9

Double Your Money, Get Twice as Much

In chapter 8, there is only one eliminator for =, called **cong**.

But **cong** has one key limitation.

[1] What is that?

What is the type of a **cong**-expression?

[2] By the Law of **cong**, if *target* is an

(= *X from to*)

and *f* is an

(→ *X*
Y),

then

(**cong** *target f*)

is an

(= *Y* (*f from*) (*f to*)).

That's right.

How is this different from eliminators such as **ind-Nat**?

[3] An **ind-Nat**-expression can have *any* type—it all depends on the motive. But a **cong**-expression's type always has = at the top.

---

cong is a special-purpose eliminator. But there is also a more general eliminator, called **replace**.

[4] What does **replace** mean?

If two expressions are equal, then whatever is true for one is true for the other. We call this principle Leibniz's Law.†

[5] What does this have to do with **replace**?

†Leibniz's Law is also used to refer to the principle that if whatever is true for one is true for the other, then they are equal. Thank you, Gottfried Wilhelm Leibniz (1646–1716).

---

replace is more powerful than **cong** because any use of **cong** can be rewritten to a use of **replace**, just as any use of **which-Nat**, **iter-Nat**, or **rec-Nat** can be rewritten to a use of **ind-Nat**.

[6] How does **replace** differ from **cong**?

Like **cong**, **replace**'s target is an

(= *X from to*).

Unlike **cong**, however, **replace** has a motive and a base.

[7] Is a **replace**-expression's type determined by applying the motive to the target?

That is how the motive works in **ind-Nat**, but not in **replace**.

In **replace**, the motive explains *what* is true for both expressions in Leibniz's Law. It is an

(→ *X*
  $\mathcal{U}$)

because it explains how to find a $\mathcal{U}$ (and therefore a statement) from an *X*.

[8] What about the base?

The base is evidence that (*mot from*) is true. That is, the base's type is

(*mot from*).

[9] What about the whole **replace**-expression?

The whole **replace**-expression is evidence that (*mot to*) is true. In other words, its type is

(*mot to*).

[10] So **replace** replaces *from* with *to*.

Take a look at ***step-incr=add1*** in frame 8:80 on page 191.

[11] Okay. It is defined using **cong**.

The Law of replace

If *target* **is an**
  (**=** *X from to*),
mot **is an**
  (→ *X*
    $\mathcal{U}$),
and *base* **is a**
  (*mot from*)
then
  (**replace** *target*
    mot
    base)
is a
  (*mot to*).

That's right.

But it could also be defined using **replace**.

What is the **claim** again?

[12] Using the observation about ***incr*** in frame 8:68, the add1 is already on the outside of the ***incr*** as if it were ready for **cong**.

```
(claim step-incr=add1
  (Π ((n-1 Nat))
    (→ (= Nat
          (incr n-1)
          (add1 n-1))
       (= Nat
         (add1
           (incr n-1))
         (add1
           (add1 n-1))))))
```

Here is the start of a definition using **replace**.

```
(define step-incr=add1
  (λ (n-1)
    (λ (incr=add1n-1)
      (replace incr=add1n-1
               [          ]
               [       ])))))
```

The target is $incr{=}add1_{n\text{-}1}$, which is the only available proof of equality here.

[13] The target, $incr{=}add1_{n\text{-}1}$, is an

```
(= Nat
  (incr n-1)
  (add1 n-1)).
```

The whole **replace**-expression should be an

```
(= Nat
  (add1
    (incr n-1))
  (add1
    (add1 n-1))).
```

To find the motive, examine the **replace**-expression's type.

Look for the TO of the target's type.

[14] The TO is (add1 *n-1*), which is certainly in the **replace**-expression's type.

```
(= Nat
  (add1
    (incr n-1))
  (add1
    [(add1 n-1)]))
```

The motive is used to find the types of both the base and the whole **replace**-expression. The base's type is found by placing the target's type's FROM in the gray box, while the entire expression's type is found by placing the target's type's TO in the gray box.

```
(= Nat
  (add1
    (incr n-1))
  (add1
    [          ]))
```

[15] An expression that is missing a piece can be written as a λ-expression.

