Checking Dependent Types with Normalization by Evaluation: A Tutorial

by David Thrane Christiansen

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 outputs are the same. Normalization by evaluation is one way of performing this sameness check, while bidirectional type checking 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 from *The Little Typer* [Friedman18].

These notes are written assuming that you know the following:

- The basics of Racket, including pattern matching and how to define structures and functions
- 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

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.

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 `struct`s for the values, and use procedures 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 Values and Runtime Environments

```
(struct CLOS (env var body)                                    struct
    #:transparent)
  env : Env
  var : symbol?
  body : any/c
```

A *closure* packages an expression that has not yet been evaluated with the run-time 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 have the `var` field.

```
(struct CLOS (env var body) #:transparent)
```

The `#:transparent` modifier causes the resulting structures to use structural equality comparisons and to be printable. The former is useful for the test suite for these notes, while the latter is convenient for debugging. Because the documentation for structs such as `CLOS` includes the complete source code for the declaration, only the documentation is shown later in these notes.

Runtime environments provide the values for each variable. By convention, runtime environments are referred to with the Greek letter "ρ." In this implementation, environments are association lists, containing pairs of variable names and values. Earlier values override later values in the list.

> ρ can be written "rho," and is pronounced like "row."

The Racket function `assv` can be used to look up a name in an environment, returning either the first pair that matches or `#f`. For instance:

```
> (assv 'x (list (cons 'y "peaches") (cons 'x "apples")))
'(x . "apples")
> (assv 'x (list (cons 'y "peaches") (cons 'z "apples")))
#f
```

```
(extend ρ x v) → environment?                                  procedure
  ρ : environment?
  x : symbol?
  v : value?
```

Because `assv` returns the *first* matching entry, extending an environment with a new variable only needs to add to the front.

```
(define (extend ρ x v)
  (cons (cons x v) ρ))
```

1.2 The Evaluator

The evaluator consists of two procedures: `val` evaluates an expression in a run-time environment that provides values for its free variables, and `do-ap` is responsible for applying the value of a function to the value of its argument. The procedures `val` and `do-ap` are often called `eval` and `apply`; here, we use different names to prevent conflict with Racket's built-in operators.

> Racket's `apply` is slightly different, in that it accepts a *list* of arguments. Here, however, there is always precisely one argument, so the distinction is moot.

(**val** *ρ e*) → value? procedure
ρ : environment?
e : expression?

Find the value of the expression *e*, if such a value exists.

```
(define (val ρ e)
  (match e
    [`(λ (,x) ,b)
     (CLOS ρ x b)]
    [x #:when (symbol? x)
     (let ((xv (assv x ρ)))
       (if xv
           (cdr xv)
           (error 'val "Unknown variable ~a" x)))]
    [`(,rator ,rand)
     (do-ap (val ρ rator) (val ρ rand))]))
```

(**do-ap** *clos arg*) → value? procedure
clos : value?
arg : value?

Apply a function value to an argument.

```
(define (do-ap clos arg)
  (match clos
    [(CLOS ρ x b)
     (val (extend ρ x arg) b)]))
```

The names *rator* and *rand* are short for "operator" and "operand." These names go back to Peter Landin [Landin64].

Examples:

```
> (val '() '(λ (x) (λ (y) y)))
(CLOS  '()     'x '(λ (y) y))
> (val '() '((λ (x) x) (λ (x) x)))
(CLOS  '()     'x 'x)
> (val '() 'x)
val: Unknown variable x
```

1.3 Adding 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 `"ρ"`, 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.

(**run-program** *exprs*) → void? procedure
exprs : (listof expression?)

```
(define (run-program ρ exprs)
  (match exprs
    ['() (void)]
    [(cons `(define ,x ,e) rest)
     (let ([v (val ρ e)])
       (run-program (extend ρ x v) rest))]
    [(cons e rest)
     (displayln (val ρ e))
     (run-program ρ rest)]))
```

```
> (run-program '() '((define id (λ (x) x))
                     (id (λ (y) (λ (z) (z y))))))
#(struct:CLOS ((id . #(struct:CLOS () x x))) y (λ (z) (z y)))
> (run-program '() '((define z
                       (λ (f)
                         (λ (x)
                           x)))
                     (define s
                       (λ (n)
```

```
                        (λ (f)
                          (λ (x)
                            (f ((n f) x))))))
                    (s (s z))))
#(struct:CLOS ((n . #(struct:CLOS ((n . #(struct:CLOS () f (λ (x) x))) (z . #(struct:CLOS () f (λ (x) x)))) f (λ (x) (f ((n f) x))))) (z . #(struct:CLOS
```

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. Normalizing Untyped λ-Calculus demonstrates how to recover readable code from these closure values.

2 Generating 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 asterisks until it is not included in the used names.

(**add-*** *x*) → symbol? procedure
  x : symbol?

Add an asterisk to a symbol.

```
(define (add-* x)
  (string->symbol
   (string-append (symbol->string x)
                  "*")))
```

(**freshen** *used x*) → symbol? procedure
  used : (listof symbol?)
  x : symbol?

Construct a name that does not occur in *used*.

```
(define (freshen used x)
  (if (memv x used)
      (freshen used (add-* x))
      x))
```

Examples:

```
> (freshen '() 'x)
'x
> (freshen '(x x*) 'x)
'x**
> (freshen '(x y z) 'y)
'y*
```

3 Normalizing Untyped λ-Calculus

3.1 Normal Forms

```
‹expr› ::= ‹id›
         | ( λ ( ‹id› ) ‹expr› )
         | ( ‹expr› ‹expr› )
```

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.

3.2 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 Evaluating Untyped λ-Calculus 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.

In other words, the grammar of normal forms that is to be captured by the values is as follows:

```
‹norm› ::= ‹neu›
         | ( λ ( ‹id› ) ‹norm› )
‹neu›  ::= ‹id›
         | ( ‹neu› ‹norm› )
```

In addition to `CLOS`, which represents λ-expressions, we need values to represent neutral expressions.

```
(struct N-var (name))                                    struct
  name : symbol?
```

`N-var` represents a neutral variable, for which the name is saved.

```
(struct N-ap (rator rand))                               struct
  rator : neutral?
  rand : value?
```

`N-ap` represents a neutral application, in which the rator cannot yet be evaluated.

The evaluator itself should be extended to handle neutral expressions. Perhaps surprisingly, `val` remains unchanged, while `do-ap` needs an extra case for neutral applications. Here, `val` is defined again so that it can refer to the new `do-ap`.

```
(val ρ e) → value?                                       procedure
  ρ : environment?
  e : expression?
```

Evaluate an expression.

```
(define (val ρ e)
  (match e
    [`(λ (,x) ,b)
     (CLOS ρ x b)]
    [x #:when (symbol? x)
     (let ((xv (assv x ρ)))
       (if xv
           (cdr xv)
           (error 'val "Unknown variable ~v" x)))]
    [`(,rator ,rand)
     (do-ap (val ρ rator) (val ρ rand))]))
```

(**do-ap** *fun arg*) → value? procedure
  fun : value?
  arg : value?

Apply a function, potentially attaching more arguments to a neutral expression.

```
(define (do-ap fun arg)
  (match fun
    [(CLOS ρ x b)
     (val (extend ρ x arg) b)]
    ; If the argument is neutral, construct a bigger neutral expression.
    [neutral-fun
     (N-ap fun 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* value into syntax.

(**read-back** *used-names v*) → expression? procedure
  used-names : (listof symbol?)
  v : value?

```
(define (read-back used-names v)
  (match v
    [(CLOS ρ x body)
     (let* ((y (freshen used-names x))
            (neutral-y (N-var y)))
       `(λ (,y)
          ,(read-back (cons y used-names)
                      (val (extend ρ x neutral-y) body))))]
    [(N-var x) x]
    [(N-ap rator rand)
     `(,(read-back used-names rator) ,(read-back used-names rand))]))
```

Example:

```
> (read-back '() (val '() '((λ (x) (λ (y) (x y))) (λ (x) x))))
'(λ (y) y)
```

The combination of evaluation and reading back leads to normalization.

(**norm** *e*) → expression? procedure
  e : expression?

Normalize an expression.

