of reading back is *type-directed*, which means that the way the syntax of a value or neutral expression is reconstructed depends on which type it is constructed at.

6.2.2 Neutral Expressions

Each eliminator in the language, including function application, must be able to recognize neutral targets and construct a representation of itself as a neutral expression.

```
data Neutral
  = NVar Name
  | NApp Neutral Normal
  | NCar Neutral
  | NCdr Neutral
  | NIndNat Neutral Normal Normal Normal
  | NReplace Neutral Normal Normal
  | NIndAbsurd Neutral Normal
  deriving Show
```

6.2.3 Normal Forms

Normal forms are expressions. Nonetheless, a value paired with its type is sufficient to recover a normal form, justifying the name Normal.

```
data Normal = Normal Ty Value
  deriving Show
```

6.3 Definitions and Dependent Types

The simply-typed language of section 5 could store definitions separately from the context and the environment, constructing each as needed for type checking or evaluation. In a dependently-typed language, however, type checking can invoke evaluation. This means that the context needs to distinguish between free variables that result from from binding forms such as λ, Π, and Σ, for which a value is not available during type checking, and free variables that result from from definitions, which do have values during type checking.

```
data CtxEntry = Def Ty Value | IsA Ty

newtype Ctx = Ctx [(Name, CtxEntry)]

initCtx :: Ctx
initCtx = Ctx []

ctxNames :: Ctx → [Name]
ctxNames (Ctx ctx) = map fst ctx

extendCtx :: Ctx → Name → Ty → Ctx
extendCtx (Ctx ctx) x t = Ctx ((x, IsA t) : ctx)

define :: Ctx → Name → Ty → Value → Ctx
define (Ctx ctx) x t v = Ctx ((x, Def t v) : ctx)
```

When looking up a variable during type checking, only its type is necessary.

```
lookupType :: Ctx → Name → Either Message Ty
lookupType (Ctx [])            x =
  failure ("Unbound variable: " ++ show x)
lookupType (Ctx ((y, e) : ctx)) x
   | x == y =
     case e of
       Def t _ → return t
       IsA t   → return t
   | otherwise =
     lookupType (Ctx ctx) x
```

The initial environment for an invocation of the evaluator, however, is based on the current type context, because each entry in the context represents a free variable that may occur in the expression being evaluated. Each entry in Γ is converted into a neutral variable value (NVar) in the initial environment.

```
mkEnv :: Ctx → Env
mkEnv (Ctx [])              = Env []
mkEnv (Ctx ((x, e) : ctx)) = Env ((x, v) : env)
  where
    Env env = mkEnv (Ctx ctx)
    v = case e of
          Def _ v → v
          IsA t     → VNeutral t (NVar x)
```

6.4 The Evaluator

The evaluator for Tartlet is much like the evaluator from section 5.2. The main difference is that types are now also subject to evaluation. Each binding form is realized as a closure, and each eliminator is implemented by delegating to an appropriate Haskell function.

```
eval :: Env → Expr → Value
eval env (Var x)             = evalVar env x
eval env (Pi x dom ran)      = VPi (eval env dom) (Closure env x ran)
eval env (Lambda x body)     = VLambda (Closure env x body)
eval env (App rator rand)    = doApply (eval env rator) (eval env rand)
eval env (Sigma x carType cdrType) =
  VSigma (eval env carType) (Closure env x cdrType)
eval env (Cons a d)          = VPair (eval env a) (eval env d)
eval env (Car e)             = doCar (eval env e)
eval env (Cdr e)             = doCdr (eval env e)
eval env Nat                 = VNat
eval env Zero                = VZero
eval env (Add1 e)            = VAdd1 (eval env e)
eval env (IndNat tgt mot base step) =
  doIndNat (eval env tgt) (eval env mot) (eval env base) (eval env step)
eval env (Equal ty from to) = VEq (eval env ty) (eval env from) (eval env to)
eval env Same                = VSame
eval env (Replace tgt mot base) =
  doReplace (eval env tgt) (eval env mot) (eval env base)
eval env Trivial             = VTrivial
eval env Sole                = VSole
eval env Absurd              = VAbsurd
eval env (IndAbsurd tgt mot) = doIndAbsurd (eval env tgt) (eval env mot)
eval env Atom                = VAtom
eval env (Tick x)            = VTick x
eval env U                   = VU
eval env (The ty e)          = eval env e
```

When typing or evaluation rules require substitution, the value that corresponds to the variable's scope will be a closure. Instead of actually implementing substitution, it's much easier and typically more efficient to evaluate the body of the closure in a suitably extended environment.

```
evalClosure :: Closure → Value → Value
evalClosure (Closure env x e) v = eval (extendEnv env x v) e
```