To find the motive, abstract over the TO of the target's type with a λ.

[16] That gives this expression:

```
(λ (k)
  (= Nat
    (add1
      (incr n-1))
    (add1
      k))).
```

But if **replace** replaces the FROM with the TO, why should we abstract over the TO, rather than the FROM?

The base's type is found by applying the motive to the FROM of the target's type. So, in this case, it is

1.
```
((λ (k)
   (= Nat
     (add1
       (incr n-1))
     (add1
       k)))
 (incr n-1))
```
2.
```
(= Nat
  (add1
    (incr n-1))
  (add1
    (incr n-1)))
```

[17] Applying the motive to an argument is like filling in the gray box.

```
(= Nat
  (add1
    (incr n-1))
  (add1
    (incr n-1) )).
```

Now that we know the base's type, what is the base?

[18] The base is

```
(same
  (add1
    (incr n-1)))
```

and leads to

```
(define step-incr=add1
  (λ (n-1)
    (λ (incr=add1_n-1)
      (replace incr=add1_n-1
               [          ]
        (same (add1 (incr n-1)))))))
```

Now define the motive.

[19] The motive takes *n-1* as an argument, just as ***step-∗*** takes *j* as an argument.

```
(claim mot-step-incr=add1
  (→ Nat Nat
    𝒰))
(define mot-step-incr=add1
  (λ (n-1 k)
    (= Nat
      (add1
        (incr n-1))
      (add1
        k))))
```

Finally, complete the definition from frame 17.

[20] Because ***step-incr=add1*** is already defined in chapter 8, this remains in a dashed box.

```
(define step-incr=add1
  (λ (n-1)
    (λ (incr=add1_n-1)
      (replace incr=add1_n-1
        (mot-step-incr=add1 n-1)
        (same (add1 (incr n-1)))))))
```

Yes, only one definition for each claim.

Now, define ***double*** to be a function that replaces each add1 in a Nat with two add1s.

```
(claim double
  (→ Nat
    Nat))
```

[21] This is a job for **iter-Nat**. The step is (**+** 2) because the normal form of (**+** 2) is

```
(λ (j)
  (add1
    (add1 j))).
```

```
(define double
  (λ (n)
    (iter-Nat n
      0
      (+ 2))))
```

(***double*** *n*) is twice as big as *n*. What is another function that finds the same answer? Call it ***twice***.

```
(claim twice
  (→ Nat
    Nat))
```

[22] How about this?

```
(define twice
  (λ (n)
    (+ n n)))
```

It happens to be the case that,

"For every Nat *n*, (***twice*** *n*) equals (***double*** *n*)."

How can this statement be written as a type?

[23] Because this statement is likely to get a proof, it gets a name.

```
(claim twice=double
  (Π ((n Nat))
    (= Nat (twice n) (double n))))
```

Very perceptive.

Why is this claim true?

[24] Every Nat value is either zero or has add1 at the top. Both (***twice*** zero) and (***double*** zero) are zero.

What about add1?

[25] For add1,
(***twice*** (add1 *n-1*))
is the same Nat as
(**+** (add1 *n-1*) (add1 *n-1*)),
but
(***double*** (add1 *n-1*))
is the same Nat as
(add1 (add1 (***double*** *n-1*))).

Is
(**+** (add1 *n-1*) (add1 *n-1*))
the same Nat as
(add1 (add1 (**+** *n-1* *n-1*)))?

[26] No, it isn't.

But surely they must be equal.

That's right.

To prove ***twice=double***, an extra proof is needed. While an add1 around **+**'s *first* argument can be moved above **+**, an add1 around **+**'s *second* argument cannot be—at least not without a proof.