```
(define (norm ρ e)
  (read-back '() (val ρ e)))
```

With normalization, a program can be run in a more friendly manner. Each expression's normal form can be displayed instead of its value.

(**run-program** *ρ exprs*) → void? procedure
  ρ : (listof (pair symbol? value?))
  exprs : (listof expression?)

```
(define (run-program ρ exprs)
  (match exprs
    [(list) (void)]
    [(list `(define ,x ,e) rest ...)
     (let ([v (val ρ e)])
       (run-program (extend ρ x v) rest))]
    [(list e rest ...)
     (displayln (norm ρ e))
     (run-program ρ rest)]))
```

3.3 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 as a function that takes another function and some starting value, and applies the function *n* times to the starting value.

(**with-numerals** *e*) → program? procedure
  e : expression?

Place the expression `e` into a context where the basic Church numerals are defined.

```
(define (with-numerals e)
  `((define church-zero
      (λ (f)
        (λ (x)
          x)))
    (define church-add1
      (λ (n-1)
        (λ (f)
          (λ (x)
            (f ((n-1 f) x))))))
    ,e))
```

(**to-church** *n*) → expression? procedure

n : exact-nonnegative-integer?

Convert a natural number into its Church encoding.

```
(define (to-church n)
  (cond [(zero? n) 'church-zero]
        [(positive? n)
         (let ([church-of-n-1 (to-church (sub1 n))])
           `(church-add1 ,church-of-n-1))]))
```

Examples:

```
> (run-program '() (with-numerals (to-church 0)))
(λ (f) (λ (x) x))
> (run-program '() (with-numerals (to-church 1)))
(λ (f) (λ (x) (f x)))
> (run-program '() (with-numerals (to-church 4)))
(λ (f) (λ (x) (f (f (f (f x))))))
```

---

church-add : expression? value

Addition of Church numerals.

```
(define church-add
  `(λ (j)
     (λ (k)
       (λ (f)
         (λ (x)
           ((j f) ((k f) x)))))))
```

Example:

```
> (run-program '() (with-numerals `((,church-add ,(to-church 2)) ,(to-church 2))))
(λ (f) (λ (x) (f (f (f (f x))))))
```

4 Error handling

Previously, programs that contained errors such as unbound variables or malformed syntax would simply crash the evaluator. Mismatched types, however, are not *errors*—they are instead useful feedback that can be used while constructing a program. This calls for a somewhat more convenient presentation of the fact that type checking can fail.

(struct **go** (result) struct

#:transparent)

result : any/c

(struct **stop** (expr message) struct

#:transparent)

expr : expression?

message : string?

`go` is a wrapper around a successful result, indicating that its contents have passed some test. `stop` represents that something did not pass, and it contains the expression where things went wrong as well as an explanation of why they went wrong. To make the implementation more user-friendly, consider replacing the expression argument to `stop` with something like a source location and the string with some kind of structured representation of type mismatches.

The next definition, `go-on`, requires an additional import. Add

```
(require (for-syntax syntax/parse))
```

to the module in order to import the library that defines `syntax-parse`. If you are working in a language other than `racket`, such as `racket/base`, then add

```
(require (for-syntax racket/base syntax/parse))
```

instead.

(**go-on** ([*pattern expr*] ...) *result*) syntax

Explicitly pattern-matching on the contents of `go` can be tedious to read and write, so the `go-on` macro allows a syntax similar to `let*`, but where the whole expression is `stop` if any subcomputations yield `stop`. This is similar to Haskell's `do`-notation for the `Either` monad.

```
(define-syntax (go-on stx)
  (syntax-parse stx
    [(go-on () result)
     (syntax/loc stx
       result)]
    [(go-on ([pat0 e0] [pat e] ...) result)
     (syntax/loc stx
       (match e0
         [(go pat0) (go-on ([pat e] ...) result)]
         [(go v) (error 'go-on "Pattern did not match value ~v" v)]
         [(stop expr msg) (stop expr msg)]))]))
```

Examples:

```
> (define (bigger-than-two n)
    (if (> n 2)
        (go n)
        (stop n "Not greater than two")))
> (go-on ([x (bigger-than-two 4)]
          [y (bigger-than-two 5)])
    (go (+ x y)))
(go  9)
> (go-on ([x (bigger-than-two 1)]
          [y (bigger-than-two 5)])
    (go (+ x y)))
(stop  1 "Not greater than two")
> (go-on ([x (bigger-than-two 4)]
          [y (bigger-than-two -3)])
    (go (+ x y)))
(stop  -3 "Not greater than two")
```

5 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 without type annotations on functions:

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

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

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

If the rules are read as the source code of a program for checking a program against a type, the rule for $\lambda x.\, e$ is not immediately translatable, because there is nowhere to get the type t_1 from.

Bidirectional type checking is a 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.

5.1 Types

The Racket representation of types puts the operators in prefix position, so the function type has the arrow at the beginning.

```
‹type› ::= Nat
         | ( → ‹type› ‹type› )
```

(**type=?** *t1 t2*) → boolean? procedure

t1 : any/c

t2 : any/c

Two types are equal when they are valid types that match structurally.

```
(define (type=? t1 t2)
  (match* (t1 t2)
    [('Nat 'Nat) #t]
    [(`(→ ,A1 ,B1) `(→ ,A2 ,B2))
     (and (type=? A1 A2) (type=? B1 B2))]
    [(_ _) #f]))
```

(**type?** *t*) → boolean? procedure

t : any/c

If something is equal to itself as a type, then it is a type. Thus, there is no need to write a second recursive function to recognize the valid types.

```
(define (type? t)
  (type=? t t))
```

Examples:

```
> (type? 'Nat)
#t
> (type? '(Nat))
#f
> (type? '(→ Nat Nat))
#t
> (type=? 'Nat 'Nat)
#t
> (type=? '(→ Nat (→ Nat Nat))
          '(→ Nat (→ Nat Nat)))
#t
> (type=? '(→ (→ Nat Nat) Nat)
          '(→ Nat (→ Nat Nat)))
#f
```

5.2 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 `→` 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.

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.

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.

$$\frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e \in A \Rightarrow A} \qquad \frac{\Gamma \vdash e \Rightarrow B \quad B = A}{\Gamma \vdash e \Leftarrow A}$$

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.

$$\frac{}{\Gamma_1, x : A, \Gamma_2 \vdash x \Rightarrow A} \qquad \frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e \Leftarrow A \to B} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \to B \quad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}$$

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.

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

Expressed as a program, these rules become the procedures `synth` and `check`.

```
(synth Γ e) → (perhaps/c type?)                                  procedure
  Γ : context?
  e : expression?
```

```
(define (synth Γ e)
  (match e
    ; Type annotations
    [`(the ,t ,e2)
     (if (not (type? t))
         (stop e (format "Invalid type ~a" t))
         (go-on ([_ (check Γ e2 t)])
           (go t)))]
    ; Recursion on Nat
    [`(rec ,type ,target ,base ,step)
     (go-on ([target-t (synth Γ target)]
             [_ (if (type=? target-t 'Nat)
                    (go 'ok)
                    (stop target (format "Expected Nat, got ~v"
                                         target-t)))]
             [_ (check Γ base type)]
             [_ (check Γ step `(→ Nat (→ ,type ,type)))])
       (go type))]
    [x #:when (and (symbol? x)
                   (not (memv x '(the rec λ zero add1))))
     (match (assv x Γ)
       [#f (stop x "Variable not found")]
       [(cons _ t) (go t)])]
```

```
    [`(,rator ,rand)
     (go-on ([rator-t (synth Γ rator)])
       (match rator-t
         [`(→ ,A ,B)
          (go-on ([_ (check Γ rand A)])
            (go B))]
         [else (stop rator (format "Not a function type: ~v"
                                   rator-t))]))]))
```

(**check** *Γ e t*) → (perhaps/c 'ok) procedure

Γ : context?
e : expression?
t : type?