6.4.1 Eliminators

Each eliminator is realized by a Haskell function. This function checks whether the target of elimination is neutral, and if so, it produces a new neutral expression. Otherwise, it finds the resulting value.

```
doCar :: Value → Value
doCar (VPair v1 v2) = v1
doCar (VNeutral (VSigma aT dT) neu) =
  VNeutral aT (NCar neu)
doCdr :: Value → Value
doCdr (VPair v1 v2) = v2
doCdr v@(VNeutral (VSigma aT dT) neu) =
  VNeutral (evalClosure dT (doCar v)) (NCdr neu)
```

Note that the **cdr** of a neutral expression contains the **car** of that expression in its type annotation. Similarly, the type of a neutral application contains the value of the argument.

```
doApply :: Value → Value → Value
doApply (VLambda closure) arg =
  evalClosure closure arg
doApply (VNeutral (VPi dom ran) neu) arg =
  VNeutral (evalClosure ran arg) (NApp neu (Normal dom arg))
```

Because every Absurd is neutral, *doIndAbsurd* has only cases for neutral targets.

```
doIndAbsurd :: Value → Value → Value
doIndAbsurd (VNeutral VAbsurd neu) mot =
  VNeutral mot (NIndAbsurd neu (Normal VU mot))
```

When the equality proof in **replace** is same, both sides of the equality are the same, so the base case can be returned as-is.

```
doReplace :: Value → Value → Value → Value
doReplace VSame                                mot base =
  base
doReplace (VNeutral (VEq ty from to) neu) mot base =
  VNeutral (doApply mot to)
           (NReplace neu (Normal motT mot) (Normal baseT base))
  where
    motT  = VPi ty (Closure (Env []) (Name "x") U)
    baseT = doApply mot from
```

Because the type of the step for **ind-Nat** is somewhat complicated, it can be convenient to define a helper. Placing the motive in an environment allows the type to refer to it.

```
indNatStepType :: Value → Value
indNatStepType mot =
  eval (Env [(Name "mot", mot)])
    (Pi (Name "n-1") Nat
      (Pi (Name "almost") (App (Var (Name "mot"))
                               (Var (Name "n-1")))
        (App (Var (Name "mot"))
             (Add1 (Var (Name "n-1"))))))

doIndNat :: Value → Value → Value → Value → Value
doIndNat VZero                      mot base step =
  base
doIndNat (VAdd1 v)                  mot base step =
  doApply (doApply step v) (doIndNat v mot base step)
doIndNat tgt@(VNeutral VNat neu) mot base step =
  VNeutral (doApply mot tgt)
    (NIndNat neu
      (Normal (VPi VNat (Closure (Env []) (Name "k") U)) mot)
      (Normal (doApply mot VZero) base)
      (Normal (indNatStepType mot) step))
```

6.4.2 Reading Back

Just as in section 5, reading back from values into syntax is accomplished via two mutually-recursive procedures: *readBackNormal* and *readBackNeutral*. The former converts a Normal value into the syntax of its normal form, and the latter converts a neutral expression into its syntax. Because reading back will occur during type checking, instead of passing a list of used names, the type checking context is used to prevent capture.

```
readBackNormal :: Ctx → Normal → Expr
readBackNormal ctx (Normal t v) = readBackTyped ctx t v

readBackTyped :: Ctx → Ty → Value → Expr
readBackTyped ctx VNat VZero = Zero
readBackTyped ctx VNat (VAdd1 v) = Add1 (readBackTyped ctx VNat v)
```

In section 5.3, everything with a function type is immediately read back as having a Lambda on top. This implements the η-rule for functions. Similarly, in Tartlet, functions are η-expanded.

```
readBackTyped ctx (VPi dom ran) fun =
  Lambda x
    (readBackTyped
      (extendCtx ctx x dom)
      (evalClosure ran xVal)
      (doApply fun xVal))
  where
    x    = freshen (ctxNames ctx) (closureName ran)
    xVal = VNeutral dom (NVar x)
```

Pairs are also η-expanded. Every value with a pair type, whether it is neutral or not, is read back with Cons at the top.

```
readBackTyped ctx (VSigma aT dT) pair =
  Cons (readBackTyped ctx aT carVal)
    (readBackTyped ctx (evalClosure dT carVal) cdrVal)
  where
    carVal = doCar pair
    cdrVal = doCdr pair
```

The η-rule for Trivial states that all of its inhabitants are the same as sole. This is implemented by reading the all back as sole.

```
readBackTyped ctx VTrivial val = Sole
```