[27] Right, because only the first argument is the target of **iter-Nat** in **+**'s definition.

Even though
(**+** *n* (add1 *j*))
is not the same Nat as
(add1 (**+** *n* *j*)),
they are equal Nats.

[28] They are not the same, but one can be **replace**d with the other.

The statement to be proved is ***add1+=+add1***.

```
(claim add1+=+add1
  (Π ((n Nat)
      (j Nat))
    (= Nat
      (add1 (+ n j))
      (+ n (add1 j)))))
```

[29] This looks like a job for **ind-Nat**.

```
(define add1+=+add1
  (λ (n j)
    (ind-Nat n
      (mot-add1+=+add1 j)
      (same (add1 j))
      (step-add1+=+add1 j))))
```

The motive and the step both need j, just like ***step-∗***. The base is

(same (add1 j)).

Why is the base

(same (add1 j))?

[30] Because

(add1 (+ zero j))

is the same Nat as

(add1 j)

and

(+ zero (add1 j))

is also the same Nat as

(add1 j).

What is ***mot-add1+=+add1***?

[31] It is the type of the **ind-Nat**-expression, abstracted over the target. In other words, every occurrence of n in the claim ***add1+=+add1*** becomes k.

```
(claim mot-add1+=+add1
  (→ Nat Nat
    𝒰))
(define mot-add1+=+add1
  (λ (j k)
    (= Nat
      (add1 (+ k j))
      (+ k (add1 j)))))
```

Here is ***step-add1+=+add1***'s type.

```
(claim step-add1+=+add1
  (Π ((j Nat)
      (n-1 Nat))
    (→ (mot-add1+=+add1 j
         n-1)
      (mot-add1+=+add1 j
        (add1 n-1)))))
```

What is a more explicit way to write

```
(mot-add1+=+add1 j
  (add1 n-1))?
```

[32] Applying ***mot-add1+=+add1*** gives

```
(= Nat
  (add1 (+ (add1 n-1) j))
  (+ (add1 n-1) (add1 j))).
```

That type and

```
(= Nat
  (add1 (add1 (+ n-1 j)))
  (add1 (+ n-1 (add1 j))))
```

are the same type. This is because the first argument to + is the target of iter-Nat.

Now define ***step-add1+=+add1***.

[33] It uses cong.

```
(define step-add1+=+add1
  (λ (j n-1)
    (λ (add1+=+add1_n-1)
      (cong add1+=+add1_n-1
        (+ 1)))))
```

What role do cong and (+ 1) play in the definition?

[34] $add1{+}{=}{+}add1_{n\text{-}1}$ is an

```
(= Nat
  (add1 (+ n-1 j))
  (+ n-1 (add1 j))),
```

so using (+ 1) with cong wraps both the FROM and the TO with add1, which gives the type from frame 32.

The definition of ***add1+=+add1*** now deserves a solid box because every name that it mentions is now defined.

[35] Here it is.

```
(define add1+=+add1
  (λ (n j)
    (ind-Nat n
      (mot-add1+=+add1 j)
      (same (add1 j))
      (step-add1+=+add1 j))))
```

Because of frame 35, it is *true* that, for all Nats n and j,

(add1 (+ *n j*))

equals

(+ *n* (add1 *j*)).

[36] Right.

This also means that

(add1 (+ *n-1 n-1*))

equals

(+ *n-1* (add1 *n-1*))

because n and j can both be *n-1*.

What expression has the type

```
(= Nat
  (add1 (+ n-1 n-1))
  (+ n-1 (add1 n-1)))?
```

[37] The expression

(***add1+=+add1*** *n-1 n-1*)

is an

```
(= Nat
  (add1 (+ n-1 n-1))
  (+ n-1 (add1 n-1))).
```

Now, use the fact that

(+ *n-1* (add1 *n-1*))

equals

(add1 (+ *n-1 n-1*))

to prove ***twice=double***.