```
(define (check Γ e t)
  (match e
    ['zero
     (if (type=? t 'Nat)
         (go 'ok)
         (stop e (format "Tried to use ~v for zero" t)))]
    [`(add1 ,n)
     (if (type=? t 'Nat)
         (go-on ([_ (check Γ n 'Nat)])
           (go 'ok))
         (stop e (format "Tried to use ~v for add1" t)))]
    [`(λ (,x) ,b)
     (match t
       [`(→ ,A ,B)
        (go-on ([_ (check (extend Γ x A) b B)])
          (go 'ok))]
       [non-arrow
        (stop e (format "Instead of → type, got ~a" non-arrow))])]
    [other
     (go-on ([t2 (synth Γ e)])
       (if (type=? t t2)
           (go 'ok)
           (stop e
                 (format "Synthesized type ~v where type ~v was expected"
                         t2
                         t))))]))
```

Examples:

```
> (synth (list (cons 'x 'Nat)) 'x)
(go 'Nat)
> (check '() 'zero 'Nat)
(go 'ok)
> (check '() '(add1 zero) 'Nat)
(go 'ok)
> (check '() '(λ (x) x) '(→ Nat Nat))
(go 'ok)
> (check '()
         '(λ (j)
            (λ (k)
              (rec Nat j k (λ (n-1)
                             (λ (sum)
                               (add1 sum))))))
         '(→ Nat (→ Nat Nat)))
(go 'ok)
```

5.3 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 `the` can be used to annotate checked forms such as functions.

(**check-program** *Γ prog*) → (perhaps/c context?) procedure

Γ : context?
prog : (listof (or/c expression? (list/c 'define symbol? expression?)))

Type check a series of definitions and/or expressions.

```
(define (check-program Γ prog)
  (match prog
    ['()
```

```
      (go Γ)]
     [(cons `(define ,x ,e) rest)
      (go-on ([t (synth Γ e)])
        (check-program (extend Γ x t) rest))]
     [(cons e rest)
      (go-on ([t (synth Γ e)])
        (begin
          (printf "~a has type ~a\n" e t)
          (check-program Γ rest)))]))
```

Example:

```
> (check-program '()
                 '((define three
                     (the Nat
                          (add1 (add1 (add1 zero)))))
                   (define +
                     (the (→ Nat (→ Nat Nat))
                          (λ (n)
                            (λ (k)
                              (rec Nat n
                                k
                                (λ (pred)
                                  (λ (almost-sum)
                                    (add1 almost-sum))))))))
                   (+ three)
                   ((+ three) three)))
(+ three) has type (→ Nat Nat)
((+ three) three) has type Nat
(go  '((+ → Nat (→ Nat Nat)) (three . Nat)))
```

The last line of that output maps `+` to the type `(→ Nat (→ Nat Nat))`, because in Racket, both pairs and lists are formed with `cons`, and the printer has no way to tell whether a particular pair is intended as part of a list or as a pair whose `cdr` is a list.

Example:

```
> (cons '+ '(→ Nat (→ Nat Nat)))
'(+ → Nat (→ Nat Nat))
```

6 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 $\Gamma \vdash e_1 = e_2 : A$ means that the type A considers the expressions e_1 and e_2 to be equivalent, and presumes that both e_1 and e_2 have type A. In this section, we use the following rules:

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

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

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

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

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

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.

6.1 Values for Typed NbE

```
(struct ZERO ()                                                    struct
    #:transparent)
```

```
(struct ADD1 (pred)                                    struct
    #:transparent)
  pred : value?
(struct NEU (type neu)                                 struct
    #:transparent)
  type : type?
  neu : neutral?
```

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 `NEU` is used to track these types. This embedding of expressions into values is frequently referred to as *reflection*, and written with an up arrow ↑.

```
(struct N-var (name)                                   struct
    #:transparent)
  name : symbol?
(struct N-ap (rator rand))                             struct
  rator : neutral?
  rand : norm?
(struct N-rec (type target base step)                  struct
    #:transparent)
  type : type?
  target : neutral?
  base : norm?
  step : norm?
```

The neutral expressions are constructed similarly to those in untyped normalization by evaluation. The embedded values, however, now contain type annotations, provided with `THE`.

```
(struct THE (type value)                               struct
    #:transparent)
  type : type?
  value : value?
(norm? v) → boolean?                                   procedure
  v : any/c
```

Normalized expressions waiting to be read back are saved together with their types. This is commonly referred to as *reification*, making a value into an expression, and it is frequently written with a down arrow ↓.

```
(define (norm? v)
  (THE? v))
```

6.2 The Evaluator

Just as in Normalizing Untyped λ-Calculus, normalization consists of evaluation followed by reading back. Here, introduction and elimination rules for natural numbers are included.

```
(val ρ e) → value?                                     procedure
  ρ : environment?
  e : expression?
```

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.

```
(define (val ρ e)
  (match e
    [`(the ,type ,expr)
     (val ρ expr)]
    ['zero (ZERO)]
    [`(add1 ,n) (ADD1 (val ρ n))]
    [x #:when (and (symbol? x)
                   (not (memv x '(the zero add1 λ rec))))
     (cdr (assv x ρ))]
    [`(λ (,x) ,b)
     (CLOS ρ x b)]
    [`(rec ,type ,target ,base ,step)
     (do-rec type (val ρ target) (val ρ base) (val ρ step))]
    [`(,rator ,rand)
     (do-ap (val ρ rator) (val ρ rand))]))
```

```
                                                       procedure
```

```
(do-ap fun arg) → value?
  fun : value?
  arg : value?
(do-rec type target base step) → value?                    procedure
  type : type?
  target : value?
  base : value?
  step : value?
```

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.

```
(define (do-ap fun arg)
  (match fun
    [(CLOS ρ x e)
     (val (extend ρ x arg) e)]
    [(NEU `(→ ,A ,B) ne)
     (NEU B (N-ap ne (THE A arg)))]))
```

```
(define (do-rec type target base step)
  (match target
    [(ZERO) base]
    [(ADD1 n)
     (do-ap (do-ap step n)
            (do-rec type n base step))]
    [(NEU 'Nat ne)
     (NEU type
          (N-rec type
                 ne
                 (THE type base)
                 (THE `(→ Nat (→ ,type ,type)) step)))]))
```

6.3 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.

```
(read-back used-names type value) → expression?            procedure
  used-names : (listof symbol?)
  type : type?
  value : value?
(read-back-neutral used-names ne) → expression?            procedure
  used-names : (listof symbol?)
  ne : neutral?
```

```
(define (read-back used-names type value)
  (match type
    ['Nat
     (match value
       [(ZERO) 'zero]
       [(ADD1 n) `(add1 ,(read-back used-names 'Nat n))]
       [(NEU _ ne)
        (read-back-neutral used-names ne)])]
    [`(→ ,A ,B)
     (let ([x (freshen used-names 'x)])
       `(λ (,x)
          ,(read-back (cons x used-names)
                      B
                      (do-ap value (NEU A (N-var x))))))]))
(define (read-back-neutral used-names ne)
  (match ne
    [(N-var x) x]
    [(N-ap fun (THE arg-type arg))
     `(,(read-back-neutral used-names fun)
       ,(read-back used-names arg-type arg))]
    [(N-rec type target (THE base-type base) (THE step-type step))
     `(rec ,type
        ,(read-back-neutral used-names target)
        ,(read-back used-names base-type base)
        ,(read-back used-names step-type step))]))
```

6.4 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.

```
(struct def (type value)                                    struct
    #:transparent)
  type : type?
  value : value?
```

For defined names, both the type and the value must be stored in order to both type-check and evaluate the remaining expressions. A `def` consists of a type and a value of that type.

Similarly to Γ and ρ, definitions are saved in an association list, which will be referred to as Δ. The definitions can be converted to both a type-checking context and a run-time environment, as it contains both types and definitions.

> The Greek letter Δ is pronounced "delta." Here, it stands for definitions.

```
(defs->ctx Δ) → context?                                    procedure
  Δ : definitions?
(defs->env Δ) → environment?                                procedure
  Δ : definitions?
```

```
(define (defs->ctx Δ)
  (match Δ
    ['() '()]
    [(cons (cons x (def type _)) rest)
     (extend (defs->ctx rest) x type)]))
(define (defs->env Δ)
  (match Δ
    ['() '()]
    [(cons (cons x (def _ value)) rest)
     (extend (defs->env rest) x value)]))
```

```
(run-program Δ prog) → (perhaps/c definitions?)             procedure
  Δ : definitions?
  prog : (listof (or/c (list 'define symbol? expression?)
                       expression?))
```

Running a typed program with definitions is similar to its untyped counterpart. Definitions are type checked, evaluated, and saved; expressions are type checked and evaluated, and their normal forms are printed.