Finally the η-rule for Absurd states that all expressions with type Absurd are the same as all others. Because there are no values of type Absurd, it doesn't work to read it back as a value. Many implementations read Absurd expressions back as some special syntax that there is no parser for. In Tartlet, the expression is read back with a type annotation, and $\alpha Equiv$ is set up to ignore the contents of `the`-expressions where the type is Absurd.

```
readBackTyped ctx VAbsurd (VNeutral VAbsurd neu) =
  The Absurd (readBackNeutral ctx neu)
readBackTyped ctx (VEq _ _ _) VSame = Same
readBackTyped ctx VAtom (VTick x) = Tick x
readBackTyped ctx VU VNat = Nat
readBackTyped ctx VU VAtom = Atom
readBackTyped ctx VU VTrivial = Trivial
readBackTyped ctx VU VAbsurd = Absurd
readBackTyped ctx VU (VEq t from to) =
  Equal (readBackTyped ctx VU t)
        (readBackTyped ctx t from)
        (readBackTyped ctx t to)
readBackTyped ctx VU (VSigma aT dT) = Sigma x a d
  where
    x = freshen (ctxNames ctx) (closureName dT)
    a = readBackTyped ctx VU aT
    d = readBackTyped (extendCtx ctx x aT)
          VU
          (evalClosure dT (VNeutral aT (NVar x)))
readBackTyped ctx VU (VPi aT bT) = Pi x a b
  where
    x = freshen (ctxNames ctx) (closureName bT)
    a = readBackTyped ctx VU aT
    b = readBackTyped (extendCtx ctx x aT)
          VU
          (evalClosure bT (VNeutral aT (NVar x)))
readBackTyped ctx VU VU = U
readBackTyped ctx t (VNeutral t' neu) =
  readBackNeutral ctx neu
readBackTyped _ otherT otherE = error $ (show otherT) ++ show otherE

readBackNeutral :: Ctx → Neutral → Expr
readBackNeutral ctx (NVar x) = Var x
readBackNeutral ctx (NApp neu arg) =
  App (readBackNeutral ctx neu) (readBackNormal ctx arg)
readBackNeutral ctx (NCar neu) = Car (readBackNeutral ctx neu)
readBackNeutral ctx (NCdr neu) = Cdr (readBackNeutral ctx neu)
readBackNeutral ctx (NIndNat neu mot base step) =
  IndNat (readBackNeutral ctx neu)
         (readBackNormal ctx mot)
         (readBackNormal ctx base)
         (readBackNormal ctx step)
readBackNeutral ctx (NReplace neu mot base) =
  Replace (readBackNeutral ctx neu)
          (readBackNormal ctx mot)
          (readBackNormal ctx base)
```

The final case in *readBackNeutral* is another source of expressions with type Absurd. Thus, it emits a type annotation so that *αEquiv* will skip the expression itself. This is part of the implementation of the η-rule for Absurd.

```
readBackNeutral ctx (NIndAbsurd neu mot) =
  IndAbsurd
    (The Absurd (readBackNeutral ctx neu))
    (readBackNormal ctx mot)
```

6.5 Type Checking

When examining types, looking for specific type constructors, the type checker matches against their *values*. This ensures that the type checker never forgets to normalize before checking, which could lead to types that contain unrealized computation not being properly matched. For instance, the typing rules for **ind-Nat** might give rise to the type ((λ (k) Atom) zero) for the base. Attempting to use that expression as the type for 'sandwich would be incorrect without first reducing it. Using values, which cannot even represent redexes, removes the need to worry about normalization prior to inspection.

6.5.1 Type Synthesis

```
$synth$ :: Ctx → Expr → Either Message Ty
$synth\ ctx$ (Var $x$) =
  do $t \leftarrow lookupType\ ctx\ x$
     $return\ t$
```

$$\frac{\Gamma \vdash A \Leftarrow \mathcal{U} \qquad \Gamma, x : A \vdash D \Leftarrow \mathcal{U}}{\Gamma \vdash (\Pi\ ((x\ A))\ D) \Rightarrow \mathcal{U}}$$

```
$synth\ ctx$ (Pi $x\ a\ b$) =
  do $check\ ctx\ a$ VU
     $check\ (extendCtx\ ctx\ x\ (eval\ (mkEnv\ ctx)\ a))\ b$ VU
     $return$ VU
```

$$\frac{\Gamma \vdash rator \Rightarrow (\Pi\ ((x\ A))\ D) \qquad \Gamma \vdash rand \Leftarrow A}{\Gamma \vdash (rator\ rand) \Rightarrow D[rand/x]}$$

```
$synth\ ctx$ (App $rator\ rand$) =
  do $funTy \leftarrow synth\ ctx\ rator$
     $(a, b) \leftarrow isPi\ ctx\ funTy$
     $check\ ctx\ rand\ a$
     $return\ (evalClosure\ b\ (eval\ (mkEnv\ ctx)\ rand))$
```