[38] The statement in frame 24 suggests an **ind-Nat**-expression.

```
(define twice=double
  (λ (n)
    (ind-Nat n
      mot-twice=double
      (same zero)
      step-twice=double)))
```

What is ***mot-twice=double***?

[39] It follows the usual approach of abstracting over the target.

```
(claim mot-twice=double
  (→ Nat
    𝒰))
(define mot-twice=double
  (λ (k)
    (= Nat
      (twice k)
      (double k))))
```

What about ***step-twice=double***?

[40] ***step-twice=double***'s type is built the same way as for every other step.

```
(claim step-twice=double
  (Π ((n-1 Nat))
    (→ (mot-twice=double n-1)
      (mot-twice=double (add1 n-1)))))
```

Here's the beginning of the definition.

```
(define step-twice=double
  (λ (n-1)
    (λ (twice=double_n-1)
      [          ])))
```

What is $twice{=}double_{n\text{-}1}$'s type?

[41] $twice{=}double_{n\text{-}1}$ is an

```
(= Nat
  (twice n-1)
  (double n-1)).
```

The box's type is

```
(= Nat
  (twice (add1 n-1))
  (double (add1 n-1))),
```

and that type and

```
(= Nat
  (add1
    (+ n-1 (add1 n-1)))
  (add1
    (add1 (double n-1))))
```

are the same type.

[42] Frame 24 explains why

```
(double (add1 n-1))
```

is the same Nat as

```
(add1
  (add1 (double n-1))).
```

Why is

```
(twice (add1 n-1))
```

the same Nat as

```
(add1
  (+ n-1 (add1 n-1)))?
```

An observation about + comes in handy. No matter which Nats j and k are,

```
1. (+ (add1 j) k)
2. (iter-Nat (add1 j)
     k
     step-+)
3. (step-+
     (iter-Nat j
       k
       step-+))
4. (add1
     (iter-Nat j
       k
       step-+))
5. (add1
     (+ j k)).
```

[43] This is very much like the observation about *incr* on page 189.

Observation about +

No matter which Nats j **and** k **are,**

(+ (add1 j) k)

is the same Nat **as**

(add1
(+ j k)).

Using this observation about +,

1. (*twice* (add1 *n-1*))
2. (+ (add1 *n-1*) (add1 *n-1*))
3. (add1
(+ *n-1* (add1 *n-1*)))

Can **cong** do the job?

[44] The expression

(**cong** *twice=double*$_{n\text{-}1}$
(+ 2))

is an

(= Nat
(add1
(add1 (+ *n-1* *n-1*)))
(add1
(add1 (*double* *n-1*)))),

which is not the same type.

It is not the same type, but it is *nearly* the same type.

[45] Replacing

(add1 (+ *n-1* *n-1*))

with

(+ *n-1* (add1 *n-1*)),

in the type would do the trick.

Because

(add1 (+ *n-1* *n-1*))

equals

(+ *n-1* (add1 *n-1*)),

replace can move the add1 from the second argument of + to the outside.

[46] Right, because **replace** is used when the type of something *nearly* fits, and the part that doesn't is equal to something that would make it fit.

In this case, which part of the type of

```
(cong twice=double_{n-1}
  (+ 2))
```

fits?

[47] Everything but this gray box fits just fine.

```
(= Nat
  (add1
    [          ])
  (add1
    (add1 (double n-1))))
```

Now define the motive.

mot-step-twice=double needs an extra argument, just like *step-∗*.

```
(claim mot-step-twice=double
  (→ Nat Nat
    𝒰))
```

[48] The empty box becomes a λ-expression's variable.

```
(define mot-step-twice=double
  (λ (n-1 k)
    (= Nat
      (add1
        k)
      (add1
        (add1 (double n-1))))))
```

What is the target of the **replace**-expression?