```
(define (run-program Δ prog)
  (match prog
    ['() (go Δ)]
    [(cons `(define ,x ,e) rest)
     (go-on ([type (synth (defs->ctx Δ) e)])
       (run-program (extend Δ x (def type (val (defs->env Δ) e)))
                    rest))]
    [(cons e rest)
     (let ([Γ (defs->ctx Δ)]
           [ρ (defs->env Δ)])
       (go-on ([type (synth Γ e)])
         (let ([v (val ρ e)])
           (begin
             (printf "(the ~a\n  ~a)\n"
                     type
                     (read-back (map car Γ) type v))
             (run-program Δ rest))))))]))
```

Example:

```
> (run-program '() '((define +
                       (the (→ Nat
                               (→ Nat
                                  Nat))
                            (λ (x)
                              (λ (y)
                                (rec Nat x
                                  y
                                  (λ (_)
                                    (λ (sum)
                                      (add1 sum))))))))
                     +
                     (+ (add1 (add1 zero)))
                     ((+ (add1 (add1 zero))) (add1 zero))))
(the (→ Nat (→ Nat Nat))
```

```
   (λ (x) (λ (x*) (rec Nat x x* (λ (x**) (λ (x***) (add1 x***)))))))
 (the (→ Nat Nat)
   (λ (x) (add1 (add1 x))))
 (the Nat
   (add1 (add1 (add1 zero))))
 (go  (list  (cons  '+ (def  '(→ Nat (→ Nat Nat)) (CLOS  '()     'x '(λ (y) (rec Nat x y (λ (_) (λ (sum) (add1 sum))))))))))
```

7 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, `U`, is a `U`. This makes it inconsistent as a logic, but it is still safe as a programming language.

7.1 The Language

The Tartlet syntax consists of S-expressions that include a fixed set of keywords. Any Racket symbol that is not a keyword is a valid variable name. There are three binding forms: `λ`, `Π`, and `Σ`.

```
‹expr› ::= ‹id›
         | ( Π ( ( ‹id› ‹expr› ) ) ‹expr› )
         | ( λ ( ‹id› ) ‹expr› )
         | ( ‹expr› ‹expr› )
         | ( Σ ( ( ‹id› ‹expr› ) ) ‹expr› )
         | ( cons ‹expr› ‹expr› )
         | ( car ‹expr› )
         | ( cdr ‹expr› )
         | Nat
         | zero
         | ( add1 ‹expr› )
         | ( ind-Nat ‹expr› ‹expr› ‹expr› ‹expr› )
         | ( = ‹expr› ‹expr› ‹expr› )
         | same
         | ( replace ‹expr› ‹expr› ‹expr› )
         | Trivial
         | sole
         | Absurd
         | ( ind-Absurd ‹expr› ‹expr› )
         | Atom
         | ' ‹id›
         | U
         | ( the ‹expr› ‹expr› )
```

7.1.1 Identifiers

Identifiers are symbols that are not already keywords. The symbol `quote` is reserved because, internally, `'a` is represented as `(list 'quote a)`.

keywords : `(listof symbol?)` — value

The list of reserved keywords

```
(define keywords
  (list 'define
        'U
        'Nat 'zero 'add1 'ind-Nat
        'Σ 'Sigma 'cons 'car 'cdr
        'Π 'Pi 'λ 'lambda
        '= 'same 'replace
        'Trivial 'sole
        'Absurd 'ind-Absurd
        'Atom 'quote
        'the))
```

(**keyword?** *x*) → boolean? — procedure

x : any/c

Check whether *x* is a valid keyword.

```
(define (keyword? x)
  (if (memv x keywords)
```

```
      #t
      #f))
```

(**var?** *x*) → boolean? procedure
x : any/c

Check whether *x* is a valid variable name.

```
(define (var? x)
  (and (symbol? x)
       (not (keyword? x))))
```

7.1.2 Program α-equivalence

While checking simple types could use `type=?` to compare two types, α-equivalence is needed in a dependently-typed language because types are programs that can bind variables. In the interest of keeping the α-equivalence procedure short, it does not reject invalid programs, so it cannot be used to check for syntactically valid programs the way that the reflexive case of `type=?` could be used to test for syntactically valid simple types.

(**α-equiv?** *e1 e2*) → boolean? procedure
e1 : expression?
e2 : expression?

Determine whether two expressions are α-equivalent.

```
(define (α-equiv? e1 e2)
  (α-equiv-aux e1 e2 '() '()))
```

(**α-equiv-aux** *e1 e2 xs1 xs2*) → boolean? procedure
e1 : expression?
e2 : expression?
xs1 : (listof (pair symbol symbol))
xs2 : (listof (pair symbol symbol))

The real work of checking for α-equivalence uses more state to track bindings. When a binding form is encountered, a fresh symbol is generated (with `gensym`) and saved as the target of both sides. Then, when comparing names, this table of fresh name mappings is consulted to determine whether the variables are bound, and if so, whether they are bound by the same binder.

In the second pattern case, there are three interesting possibilities. If both variables are bound, the `gensym`s representing their binding sites are compared. If they are both free, they are compared directly. If one is bound and the other free, then they will never match because a `gensym` is never equal to a user-provided symbol.

```
(define (α-equiv-aux e1 e2 xs1 xs2)
  (match* (e1 e2)
    [(kw kw)
     #:when (keyword? kw)
     #t]
    [(x y)
     #:when (and (var? x) (var? y))
     (match* ((assv x xs1) (assv y xs2))
       [(#f #f) (eqv? x y)]
       [((cons _ b1) (cons _ b2)) (eqv? b1 b2)]
       [(_ _) #f])]
    [(`(λ (,x) ,b1) `(λ (,y) ,b2))
     (let ([fresh (gensym)])
       (let ([bigger1 (cons (cons x fresh) xs1)]
             [bigger2 (cons (cons y fresh) xs2)])
         (α-equiv-aux b1 b2 bigger1 bigger2)))]
    [(`(Π ((,x ,A1)) ,B1) `(Π ((,y ,A2)) ,B2))
     (and (α-equiv-aux A1 A2 xs1 xs2)
          (let ([fresh (gensym)])
            (let ([bigger1 (cons (cons x fresh) xs1)]
                  [bigger2 (cons (cons y fresh) xs2)])
              (α-equiv-aux B1 B2 bigger1 bigger2))))]
    [(`(Σ ((,x ,A1)) ,B1) `(Σ ((,y ,A2)) ,B2))
     (and (α-equiv-aux A1 A2 xs1 xs2)
          (let ([fresh (gensym)])
            (let ([bigger1 (cons (cons x fresh) xs1)]
                  [bigger2 (cons (cons y fresh) xs2)])
              (α-equiv-aux B1 B2 bigger1 bigger2))))]
    [(`',x  `',y)
     (eqv? x y)]
    ; This, together with read-back-norm, implements the η law for Absurd.
    [(`(the Absurd ,e1) `(the Absurd ,e2))
```

```
      #t]
     [((cons op args1) (cons op args2))
      #:when (keyword? op)
      (and (= (length args1) (length args2))
           (for/and ([arg1 (in-list args1)]
                     [arg2 (in-list args2)])
             (α-equiv-aux arg1 arg2 xs1 xs2)))]
     [((list rator1 rand1) (list rator2 rand2))
      (and (α-equiv-aux rator1 rator2 xs1 xs2)
           (α-equiv-aux rand1 rand2 xs1 xs2))]
     [(_ _) #f]))
```

7.2 Values and Normalization

7.2.1 The Values

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

```
(struct PI (domain range)                                    struct
    #:transparent)
  domain : value?
  range : closure?
(struct LAM (body)                                           struct
    #:transparent)
  body : closure?
(struct SIGMA (car-type cdr-type)                            struct
    #:transparent)
  car-type : value?
  cdr-type : closure?
(struct PAIR (car cdr)                                       struct
    #:transparent)
  car : value?
  cdr : value?
(struct NAT ()                                               struct
    #:transparent)
(struct ZERO ()                                              struct
    #:transparent)
(struct ADD1 (pred)                                          struct
    #:transparent)
  pred : value?
(struct EQ (type from to)                                    struct
    #:transparent)
  type : value?
  from : value?
  to : value?
(struct SAME ()                                              struct
    #:transparent)
(struct TRIVIAL ()                                           struct
    #:transparent)
(struct SOLE ()                                              struct
    #:transparent)
(struct ABSURD ()                                            struct
    #:transparent)
(struct ATOM ()                                              struct
    #:transparent)
(struct QUOTE (symbol)                                       struct
    #:transparent)
  symbol : symbol?
(struct UNI ()                                               struct
    #:transparent)
```

The values represent the introduction forms and the type constructors. Because they do not contain any potential computation other than neutral expressions, they represent only the values of Tartlet.

Just as in Typed Normalization by Evaluation, 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. Because Tartlet types are programs like any other, however, the type annotation is a value.