$$\frac{\Gamma \vdash A \Leftarrow \mathcal{U} \qquad \Gamma, x : A \vdash D \Leftarrow \mathcal{U}}{\Gamma \vdash (\Sigma\ ((x\ A))\ D) \Rightarrow \mathcal{U}}$$

```
$synth\ ctx$ (Sigma $x\ a\ b$) =
  do $check\ ctx\ a$ VU
     $check\ (extendCtx\ ctx\ x\ (eval\ (mkEnv\ ctx)\ a))\ b$ VU
     $return$ VU
```

$$\frac{\Gamma \vdash e \Rightarrow (\Sigma\ ((x\ A))\ D)}{\Gamma \vdash (\textsf{car}\ e) \Rightarrow A} \qquad \frac{\Gamma \vdash e \Rightarrow (\Sigma\ ((x\ A))\ D)}{\Gamma \vdash (\textsf{cdr}\ e) \Rightarrow D[(\textsf{car}\ e)/x]}$$

```
$synth\ ctx$ (Car $e$) =
  do $t \leftarrow synth\ ctx\ e$
```

```
        $(aT, dT) \leftarrow isSigma\ ctx\ t$
        $return\ aT$
    $synth\ ctx\ (\mathsf{Cdr}\ e) =$
      do $t \leftarrow synth\ ctx\ e$
        $(aT, dT) \leftarrow isSigma\ ctx\ t$
        $return\ (evalClosure\ dT\ (doCar\ (eval\ (mkEnv\ ctx)\ e)))$
```

$$\frac{}{\Gamma \vdash \mathsf{Nat} \Rightarrow \mathcal{U}}$$

```
    $synth\ ctx\ \mathsf{Nat} = return\ \mathsf{VU}$
```

$$\frac{\Gamma \vdash tgt \Rightarrow \mathsf{Nat} \qquad \Gamma \vdash mot \Leftarrow (\Pi\ ((x\ \mathsf{Nat}))\ \mathcal{U}) \qquad \Gamma \vdash base \Leftarrow (mot\ \mathsf{zero}) \qquad \Gamma \vdash step \Leftarrow (\Pi\ ((k\ \mathsf{Nat}))\ (\Pi\ ((a\ (mot\ k)))\ (mot\ (\mathsf{add1}\ k))))}{\Gamma \vdash (\textbf{ind-Nat}\ tgt\ mot\ base\ step) \Rightarrow (mot\ tgt)}$$

```
    $synth\ ctx\ (\mathsf{IndNat}\ tgt\ mot\ base\ step) =$
      do $t \leftarrow synth\ ctx\ tgt$
        $isNat\ ctx\ t$
        let $tgtV\ \ = eval\ (mkEnv\ ctx)\ tgt$
            $motTy = eval\ (\mathsf{Env}\ [\,])\ (\mathsf{Pi}\ (\mathsf{Name}\ \texttt{"x"})\ \mathsf{Nat}\ \mathsf{U})$
        $check\ ctx\ mot\ motTy$
        let $motV = eval\ (mkEnv\ ctx)\ mot$
        $check\ ctx\ base\ (doApply\ motV\ \mathsf{VZero})$
        $check\ ctx\ step\ (indNatStepType\ motV)$
        $return\ (doApply\ motV\ tgtV)$
```

$$\frac{\Gamma \vdash A \Leftarrow \mathcal{U} \qquad \Gamma \vdash from \Leftarrow A \qquad \Gamma \vdash to \Leftarrow A}{\Gamma \vdash (=\ A\ from\ to) \Rightarrow \mathcal{U}}$$

```
    $synth\ ctx\ (\mathsf{Equal}\ ty\ from\ to) =$
      do $check\ ctx\ ty\ \mathsf{VU}$
        let $tyV = eval\ (mkEnv\ ctx)\ ty$
        $check\ ctx\ from\ tyV$
        $check\ ctx\ to\ tyV$
        $return\ \mathsf{VU}$
```

$$\frac{\Gamma \vdash tgt \Rightarrow (=\ A\ from\ to) \qquad \Gamma \vdash mot \Leftarrow (\Pi\ ((x\ A))\ \mathcal{U}) \qquad \Gamma \vdash base \Leftarrow (mot\ from)}{\Gamma \vdash (\textbf{replace}\ tgt\ mot\ base) \Rightarrow (mot\ to)}$$

```
    $synth\ ctx\ (\mathsf{Replace}\ tgt\ mot\ base) =$
      do $t \leftarrow synth\ ctx\ tgt$
        $(ty, from, to) \leftarrow isEqual\ ctx\ t$
        let $motTy = eval\ (\mathsf{Env}\ [(\mathsf{Name}\ \texttt{"ty"}, ty)])\ (\mathsf{Pi}\ (\mathsf{Name}\ \texttt{"x"})\ (\mathsf{Var}\ (\mathsf{Name}\ \texttt{"ty"}))\ \mathsf{U})$
        $check\ ctx\ mot\ motTy$
        let $motV = eval\ (mkEnv\ ctx)\ mot$
        $check\ ctx\ base\ (doApply\ motV\ from)$
        $return\ (doApply\ motV\ to)$
```