[49] The expression

```
(add1 (+ n-1 n-1))
```

should be replaced by

```
(+ n-1 (add1 n-1)),
```

so the target should be

```
(add1+=+add1 n-1 n-1).
```

Here is the definition so far.

```
(define step-twice=double
  (λ (n-1)
    (λ (twice=double_{n-1})
      (replace (add1+=+add1 n-1 n-1)
        (mot-step-twice=double n-1)
        [          ]))))
```

[50] The base is the expression whose type is nearly right, which is

```
(cong twice=double_{n-1}
  (+ 2)).
```

What is the complete definition of ***step-twice=double***?

[51] The function that is one of the arguments to **cong** is (**+** 2).

```
(define step-twice=double
  (λ (n-1)
    (λ (twice=double_n-1)
      (replace (add1+=+add1 n-1 n-1)
        (mot-step-twice=double n-1)
        (cong twice=double_n-1
          (+ 2))))))
```

And, finally, ***twice=double*** deserves a solid box.

[52] So far, the type of each **replace**-expression has **=** at the top.

```
(define twice=double
  (λ (n)
    (ind-Nat n
      mot-twice=double
      (same zero)
      step-twice=double)))
```

Good point. **replace** is useful because by writing an appropriate motive, it can have any type.

Find two proofs that,

"(***twice*** 17) equals (***double*** 17)."

```
(claim twice=double-of-17
  (= Nat (twice 17) (double 17)))
(claim twice=double-of-17-again
  (= Nat (twice 17) (double 17)))
```

[53] If a statement is true for every Nat, then it is true for 17. One way to prove it is to apply ***twice=double*** to 17.

```
(define twice=double-of-17
  (twice=double 17))
```

This is similar to ***twin-Atom*** in frame 4:54.

What's the other proof?

[54] (*twice* 17) is already the same Nat as (*double* 17), so same can also be used.

```
(define twice=double-of-17-again
  (same 34))
```

In fact, (same 34) is even the value of *twice=double-of-17*.

Define a function called *twice-Vec* that duplicates each entry in a Vec. For example, the normal form of

```
(twice-Vec Atom 3
  (vec:: 'chocolate-chip
    (vec:: 'oatmeal-raisin
      (vec:: 'vanilla-wafer
        vecnil))))
```

is

```
(vec:: 'chocolate-chip
  (vec:: 'chocolate-chip
    (vec:: 'oatmeal-raisin
      (vec:: 'oatmeal-raisin
        (vec:: 'vanilla-wafer
          (vec:: 'vanilla-wafer
            vecnil)))))).
```

[55] What should the type be?

As the name suggests, the function makes a Vec with *twice* as many entries.

```
(claim twice-Vec
  (Π ((E U)
      (ℓ Nat))
    (→ (Vec E ℓ)
      (Vec E (twice ℓ)))))
```

[56] This sounds difficult.

Why is that?

[57] Because the type depends on a Nat, the function suggests using **ind-Nat** with a step that uses vec:: twice.

To use vec::, the desired length must have add1 on top. The length of this Vec, however, will have only one add1 on top.

Why is there only a single add1 at the top of the length?

[58] Based on observation about **+** from page 210,

(***twice*** (add1 *n-1*))

is the same

Nat

as

(add1 (**+** *n-1* (add1 *n-1*))).

Here's a more direct way to state the problem.

```
(claim double-Vec
  (Π ((E U)
      (ℓ Nat))
    (→ (Vec E ℓ)
      (Vec E (double ℓ)))))
```

[59] That is easier to define with **ind-Nat**. Here's the base.

```
(claim base-double-Vec
  (Π ((E U))
    (→ (Vec E zero)
      (Vec E (double zero)))))
(define base-double-Vec
  (λ (E)
    (λ (es)
      vecnil)))
```

That's right—doubling an empty Vec is still empty. What about the motive?

```
(claim mot-double-Vec
  (→ U Nat
    U))
```

[60] It can be found by abstracting over zero in the base's type.

```
(define mot-double-Vec
  (λ (E k)
    (→ (Vec E k)
      (Vec E (double k)))))
```