```
(struct NEU (type neutral)                                    struct
    #:transparent)
  type : value?
  neutral : neutral?
```

Finally, we're going to need to construct type values directly from time to time, and building syntax is much less convenient than building the values directly, due to concerns such as variable capture. So there is one new kind of closure that uses Racket's closures.

```
(struct H-O-CLOS (x fun)                                      struct
    #:transparent)
  x : symbol?
  fun : (-> value? value?)
```

`H-O-CLOS` is an alternative representation of closures that reuses Racket functions and their built-in closures. This is to avoid unnecessary and inconvenient round-tripping through syntax when the type checker needs to construct a type that contains bound variables.

```
(closure? c) → boolean?                                       procedure
  c : any/c
```

Check whether something is one of the two forms of closures.

```
(define (closure? c)
  (or (CLOS? c) (H-O-CLOS? c)))
```

```
(closure-name c) → symbol?                                    procedure
  c : closure?
```

Extract the name of the bound variable in a closure.

```
(define (closure-name c)
  (match c
    [(CLOS _ x _) x]
    [(H-O-CLOS x _) x]))
```

7.2.2 Neutral Expressions

```
(struct N-var (name)                                          struct
    #:transparent)
  name : symbol?
```

At the basis of each neutral expression is a neutral variable.

```
(struct N-ap (fun arg)                                        struct
    #:transparent)
  fun : neutral?
  arg : normal?
(struct N-car (pair)                                          struct
    #:transparent)
  pair : neutral?
(struct N-cdr (pair)                                          struct
    #:transparent)
  pair : neutral?
(struct N-ind-Nat (target motive base step)                   struct
    #:transparent)
  target : neutral?
  motive : normal?
  base : normal?
  step : normal?
(struct N-replace (target motive base)                        struct
    #:transparent)
  target : neutral?
  motive : normal?
  base : normal?
(struct N-ind-Absurd (target motive)                          struct
    #:transparent)
  target : neutral?
  motive : normal?
```

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.

7.2.3 Normal Forms

```
(struct THE (type val)                                    struct
    #:transparent)
  type : value?
  val : value?
```

The internal representation of normal forms, constructed with `"THE"`, pairs a type value with a value classified by the type.

7.3 Definitions and Dependent Types

The simply-typed language of Typed Normalization by Evaluation 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.

```
(struct def (type value)                                  struct
    #:transparent)
  type : value?
  value : value?
(struct bind (type)                                       struct
    #:transparent)
  type : value?
```

The context contains *definitions*, represented by `def`, and *free variable bindings*, represented by `bind`.

```
(context? Γ) → boolean?                                   procedure
  Γ : any/c
```

Determine whether Γ is a context.

```
(define (context? Γ)
  (match Γ
    ['() #t]
    [(cons (cons x b) rest)
     (and (symbol? x) (or (def? b) (bind? b)) (context? rest))]
    [_ #f]))
```

```
(lookup-type x Γ) → (perhaps/c value?)                    procedure
  x : symbol?
  Γ : context?
```

Look up a variable in a context, finding its type.

```
(define (lookup-type x Γ)
  (match (assv x Γ)
    [#f (stop x "Unknown variable")]
    [(cons _ (bind type)) (go type)]
    [(cons _ (def type _)) (go type)]))
```

```
(ctx->env Γ) → environment?                               procedure
  Γ : context?
```

The initial environment for an invocation of the evaluator 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 (`N-var`) in the initial environment.

```
(define (ctx->env Γ)
  (map (lambda (binder)
         (match binder
           [(cons name (bind type))
            (cons name
                  (NEU type
                       (N-var name)))]
```

```
            [(cons name (def _ value))
             (cons name value)]))
       Γ))
```

(**extend-ctx** *Γ x type*) → context? procedure

Γ : context?
x : symbol?
type : value?

Extend a context with the type of a free variable.

```
(define (extend-ctx Γ x t)
  (cons (cons x (bind t)) Γ))
```

7.3.1 The Evaluator

(**val-of-closure** *c v*) → value? procedure

c : closure?
v : value?

Evaluate the closure *c*, instantiating its bound variable with *v*.

```
(define (val-of-closure c v)
  (match c
    [(CLOS ρ x e) (val (extend ρ x v) e)]
    [(H-O-CLOS x f) (f v)]))
```

(**val** *ρ e*) → value? procedure

ρ : environment?
e : expression?

```
(define (val ρ e)
  (match e
    [`(the ,type ,expr)
     (val ρ expr)]
    ['U (UNI)]
    [`(Π ((,x ,A)) ,B)
     (PI (val ρ A) (CLOS ρ x B))]
    [`(λ (,x) ,b)
     (LAM (CLOS ρ x b))]
    [`(Σ ((,x ,A)) ,D)
     (SIGMA (val ρ A) (CLOS ρ x D))]
    [`(cons ,a ,d)
     (PAIR (val ρ a) (val ρ d))]
    [`(car ,pr)
     (do-car (val ρ pr))]
    [`(cdr ,pr)
     (do-cdr (val ρ pr))]
    ['Nat (NAT)]
    ['zero (ZERO)]
    [`(add1 ,n) (ADD1 (val ρ n))]
    [`(ind-Nat ,target ,motive ,base ,step)
     (do-ind-Nat (val ρ target) (val ρ motive) (val ρ base) (val ρ step))]
    [`(= ,A ,from ,to)
     (EQ (val ρ A) (val ρ from) (val ρ to))]
    ['same
     (SAME)]
    [`(replace ,target ,motive ,base)
     (do-replace (val ρ target) (val ρ motive) (val ρ base))]
    ['Trivial (TRIVIAL)]
    ['sole (SOLE)]
    ['Absurd (ABSURD)]
    [`(ind-Absurd ,target ,motive) (do-ind-Absurd (val ρ target) (val ρ motive))]
    ['Atom (ATOM)]
    [`',a  (QUOTE a)]
    [`(,rator ,rand)
     (do-ap (val ρ rator) (val ρ rand))]
    [x #:when (var? x)
     (cdr (assv x ρ))]))
```

7.3.2 Eliminators

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

(**do-car** *v*) → value? procedure
  v : value?
(**do-cdr** *v*) → value? procedure
  v : value?

Note that the `cdr` of a neutral expression contains the `car` of that expression in its type annotation.

```
(define (do-car v)
  (match v
    [(PAIR a d) a]
    [(NEU (SIGMA A _) ne)
     (NEU A (N-car ne))]))
(define (do-cdr v)
  (match v
    [(PAIR a d) d]
    [(NEU (SIGMA _ D) ne)
     (NEU (val-of-closure D (do-car v))
          (N-cdr ne))]))
```

(**do-ap** *fun arg*) → value? procedure
  fun : value?
  arg : value?

Implement function application (β-reduction), taking neutral functions into account.

```
(define (do-ap fun arg)
  (match fun
    [(LAM c)
     (val-of-closure c arg)]
    [(NEU (PI A B) ne)
     (NEU (val-of-closure B arg)
          (N-ap ne (THE A arg)))]))
```

(**do-ind-Absurd** *target motive*) → value? procedure
  target : value?
  motive : value?

Because every `Absurd` is neutral, `do-ind-Absurd` has only cases for neutral targets.