$$\frac{}{\Gamma \vdash \mathsf{Trivial} \Rightarrow \mathcal{U}}$$

```
$synth\ ctx$ Trivial = $return$ VU
```

$$\frac{}{\Gamma \vdash \mathsf{Absurd} \Rightarrow \mathcal{U}}$$

```
$synth\ ctx$ Absurd = $return$ VU
```

$$\frac{\Gamma \vdash tgt \Rightarrow \mathsf{Absurd} \qquad \Gamma \vdash mot \Leftarrow \mathcal{U}}{\Gamma \vdash (\textsf{ind-Absurd}\ tgt\ mot) \Rightarrow mot}$$

```
$synth\ ctx$ (IndAbsurd $tgt\ mot$) =
  do $t \leftarrow synth\ ctx\ tgt$
    $isAbsurd\ ctx\ t$
    $check\ ctx\ mot$ VU
    $return\ (eval\ (mkEnv\ ctx)\ mot)$
```

$$\frac{}{\Gamma \vdash \mathsf{Atom} \Rightarrow \mathcal{U}}$$

```
$synth\ ctx$ Atom = $return$ VU
```

$$\frac{}{\Gamma \vdash \mathcal{U} \Rightarrow \mathcal{U}}$$

```
$synth\ ctx$ U = $return$ VU
```

$$\frac{\Gamma \vdash t \Leftarrow \mathcal{U} \qquad \Gamma \vdash e \Leftarrow t}{\Gamma \vdash (\texttt{the}\ t\ e) \Rightarrow t}$$

```
$synth\ ctx$ (The $ty\ expr$) =
  do $check\ ctx\ ty$ VU
    let $tyV = eval\ (mkEnv\ ctx)\ ty$
    $check\ ctx\ expr\ tyV$
    $return\ tyV$

$synth\ ctx\ other$ =
  $failure$ ("Unable to synthesize a type for " ++ $show\ other$)
```

6.5.2 Type Checking

Unlike synthesis, type checking accepts a type as input. Because the checking judgment has no outputs, the computation will only either succeed or fail, with success indicated by the trivial type ().

```
$check :: \mathsf{Ctx} \to \mathsf{Expr} \to \mathsf{Ty} \to \mathsf{Either\ Message}\ ()$
```

$$\frac{\Gamma, x : A \vdash b \Leftarrow B}{\Gamma \vdash (\lambda\ (x)\ b) \Leftarrow (\Pi\ ((x\ A))\ B)}$$

```
$check\ ctx\ (\mathsf{Lambda}\ x\ body)\ t =$
  $\mathbf{do}\ (a, b) \leftarrow isPi\ ctx\ t$
      $\mathbf{let}\ xV = evalClosure\ b\ (\mathsf{VNeutral}\ a\ (\mathsf{NVar}\ x))$
      $check\ (extendCtx\ ctx\ x\ a)\ body\ xV$
```

$$\frac{\Gamma \vdash a \Leftarrow A \qquad \Gamma \vdash d \Leftarrow D[a/x]}{\Gamma \vdash (\mathsf{cons}\ a\ d) \Leftarrow (\Sigma\ ((x\ A))\ D)}$$

```
$check\ ctx\ (\mathsf{Cons}\ a\ d)\ t =$
  $\mathbf{do}\ (aT, dT) \leftarrow isSigma\ ctx\ t$
      $check\ ctx\ a\ aT$
      $\mathbf{let}\ aV = eval\ (mkEnv\ ctx)\ a$
      $check\ ctx\ d\ (evalClosure\ dT\ aV)$
```

$$\frac{}{\Gamma \vdash \mathsf{zero} \Leftarrow \mathsf{Nat}}$$

```
$check\ ctx\ \mathsf{Zero}\ t =$
  $isNat\ ctx\ t$
```

$$\frac{\Gamma \vdash n \Leftarrow \mathsf{Nat}}{\Gamma \vdash (\mathsf{add1}\ n) \Leftarrow \mathsf{Nat}}$$

```
$check\ ctx\ (\mathsf{Add1}\ n)\ t =$
  $\mathbf{do}\ isNat\ ctx\ t$
      $check\ ctx\ n\ \mathsf{VNat}$
```