How about the step?

```
(claim step-double-Vec
  (Π ((E U)
      (ℓ-1 Nat))
    (→ (→ (Vec E ℓ-1)
          (Vec E (double ℓ-1)))
       (→ (Vec E (add1 ℓ-1))
          (Vec E
            (double (add1 ℓ-1)))))))
```

[61] The step transforms a doubler for a Vec with *ℓ-1* entries into a doubler for a Vec with (add1 *ℓ-1*) entries. And

```
(double
  (add1 ℓ-1))
```

is the same Nat as

```
(add1
  (add1
    (double ℓ-1)))
```

so the two uses of vec:: are expected.

```
(define step-double-Vec
  (λ (E ℓ-1)
    (λ (double-Vec_ℓ-1)
      (λ (es)
        (vec:: (head es)
          (vec:: (head es)
            (double-Vec_ℓ-1
              (tail es))))))))
```

What is the definition of *double-Vec*?

[62] All of its parts are defined, so it deserves a solid box.

```
(define double-Vec
  (λ (E ℓ)
    (ind-Nat ℓ
      (mot-double-Vec E)
      (base-double-Vec E)
      (step-double-Vec E))))
```

Even though it is true that (*double* *n*) equals (*twice* *n*) for all Nats *n*, it is not equally easy to define dependent functions that use them. *double-Vec* is easy, while *twice-Vec* is not.

[63] That's right.

The proof that (***double*** n) equals (***twice*** n) for all Nats n can be used to define ***twice-Vec*** using ***double-Vec***.

[64] That certainly saves a lot of effort.

Solve Easy Problems First

If two functions produce equal results, then use the easier one when defining a dependent function, and then use replace to give it the desired type.

The type of

(***double-Vec*** E ℓ *es*)

is

(Vec E (***double*** ℓ)).

The (***double*** ℓ) needs to become (***twice*** ℓ).

What is the target?

```
(define twice-Vec
  (λ (E ℓ)
    (λ (es)
      (replace [          ]
        (λ (k)
          (Vec E k))
        (double-Vec E ℓ es)))))
```

[65] What about (***twice=double*** ℓ)?

That's very close, but

(***twice=double*** ℓ)

is an

(= Nat (***twice*** ℓ) (***double*** ℓ)),

which has the TO and the FROM in the wrong order.

[66] Does this mean that we need to prove ***double=twice*** now?

Luckily, that's not necessary. Another special eliminator for **=**, called **symm**[†], fixes this problem.

If *target* is an
  (**=** *X from to*),
then
  (**symm** *target*)
is an
  (**=** *X to from*).

[†]Short for "symmetry."

[67] Okay, it's possible to define ***twice-Vec***.

```
(define twice-Vec
  (λ (E ℓ)
    (λ (es)
      (replace (symm
                  (twice=double ℓ))
        (λ (k)
          (Vec E k))
        (double-Vec E ℓ es)))))
```

That's right.

[68] Whew!

The Law of symm

If e **is an (=** X *from to*$)$**, then (symm** e**) is an (=** X *to from*$)$**.**

The Commandment of symm

If x **is an** X**, then**
  (**symm** (same x))
is the same
  (**=** X x x)
as
  (same x).

Now go eat all the cookies you can find,

and dust off your lists.

10

It Also Depends on the List

Before we get started, here are three more expectations. Have you ...

1. figured out why we need induction,
2. understood **ind-Nat**, and
3. built a function with induction?

[1] More expectations! Here are all the expectations from frame 5:2, together with these three new expectations. The expectations are to have

- cooked ratatouille,
- eaten two pieces of cherry pie,
- tried to clean up with a non-napkin,
- understood **rec-Nat**, and
- slept until well-rested; as well as

1. figured out why we need induction,
2. understood **ind-Nat**, and
3. built a function with induction.