```
(define (do-ind-Absurd target motive)
  (match target
    [(NEU (ABSURD) ne)
     (NEU motive (N-ind-Absurd ne (THE (UNI) motive)))]))
```

(**do-replace** *target motive base*) → value? procedure
  target : value?
  motive : value?
  base : value?

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

```
(define (do-replace target motive base)
  (match target
    [(SAME) base]
    [(NEU (EQ A from to) ne)
     (NEU (do-ap motive to)
          (N-replace ne
                     (THE (PI A (H-O-CLOS 'x (lambda (_) (UNI))))
                          motive)
                     (THE (do-ap motive from)
                          base)))]))
```

(**do-ind-Nat** *target motive base step*) → value? procedure
  target : value?
  motive : value?
  base : value?
  step : value?
(**ind-Nat-step-type** *motive*) → value? procedure
  motive : value?

The run-time implementation of `ind-Nat` uses the helper `ind-Nat-step-type` to construct the type of the step for `ind-Nat`.

```
(define (do-ind-Nat target motive base step)
  (match target
    [(ZERO) base]
    [(ADD1 n) (do-ap (do-ap step n) (do-ind-Nat n motive base step))]
    [(NEU (NAT) ne)
     (NEU (do-ap motive target)
          (N-ind-Nat
           ne
           (THE (PI (NAT)
                    (H-O-CLOS 'k (lambda (k) (UNI))))
                motive)
           (THE (do-ap motive (ZERO)) base)
           (THE (ind-Nat-step-type motive)
                step)))]))
(define (ind-Nat-step-type motive)
  (PI (NAT)
      (H-O-CLOS 'n-1
                (lambda (n-1)
                  (PI (do-ap motive n-1)
                      (H-O-CLOS 'ih
                                (lambda (ih)
                                  (do-ap motive (ADD1 n-1)))))))))
```

7.3.3 Reading Back

Just as in Typed Normalization by Evaluation, reading back from values into syntax is accomplished via two mutually-recursive procedures: `read-back-norm` and `read-back-neutral`.

(**read-back-norm** *Γ norm*) → expression? — procedure

Γ : context?

norm : norm?

Convert a normal value into the syntax that represents it.

```
(define (read-back-norm Γ norm)
  (match norm
    [(THE (NAT) (ZERO)) 'zero]
    [(THE (NAT) (ADD1 n))
     `(add1 ,(read-back-norm Γ (THE (NAT) n)))]
    [(THE (PI A B) f)
     (define x (closure-name B))
     (define y (freshen (map car Γ) x))
     (define y-val (NEU A (N-var y)))
     `(λ (,y)
        ,(read-back-norm (extend-ctx Γ y A)
                         (THE (val-of-closure B y-val)
                              (do-ap f y-val))))]
    [(THE (SIGMA A D) p)
     (define the-car (THE A (do-car p)))
     (define the-cdr (THE (val-of-closure D the-car) (do-cdr p)))
     `(cons ,(read-back-norm Γ the-car) ,(read-back-norm Γ the-cdr))]
    [(THE (TRIVIAL) _) 'sole]
    [(THE (ABSURD) (NEU (ABSURD) ne))
     `(the Absurd
           ,(read-back-neutral Γ ne))]
    [(THE (EQ A from to) (SAME)) 'same]
    [(THE (ATOM) (QUOTE x)) `',x]
    [(THE (UNI) (NAT)) 'Nat]
    [(THE (UNI) (ATOM)) 'Atom]
    [(THE (UNI) (TRIVIAL)) 'Trivial]
    [(THE (UNI) (ABSURD)) 'Absurd]
    [(THE (UNI) (EQ A from to))
     `(= ,(read-back-norm Γ (THE (UNI) A))
         ,(read-back-norm Γ (THE A from))
         ,(read-back-norm Γ (THE A to)))]
    [(THE (UNI) (SIGMA A D))
     (define x (closure-name D))
     (define y (freshen (map car Γ) x))
     `(Σ ((,y ,(read-back-norm Γ (THE (UNI) A))))
         ,(read-back-norm (extend-ctx Γ y A)
                          (THE (UNI) (val-of-closure D (NEU A (N-var y))))))]
    [(THE (UNI) (PI A B))
     (define x (closure-name B))
     (define y (freshen (map car Γ) x))
     `(Π ((,y ,(read-back-norm Γ (THE (UNI) A))))
         ,(read-back-norm (extend-ctx Γ y A)
                          (THE (UNI) (val-of-closure B (NEU A (N-var y))))))]
    [(THE (UNI) (UNI)) 'U]
```

```
    [(THE t1 (NEU t2 ne))
     (read-back-neutral Γ ne)]))
```

(**read-back-neutral** *Γ neu*) → expression? procedure

Γ : context?

neu : neutral?

Convert a neutral expression into its representation as syntax. The only case that is not immediately the same as the others is that for `ind-Absurd`; it adds a type annotation to its target that, together with a special case in α-equiv?, causes all neutral inhabitants of `Absurd` to be identified with one another.

```
(define (read-back-neutral Γ neu)
  (match neu
    [(N-var x) x]
    [(N-ap ne rand)
     `(,(read-back-neutral Γ ne)
       ,(read-back-norm Γ rand))]
    [(N-car ne) `(car ,(read-back-neutral Γ ne))]
    [(N-cdr ne) `(cdr ,(read-back-neutral Γ ne))]
    [(N-ind-Nat ne motive base step)
     `(ind-Nat ,(read-back-neutral Γ ne)
               ,(read-back-norm Γ motive)
               ,(read-back-norm Γ base)
               ,(read-back-norm Γ step))]
    [(N-replace ne motive base)
     `(replace ,(read-back-neutral Γ ne)
               ,(read-back-norm Γ motive)
               ,(read-back-norm Γ base))]
    [(N-ind-Absurd ne motive)
     `(ind-Absurd (the Absurd ,(read-back-neutral Γ ne))
                  ,(read-back-norm Γ motive))]))
```

7.4 Type Checking

7.4.1 The Type Checker

Like Pie and many other implementations of type theory, the Tartlet type checker is an *elaborating* type checker. This means that, instead of simply indicating that an expression is well-typed, it returns a version of the expression with more details inserted. In other words, the language accepted by the type checker contains structures that are not understood by the normalizer, and the type checker emits expressions in that simpler language.

In an elaborating bidirectional type checker, checking emits an elaborated equivalent of the input expression, while synthesis emits both an elaborated expression and its type, with its type being in the core language of the normalizer. Many elaborating type checkers also re-check the expression in the simpler language; to keep these notes shorter, Tartlet dispenses with that step.

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.

(**synth** *Γ e*) → (list/c 'the expr? expr?) procedure

Γ : context?

e : expr?

Type synthesis constructs a type by examining an expression, returning both.

```
(define (synth Γ e)
  (match e
    [`(the ,type ,expr)
     (go-on ([t-out (check Γ type (UNI))]
             [e-out (check Γ expr (val (ctx->env Γ) t-out))])
       (go `(the ,t-out ,e-out)))]
    ['U
     (go '(the U U))]
    [`(,(or 'Σ 'Sigma) ((,x ,A)) ,D)
     (go-on ([A-out (check Γ A (UNI))]
             [D-out (check (extend-ctx Γ x (val (ctx->env Γ) A-out)) D (UNI))])
       (go `(the U (Σ ((,x ,A-out)) ,D-out))))]
    [`(car ,pr)
     (go-on ([`(the ,pr-ty ,pr-out) (synth Γ pr)])
       (match (val (ctx->env Γ) pr-ty)
```