The rule for same requires a bit of interpretation to become an algorithm. The repeated occurrence of e in the desired type is realized by checking both the equated values for sameness.

$$\frac{}{\Gamma \vdash \mathsf{same} \Leftarrow (=\ A\ e\ e)}$$

```
check ctx Same t =
  do (t, from, to) ← isEqual ctx t
     convert ctx t from to
```

$$\frac{}{\Gamma \vdash \mathsf{sole} \Leftarrow \mathsf{Trivial}}$$

```
check ctx Sole t =
  isTrivial ctx t
```

$$\frac{}{\Gamma \vdash {'a} \Leftarrow \mathsf{Atom}}$$

```
check ctx (Tick a) t =
  isAtom ctx t
```

$$\frac{\Gamma \vdash e \Rightarrow t' \qquad \Gamma \vdash t' \equiv t : \mathcal{U}}{\Gamma \vdash e \Leftarrow t}$$

```
check ctx other t =
  do t′ ← synth ctx other
    convert ctx VU t′ t
```

6.5.3 Sameness

Because every type is described by $\mathcal{U}$, there is no need for separate forms of judgment for sameness of types and sameness of expressions described by types. Because reading back an expression yields a normal form, conversion only needs to read back values and check them for α-equivalence.

```
convert :: Ctx → Ty → Value → Value → Either Message ()
convert ctx t v1 v2 =
  if αEquiv e1 e2
    then return ()
    else failure (show e1 ++ " is not the same type as " ++ show e2)
  where
    e1 = readBackTyped ctx t v1
    e2 = readBackTyped ctx t v2
```

6.5.4 Helpers

Some parts of the type checker rely on types being in a particular form. Instead of using a **case**-expression each time, it can be convenient to abstract out the logic of matching against a particular type value and returning its sub-parts.

```
unexpected :: Ctx → String → Value → Either Message a
unexpected ctx msg t = failure (msg ++ ": " ++ show e)
  where
    e = readBackTyped ctx VU t


isPi :: Ctx → Value → Either Message (Ty, Closure)
isPi _   (VPi a b) = return (a, b)
isPi ctx other     = unexpected ctx "Not a Pi type" other


isSigma :: Ctx → Value → Either Message (Ty, Closure)
isSigma _ (VSigma a b) = return (a, b)
isSigma ctx other      = unexpected ctx "Not a Sigma type" other


isNat :: Ctx → Value → Either Message ()
isNat _   VNat = return ()
isNat ctx other = unexpected ctx "Not Nat" other


isEqual :: Ctx → Value → Either Message (Ty, Value, Value)
isEqual _   (VEq ty from to) = return (ty, from, to)
isEqual ctx other            = unexpected ctx "Not an equality type" other


isAbsurd :: Ctx → Value → Either Message ()
isAbsurd _   VAbsurd = return ()
isAbsurd ctx other   = unexpected ctx "Not Absurd: " other


isTrivial :: Ctx → Value → Either Message ()
isTrivial _   VTrivial = return ()
isTrivial ctx other    = unexpected ctx "Not Trivial" other


isAtom :: Ctx → Value → Either Message ()
isAtom _   VAtom = return ()
isAtom ctx other = unexpected ctx "Not Atom" other
```

6.5.5 Definitions

The top-level context in Tartlet provides two syntactic constructs: definitions, which extend the context with a new defined name, and examples, which are expressions to be type checked and evaluated immediately. These two possibilities are represented in a datatype.

```
data Toplevel = Define Name Expr | Example Expr
```

Pie, from *The Little Typer*, allows separate type declarations for identifiers using the claim keyword. In Tartlet, on the other hand, definitions must have synthesizable types, and will likely have `the` at the top level.

The type Output describes the messages that can be displayed by Tartlet. There is only one: the type and normal form of an example. In a larger implementation, this type can be used for purposes such as tracking the binding structure of the document to facilitate operations like "go to definition" or tracking the types of each subexpression to allow useful tooltips.

```
data Output = ExampleOutput Expr
  deriving (Eq, Show)
```

Whether checking a file of code or implementing a REPL, *toplevel* can be used to track the state and output of Tartlet. The tuple that is returned contains output in its first projection, and the current global context in its second. The intention is that all the output is concatenated, while the global context is treated as a state, with the new context resulting from a definition being passed into the next operation.

```
toplevel :: Ctx → Toplevel → Either Message ([Output], Ctx)
toplevel ctx (Define x e) =
  case lookupType ctx x of
    Right _ → failure ("The name " ++ show x ++ " is already defined.")
    Left  _ →
      do t ← synth ctx e
         let v = eval (mkEnv ctx) e
         return ([], define ctx x t v)
toplevel ctx (Example e) =
  do  t ← synth ctx e
      let v  = eval (mkEnv ctx) e
          e′ = readBackTyped ctx t v
          t′ = readBackTyped ctx VU t
      return ([ExampleOutput (The t′ e′)], ctx)
```