It seems that these lists are mismatched. The lists from chapter 5 don't have obvious lengths, while these lists do.

```
(claim more-expectations
  (Vec Atom 3))
(define more-expectations
  (vec:: 'need-induction
    (vec:: 'understood-induction
      (vec:: 'built-function vecnil))))
```

[2] But ***append*** can't mix a List and a Vec.

No, it can't. That is a job for ***vec-append***, which is not yet defined. To use ***vec-append*** on a List, we must transform it into a Vec.

[3] But to build a Vec, don't we need a number of entries?

There is another possibility.

Previous definitions that used Vec accepted the number of entries as arguments. But with a new twist on an old type, it is possible to build the Vec and its length together.

[4] What is that new twist?

What does it mean for a value to be a
(Pair *A* *D*)?

[5] A value is a (Pair *A* *D*) if

1. it has cons at the top,
2. its **car** is an *A*, and
3. its **cdr** is a *D*.

If
(cons *a* *d*)
is a
(Σ † ((*x* *A*))
D),
then *a*'s type is *A* and *d*'s type is found by consistently replacing every *x* in *D* with *a*.

[6] When is
(Σ ((*x* *A*))
D)
a type?

---

† Σ is pronounced "sigma;" also written `Sigma`.

The expression

(Σ ((*x* *A*))
  D)

is a type when

1. *A* is a type, and
2. *D* is a type if *x* is an *A*.†

Is

(Σ ((*bread* Atom))
  (= Atom *bread* 'bagel))

a type?

†Another way to say this is "*D* is a family of types over *A*." This terminology is also used for the body of a Π-expression.

---

[7] Yes, because Atom is a type, and

(= Atom *bread* 'bagel)

is a type when *bread* is an Atom.

What expression has the type

(Σ ((*bread* Atom))
  (= Atom *bread* 'bagel))?

[8] How about (cons 'bagel (same 'bagel))?

Indeed.

Is

(Σ ((*A* *U*))
  A)

a type?

[9] *U* is a type, and *A* is certainly a type when *A* is a *U*.

The Law of Σ

The expression

$(\Sigma\ ((x\ A))$
$\quad D)$

is a type when A is a type, and D is a type if x is an A.

The Commandment of cons

If p is a

$(\Sigma\ ((x\ A))$
$\quad D)$,

then p is the same as

$(\mathsf{cons}\ (\mathsf{car}\ p)\ (\mathsf{cdr}\ p))$.

Name three expressions that have that type.

[10] Nat is a $\mathcal{U}$ and 4 is a Nat, so

```
(cons Nat 4)
```

is a

```
(Σ ((A U))
  A).
```

Two more expressions with that type are

```
(cons Atom 'porridge),
```

and

```
(cons (→ Nat
         Nat)
  (+ 7)).
```

Is

```
(cons 'toast
  (same (:: 'toast nil)))
```

a

```
(Σ ((food Atom))
  (= (List Atom)
    (:: food nil)
    (:: 'toast nil)))?
```

[11] Yes, it is, because consistently replacing *food* with 'toast in

```
(= (List Atom)
  (:: food nil)
  (:: 'toast nil))
```

is

```
(= (List Atom)
  (:: 'toast nil)
  (:: 'toast nil)),
```

so (same (:: 'toast nil)) is acceptable.

What is the relationship between Σ and Pair?

(Pair A D) is a short way of writing

```
(Σ ((x A))
  D)
```

where x is not used in D.

[12] This is similar to how some Π-expressions can be written as →-expressions, from frame 6:40.

How can Σ combine a number of entries with a Vec?

Like this:

```
(Σ ((ℓ Nat))
  (Vec Atom ℓ)).
```

[13] What values have that type?

Here are seventeen 'peas:

```
(cons 17 (peas 17)).
```

Now give another.

[14] How about a nice breakfast?

```
(cons 2
  (vec:: 'toast-and-jam
    (vec:: 'tea vecnil)))
```

It's good to start the day off right.