```
        [(SIGMA A D)
         (go `(the ,(read-back-norm Γ (THE (UNI) A)) (car ,pr-out)))]
        [non-SIGMA
         (stop e (format "Expected Σ, got ~v"
                         (read-back-norm Γ (THE (UNI) non-SIGMA))))]))]
    [`(cdr ,pr)
     (go-on ([`(the ,pr-ty ,pr-out) (synth Γ pr)])
       (match (val (ctx->env Γ) pr-ty)
         [(SIGMA A D)
          (define the-car (do-car (val (ctx->env Γ) pr-out)))
          (go `(the ,(read-back-norm Γ (THE (UNI) (val-of-closure D the-car)))
                    (cdr ,pr-out)))]
         [non-SIGMA
          (stop e (format "Expected Σ, got ~v"
                          (read-back-norm Γ (THE (UNI) non-SIGMA))))]))]
    ['Nat (go '(the U Nat))]
    [`(ind-Nat ,target ,motive ,base ,step)
     (go-on ([target-out (check Γ target (NAT))]
             [motive-out (check Γ motive (PI (NAT) (H-O-CLOS 'n (lambda (_) (UNI)))))]
             [motive-val (go (val (ctx->env Γ) motive-out))]
             [base-out (check Γ base (do-ap motive-val (ZERO)))]
             [step-out (check Γ
                              step
                              (ind-Nat-step-type motive-val))])
       (go `(the ,(read-back-norm
                   Γ
                   (THE (UNI)
                        (do-ap motive-val (val (ctx->env Γ) target-out))))
                 (ind-Nat ,target-out ,motive-out ,base-out ,step-out))))]
    [`(= ,A ,from ,to)
     (go-on ([A-out (check Γ A (UNI))]
             [A-val (go (val (ctx->env Γ) A-out))]
             [from-out (check Γ from A-val)]
             [to-out (check Γ to A-val)])
       (go `(the U (= ,A-out ,from-out ,to-out))))]
    [`(replace ,target ,motive ,base)
     (go-on ([`(the ,target-t ,target-out) (synth Γ target)])
       (match (val (ctx->env Γ) target-t)
         [(EQ A from to)
          (go-on ([motive-out
                   (check Γ
                          motive
                          (PI A (H-O-CLOS 'x (lambda (x) (UNI)))))]
                  [motive-v (go (val (ctx->env Γ) motive-out))]
                  [base-out (check Γ base (do-ap motive-v from))])
            (go `(the ,(read-back-norm Γ (THE (UNI) (do-ap motive-v to)))
                      (replace ,target-out ,motive-out ,base-out))))]
         [non-EQ
          (stop target (format "Expected =, but type is ~a" non-EQ))]))]
    [`(,(or 'Π 'Pi) ((,x ,A)) ,B)
     (go-on ([A-out (check Γ A (UNI))]
             [B-out (check (extend-ctx Γ x (val (ctx->env Γ) A-out)) B (UNI))])
       (go `(the U (Π ((,x ,A-out)) ,B-out))))]
    ['Trivial (go '(the U Trivial))]
    ['Absurd (go '(the U Absurd))]
    [`(ind-Absurd ,target ,motive)
     (go-on ([target-out (check Γ target (ABSURD))]
             [motive-out (check Γ motive (UNI))])
       (go `(the ,motive-out (ind-Absurd ,target-out ,motive-out))))]
    ['Atom (go '(the U Atom))]
    [`(,rator ,rand)
     (go-on ([`(the ,rator-t ,rator-out) (synth Γ rator)])
       (match (val (ctx->env Γ) rator-t)
         [(PI A B)
          (go-on ([rand-out (check Γ rand A)])
            (go `(the ,(read-back-norm Γ
                                       (THE (UNI)
                                            (val-of-closure B
                                                            (val (ctx->env Γ)
                                                                 rand-out))))
                      (,rator-out ,rand-out))))]
         [non-PI (stop rator
                       (format "Expected a Π type, but this is a ~a"
                               (read-back-norm Γ (THE (UNI) non-PI))))]))]
    [x #:when (var? x)
     (go-on ([t (lookup-type x Γ)])
       (go `(the ,(read-back-norm Γ (THE (UNI) t)) ,x)))]
    [none-of-the-above (stop e "Can't synthesize a type")]))
```

(**check** *Γ e t*) → (perhaps/c expr?) procedure

Γ : context?
e : expr?
t : value?

Determine whether *e* has type *t*, returning the elaborated expression on success.

```
(define (check Γ e t)
  (match e
    [`(cons ,a ,d)
     (match t
       [(SIGMA A D)
        (go-on ([a-out (check Γ a A)]
                [d-out (check Γ d (val-of-closure D (val (ctx->env Γ) a-out)))])
          (go `(cons ,a-out ,d-out)))]
       [non-SIGMA (stop e (format "Expected Σ, got ~v"
                                  (read-back-norm Γ (THE (UNI) non-SIGMA))))])]
    ['zero
     (match t
       [(NAT) (go 'zero)]
       [non-NAT (stop e (format "Expected Nat, got ~v"
                                (read-back-norm Γ (THE (UNI) non-NAT))))])]
    [`(add1 ,n)
     (match t
       [(NAT)
        (go-on ([n-out (check Γ n (NAT))])
          (go `(add1 ,n-out)))]
       [non-NAT (stop e (format "Expected Nat, got ~v"
                                (read-back-norm Γ (THE (UNI) non-NAT))))])]
    ['same
     (match t
       [(EQ A from to)
        (go-on ([_ (convert Γ A from to)])
          (go 'same))]
       [non-= (stop e (format "Expected =, got ~v"
                              (read-back-norm Γ (THE (UNI) non-=))))])]
    ['sole
     (match t
       [(TRIVIAL)
        (go 'sole)]
       [non-Trivial (stop e (format "Expected Trivial, got ~v"
                                    (read-back-norm Γ (THE (UNI) non-Trivial))))])]
    [`(,(or 'λ 'lambda) (,x) ,b)
     (match t
       [(PI A B)
        (define x-val (NEU A (N-var x)))
        (go-on ([b-out (check (extend-ctx Γ x A) b (val-of-closure B x-val))])
          (go `(λ (,x) ,b-out)))]
       [non-PI (stop e (format "Expected Π, got ~v"
                               (read-back-norm Γ (THE (UNI) non-PI))))])]
    [`',a
     (match t
       [(ATOM)
        (go `',a)]
       [non-ATOM (stop e (format "Expected Atom, got ~v"
                                 (read-back-norm Γ (THE (UNI) non-ATOM))))])]
    [none-of-the-above
     (go-on ([`(the ,t-out ,e-out) (synth Γ e)]
             [_ (convert Γ (UNI) t (val (ctx->env Γ) t-out))])
       (go e-out))]))
```

(**convert** *Γ t v1 v2*) → (perhaps/c 'ok) procedure

Γ : context?
t : value?
v1 : value?
v2 : value?

Check whether *v1* and *v2* represent the same *t*—in other words, whether their representations as syntax are α-equivalent.

```
(define (convert Γ t v1 v2)
  (define e1 (read-back-norm Γ (THE t v1)))
  (define e2 (read-back-norm Γ (THE t v2)))
  (if (α-equiv? e1 e2)
      (go 'ok)
      (stop e1 (format "Expected to be the same ~v as ~v"
                       (read-back-norm Γ (THE (UNI) t))
                       e2))))
```

7.4.2 Type Checking with Definitions

(**interact** *Γ input*) → (perhaps/c context?) procedure

Γ : context?

input : (or/c (list/c 'define symbol? expression?)
expression?)

If *input* is an expression, check and normalize it. If *input* is a definition, check it and add it to the context.

```
(define (interact Γ input)
  (match input
    [`(define ,x ,e)
     (if (assv x Γ)
         (stop x "Already defined")
         (go-on ([`(the ,ty ,expr) (synth Γ e)])
           (let ([ρ (ctx->env Γ)])
             (go (cons (cons x (def (val ρ ty) (val ρ expr)))
                       Γ))))))]
    [e
     (go-on ([`(the ,ty ,expr) (synth Γ e)])
       (let ([ρ (ctx->env Γ)])
         (begin
           (printf "Type: ~v\nNormal form:~v\n"
                   ty
                   (read-back-norm Γ
                                   (THE (val ρ ty)
                                        (val ρ expr))))
           (go Γ))))]))
```

(**run-program** *Γ input*) → (perhaps/c context?) procedure

Γ : context?

input : (listof (or/c (list/c 'define symbol? expression?)
expression?))

Check a series of definitions and expressions.