6.6 Projects

This little subset of Pie can be extended with a number of features. Here's a few ideas to get you started:

1. Add a sum type, Either, with constructors left and right and an eliminator **ind-Either**.
2. Add lists and vectors (length-indexed lists).
3. Instead of using only the Either monad for the type checker, use a reader monad as well in order to pass the type checking context, and a monad with writer and state effects to implement *toplevel*.
4. Rewrite the type checker to perform *elaboration*, translating from a new datatype of user input expressions to the current datatype of core expressions. Then, do the following, which will not require modifying the evaluator:
    (a) Write a parser for input expressions, and accurately track their source locations to report errors.
    (b) Add non-dependent function types and non-dependent pair types to the user input expressions.
    (c) Add functions that take multiple arguments, but elaborate them to single-argument Curried functions.
5. Add holes and/or named metavariables to allow incremental construction of programs/proofs.
6. Replace $\mathcal{U}$ with an infinite number of universes and a cumulativity relation. To do this, type equality checks should be replaced by a subsumption check, where each type constructor has variance rules similar to other systems with subtyping.

6.7 Putting It Together

Checking dependent types requires answering two questions:

1. How to check sameness of expressions?
2. When to check sameness of expressions?

In this tutorial, the first question was answered using normalization by evaluation, and the second using bidirectional type checking. These are not the only potential answers.

Sameness can be checked incrementally, without needing to normalize a whole expression in the case when they are not equal, or techniques such as *hereditary substitution* (Watkins et al., 2002) can be used to *only* ever have normal forms of expressions. Another change that can be made to checking the sameness judgment is, instead of returning a trivial value on success, to return the conditions under which the judgment would be evident. For instance, if expressions can contain metavariables that stand for omitted parts of programs that the programmer expects to be automatically inferrable, then sameness checking can emit a collection of constraints over these metavariables to be solved by a separate pass. This is the approach taken in Agda (Norell, 2007).

Instead of a bidirectional approach, a synthesis-only approach can be adopted, in which every expression contains sufficient information to reconstruct its type, so checking has only the final catch-all case. This approach is often taken as a second step after an elaborating type checker, to ensure that there were no mistakes in elaboration. This is particularly useful when the elaborator contains many steps that produce core language output quite different from the high-level language, such as in Idris (Brady, 2013).

7 Further Reading

7.1 Tutorials on Implementing Type Theory

Löh et al. (2010) wrote a tutorial implementation of dependent types using bidirectional type checking in Haskell, and Lennart Augustsson wrote a response[3] that uses simpler language features.

Stephanie Weirich taught a course[4] at the Oregon Programming Languages Summer School on implementing dependent types that includes a bidirectional type checker. If you learn well from video lectures, then it is worth watching. She maintains the implementation on GitHub[5].

7.2 Bidirectional Type Checking

Pierce and Turner (1998) introduced the world to bidirectional type checking; however, they cite the idea as existing unpublished folklore amongst compiler writers. Following this paper, bidirectional typing is applied to many problems; see the introduction to Dunfield and Krishnaswami (2013) for a good survey as of 2013.

Additionally, Joshua Dunfield[6] and Frank Pfenning[7] have written good introductions to bidirectional type checking; and I wrote one[8] a few years ago as well. Stephanie Weirich's previously-mentioned sessions at the Oregon Programming Languages Summer School are another good introduction.

7.3 Normalization by Evaluation

Normalization by Evaluation was invented by Berger and Schwichtenberg (1991) to implement simply typed normalization efficiently by re-using Scheme func-

[3] `https://augustss.blogspot.com/2007/10/simpler-easier-in-recent-paper-simply.html`

[4] `https://www.cs.uoregon.edu/research/summerschool/summer13/curriculum.html`

[5] `https://github.com/sweirich/pi-forall`

[6] `https://people.mpi-sws.org/~joshua/bitype.pdf`

[7] `https://www.cs.cmu.edu/~fp/courses/15312-f04/handouts/15-bidirectional.pdf`

[8] `http://www.davidchristiansen.dk/tutorials/bidirectional.pdf`

tions as the semantics of functions in the simply typed λ-calculus. Danvy (1996) extended their scheme to features such as recursion, sums, and effects, calling the resulting generalized system "Type-Directed Partial Evaluation."