```
(define (run-program Γ inputs)
  (match inputs
    ['() (go Γ)]
    [(cons d rest)
     (go-on ([new-Γ (interact Γ d)])
       (run-program new-Γ rest))]))
```

Example:

```
> (void
   (run-program '()
                '(; What are the consequences of Nat equality?
                  (define nat=consequence
                    (the (Pi ((j Nat))
                           (Pi ((k Nat))
                             U))
                         (lambda (j)
                           (lambda (k)
                             (ind-Nat j
                                      (lambda (_) U)
                                      (ind-Nat k
                                               (lambda (_) U)
                                               Trivial
                                               (lambda (_)
                                                 (lambda (_)
                                                   Absurd)))
                                      (lambda (j-1)
                                        (lambda (_)
                                          (ind-Nat k
                                                   (lambda (_) U)
                                                   Absurd
                                                   (lambda (k-1)
                                                     (lambda (_)
                                                       (= Nat j-1 k-1))))))))))))
                  ; The consequences hold for Nats that are the same
                  (define nat=consequence-refl
                    (the (Pi ((n Nat))
                           ((nat=consequence n) n))
                         (lambda (n)
                           (ind-Nat n
                                    (lambda (k)
                                      ((nat=consequence k) k))
                                    sole
```

```
                                  (lambda (n-1)
                                    (lambda (_)
                                      same))))))
                    (nat=consequence-refl zero)
                    (nat=consequence-refl (add1 (add1 zero)))
                    ; The consequences hold for all equal Nats
                    (define there-are-consequences
                      (the (Pi ((j Nat))
                             (Pi ((k Nat))
                               (Pi ((j=k (= Nat j k)))
                                 ((nat=consequence j) k))))
                           (lambda (j)
                             (lambda (k)
                               (lambda (j=k)
                                 (replace j=k
                                          (lambda (n)
                                            ((nat=consequence j) n))
                                          (nat=consequence-refl j)))))))
                    ((there-are-consequences zero) zero)
                    (((there-are-consequences zero) zero) same)
                    ((there-are-consequences (add1 zero)) (add1 zero))
                    (((there-are-consequences (add1 zero)) (add1 zero)) same)
                    ((there-are-consequences zero) (add1 zero))
                    ((there-are-consequences (add1 zero)) zero))))
Type: 'Trivial
Normal form:'sole
Type: '(= Nat (add1 zero) (add1 zero))
Normal form:'same
Type: '(Π ((j=k (= Nat zero zero))) Trivial)
Normal form:'(λ (j=k) sole)
Type: 'Trivial
Normal form:'sole
Type: '(Π ((j=k (= Nat (add1 zero) (add1 zero)))) (= Nat zero zero))
Normal form:'(λ (j=k) (replace j=k (λ (x) (ind-Nat x (λ (k) U) Absurd (λ (n-1) (λ (ih) (= Nat zero n-1))))) same))
Type: '(= Nat zero zero)
Normal form:'same
Type: '(Π ((j=k (= Nat zero (add1 zero)))) Absurd)
Normal form:'(λ (j=k) (the Absurd (replace j=k (λ (x) (ind-Nat x (λ (k) U) Trivial (λ (n-1) (λ (ih) Absurd)))) sole)))
Type: '(Π ((j=k (= Nat (add1 zero) zero))) Absurd)
Normal form:'(λ (j=k) (the Absurd (replace j=k (λ (x) (ind-Nat x (λ (k) U) Absurd (λ (n-1) (λ (ih) (= Nat zero n-1))))) same)))
```

7.5 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`.
2. Add lists and vectors (length-indexed lists).
3. Add non-dependent function types and non-dependent pair types to the type checker. This should not require modifications to the evaluator.
4. 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. Make the evaluation strategy lazy, either call-by-name or better yet call-by-need.
7. Replace `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.

7.6 Putting It Together

Checking dependent types requires answering two questions:

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

In this tutorial, the first question was answered using normalization by evaluation, and the second using bidirectional type checking. Elaboration is used during type checking to emit a simplified core language that is suitable for evaluation. These are not the only potential answers.

Equality 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* [Watkins02] can be used to *only* ever have normal forms of expressions. Another change that can be made to checking the equality judgment is, instead of returning a trivial value on success, to return the conditions under which the judgment would hold. For instance, if expressions can contain metavariables that stand for omitted parts of programs that the programmer expects to be automatically inferrable, then equality checking can emit a collection of constraints over these metavariables to be solved by a separate pass. This is the approach taken in Agda [Norell07].

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 [Brady13].

8 Further Reading

8.1 Tutorials on Implementing Type Theory

Andres Löh, Conor McBride, and Wouter Swierstra [Löh10] wrote a tutorial implementation of dependent types using bidirectional type checking in Haskell, and Lennart Augustsson wrote a response that uses simpler language features

Stephanie Weirich taught a course 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.

8.2 Bidirectional Type Checking

Pierce and Turner [Pierce98] 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 [Dunfield13] for a good survey as of 2013.

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

8.3 Normalization by Evaluation

Normalization by Evaluation was invented by Berger and Schwichtenberg [Berger91] to implement simply typed normalization efficiently by re-using Scheme functions as the semantics of functions in the simply typed λ-calculus. Olivier Danvy [Danvy96] extended their scheme to features such as recursion, sums, and effects, calling the resulting generalized system "Type-Directed Partial Evaluation."

James Chapman, Thorsten Altenkirch, and Conor McBride wrote an implementation of type theory using type-directed [Chapman06] 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 [Abel13] 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.

8.4 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 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.

Thierry Coquand [Coquand96] describes a bidirectional type checker (two years prior to Pierce and Turner [Pierce98]) 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.

Benjamin Grégoire and Xavier Leroy [Grégoire02] implemented a compiler from Coq to a versrion 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 [Kleeblatt11] 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.

Bibliography

[Abel13] Andres Abel, “Normalization by Evaluation: Dependent Types and Impredicativity,” Habilitation Thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, 2013.

[Berger91] Ulrich Berger and Helmut Schwichtenberg, “An inverse to the evaluation functional for typed λ-calculus,” Logic in Computer Science, 1991.

[Brady13] Edwin Brady, “Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation,” Journal of Functional Programming 23 (5), pp. 552–593, 2013.

[Chapman06] James Chapman, Thorsten Altenkirch, and Conor McBride, “Epigram reloaded: a standalone typechecker for ETT,” Post-Proceedings of Trends in Functional Programming, 2005.

[Coquand96] Thierry Coquand, “An algorithm for type-checking dependent types,” Science of Computer Programming 26, pp. 167–177, 1996.

[Danvy96] Olivier Danvy, “Type-Directed Partial Evaluation,” Symposium on Principles of Programming Languages, 1996.

[Dunfield13] Joshua Dunfield and Neelakantan Krishnaswami, “Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism,” International Conference on Functional Programming, 2013.

[Friedman18] Daniel P. Friedman and David Thrane Christiansen, *The Little Typer*, MIT Press, 2018. http://thelittletyper.com

[Grégoire02] Benjamin Grégoire and Xavier Leroy, “A compiled implementation of strong reduction,” International Conference on Functional Programming, 2002.

[Kleeblatt11] Dirk Kleeblatt, “On a Strongly Normalizing STG Machine With an Application to Dependent Type Checking,” PhD Thesis, Technical University of Berlin, 2011.

[Landin64] Peter Landin, “The mechanical evaluation of expressions,” Computer Journal 6, pp. 308–320, 1964.

[Löh10] Andres Löh, Conor McBride, Wouter Swierstra, “A tutorial implementation of a dependently typed lambda calculus,” Fundamenta Informaticae (102) 2, pp. 177-207, 2010.

[Norell07] Ulf Norell, “Towards a practical programming language based on dependent type theory,” PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

[Pierce98] Benjamin C. Pierce and David N. Turner, “Local Type Inference,” Symposium on Principles of Programming Languages, 1998.

[Watkins02] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker, “A concurrent logical framework I: Judgments and properties,” Technical Report CMU-CS-02-101, Carnegie Mellon University, School of Computer Science, 2002, revised May 2003.

9 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`, Rutvik Patel, Siyuan Chen, and Xie Yuheng 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.

Index