Thorsten Altenkirch and McBride (2005) wrote an implementation of type theory using type-directed reading back of values to syntax, similarly to here. Their system is written with efficiency in mind, taking careful advantage of Haskell's laziness to maintain the syntax and semantics together, computing only as much as absolutely necessary.

Andreas Abel's habilitation thesis (Abel, 2013) is a fantastic overview of a long line of research on normalization by evaluation for variants of dependent type theory, using consistent notation and explanations. The thesis's extensive bibliography cites a number of additional papers by Abel and his collaborators that I have not repeated here.

7.4 Other Approaches

Typed normalization by evaluation is far from the only way to implement conversion checking for dependent types. Indeed, normalization by evaluation has a number of characteristics that make it only suitable for certain theories: it η-expands expressions as many times as possible, but η-expansion is not valid for some theories (including many versions of Coq's Calculus of Constructions) and which furthermore can consume a lot of memory if retained; also, it fully normalizes terms when it may have been possible to determine they were identical by an immediate α-equivalence check. Because NbE with higher-order closures re-uses the implementation language's functions, achieving laziness in a strict implementation language requires additional work.

Coquand (1996) describes a bidirectional type checker (two years prior to Pierce and Turner (1998)) that uses a form of abstract machine to implement conversion checking. The machine maintains an environment and a cursor into an expression and incrementally reduces the expression under the cursor until it either fails or has checked the entire expression. This incremental approach supports the exploitation of partial α-equivalence, but another solution is necessary if η-equivalence is desired.

Grégoire and Leroy (2002) implemented a compiler from Coq to a version of OCaml's ZAM machine, resulting in massive improvements to Coq's efficiency. Much of Coq's applicability to large problems is a direct result of their work.

Dirk Kleeblatt's PhD thesis (Kleeblatt, 2011) describes an implementation of type theory in which expressions are compiled directly to machine code, using a strongly normalizing variant of the STG machine used in GHC. Because type theory is typically presented with lazy runtime semantics, this is an efficient realization.

Acknowledgments

I would like to thank Dan Friedman for constructive comments on drafts of this tutorial, Sam Tobin-Hochstadt for essential technical consultation while developing the literate programming library `brush` that was used for the original Racket version, Rutvik Patel and Siyuan Chen for catching a number of bugs, and especially Andreas Abel for his clear descriptions of normalization by evaluation. Bugs, poor explanations and mistakes are, of course, my own.

References

Abel, Andreas (2013). *Normalization by Evaluation: Dependent Types and Impredicativity.* Habilitation Thesis, Institut für Informatik, Ludwig-Maximilians-Universität München.

Berger, Ulrich and Helmut Schwichtenberg (1991). "An inverse to the evaluation functional for typed λ-calculus". In: *Logic in Computer Science.*

Brady, Edwin (2013). "Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation". In: *Journal of Functional Programming* 23.5, pp. 552–593.

Coquand, Thierry (1996). "An algorithm for type-checking dependent types". In: *Science of Computer Programming*, pp. 167–177.

Danvy, Olivier (1996). "Type-Directed Partial Evaluation". In: *Principles of Programming Languages.* POPL.

Dunfield, Joshua and Neelakantan Krishnaswami (2013). "Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism". In: *International Conference on Functional Programming.* ICFP.

Friedman, Daniel P. and David Thrane Christiansen (2018). *The Little Typer.* MIT Press.

Grégoire, Benjamin and Xavier Leroy (2002). "A compiled implementation of strong reduction". In: *International Conference on Functional Programming.* ICFP.

Kleeblatt, Dirk (2011). "On a Strongly Normalizing STG Machine With an Application to Dependent Type Checking". PhD thesis. Technical University of Berlin.

Landin, Peter (1964). "The mechanical evaluation of expressions". In: *Computer Journal* 6, pp. 308–320.

Löh, Andres, Conor McBride, and Wouter Swierstra (2010). "A tutorial implementation of a dependently typed lambda calculus". In: *Fundamenta Informaticae* 102.2, pp. 177–207.

Norell, Ulf (2007). "Towards a practical programming language based on dependent type theory". PhD thesis. Chalmers University of Technology and Göteborg University.

Pierce, Benjamin C. and David N. Turner (1998). "Local Type Inference". In: *Principles of Programming Languages.* POPL.

Thorsten Altenkirch, James Chapman andn and Conor McBride (2005). “Epigram reloaded: a standalone typechecker for ETT”. In: *Post-Proceedings of Trends in Functional Programming*.

Watkins, Kevin, Iliano Cervesato, Frank Pfenning, and David Walker (2002). *A concurrent logical framework I: Judgments and properties.* Tech. rep. revised May 2003. Carnegie Mellon University, School of Computer Science.