Types built with $\rightarrow$, Π, and = can be read as statements, and expressions of those types are proofs. Similarly, types built with Pair and Σ can be read as statements.

A (Pair A D) consists of both evidence for A and evidence for D, with cons at the top. This means that (Pair A D) can be read

"A and D"

because to give evidence for an "and" is to give evidence for both parts.

How can

```
(Pair (= Nat 2 3)
  (= Atom 'apple 'apple))
```

be read as a statement?

Evidence for

```
(Σ ((x A))
  D)
```

is a pair whose **car** is an A and whose **cdr** is evidence for the statement found by consistently replacing each x in D with the **car**.

---

[15] How can (Pair A D) be read as a statement?

---

[16] It is the statement

"2 equals 3 and 'apple equals 'apple."

There is no evidence for this statement, because there is no evidence for

"2 equals 3."

and thus nothing to put in the **car**.

---

[17] What does that mean for Σ's reading as a statement?

A Σ-expression can be read as

"there exists."

For example,

```
(Σ ((es (List Atom)))
  (= (List Atom)
    es
    (reverse Atom es)))
```

can be read as

"There exists a list of atoms that is equal to itself reversed."

[18] Is that statement even true?

Here's a proof: (cons nil (same nil)).

[19] Of course, because reversing the empty list is the empty list.

Are there any other proofs?

[20] Yes, many lists are equal forwards and backwards.† Here is another proof:

```
(cons (:: 'bialy
          (:: 'schmear
            (:: 'bialy nil)))
  (same (:: 'bialy
            (:: 'schmear
              (:: 'bialy nil))))).
```

†These lists are called *palindromes*.

How can this expression be read as a statement?

```
(Σ ((es (List Atom)))
  (= (List Atom)
    (snoc Atom es 'grape)
    (:: 'grape es))).
```

[21] "There exists a list of atoms such that adding 'grape to the back or the front does the same thing."

Now prove it.

[22] Adding 'grape to the back or front of nil does the same thing:

```
(cons nil
  (same (:: 'grape nil))).
```

That's a proof.

Is there any other proof?

[23] Any list of only 'grapes works.

Here's another one:

```
(cons (:: 'grape
          (:: 'grape
            (:: 'grape nil)))
  (same (:: 'grape
            (:: 'grape
              (:: 'grape
                (:: 'grape nil))))))).
```

There's no way to tell one 'grape from another, so front or back does not matter.

Great job.

What is the type of a function that transforms a List into a Vec?

[24] Won't ***list→vec***'s type need to use Σ?

```
(claim list→vec
  (Π ((E 𝒰))
    (→ (List E)
      (Σ ((ℓ Nat))
        (Vec E ℓ)))))
```

That's correct, at least for now.

Here is part of the definition. What goes in the box?

```
(define list→vec
  (λ (E)
    (λ (es)
      [                    ])))
```

[25] The expression in the box must check whether *es* is nil or has :: at the top. **rec-List** does that, and the target is *es*.

That's correct.

What is the base?

[26] The base is the value when *es* is nil. That should clearly be vecnil, and vecnil has 0 entries.

```
(define list→vec
  (λ (E es)
    (rec-List es
      (cons 0 vecnil)
      [          ])))
```

Why is

```
(cons 0 vecnil)
```

a

```
(Σ ((ℓ Nat))
  (Vec E ℓ))?
```

[27] Because the **car** is a Nat, specifically 0, and the **cdr** is a (Vec *E* 0).

---

step-list→vec adds one entry to a

```
(Σ ((ℓ Nat))
  (Vec E ℓ)).
```

What is the longer Vec's type?

[28] How about

```
(Σ ((ℓ Nat))
  (Vec E (add1 ℓ))),
```

because the Vec is one entry longer?

A better type is

```
(Σ ((ℓ Nat))
  (Vec E ℓ))
```

because the point of using Σ is to have a pair whose **car** is the entire length of the **cdr**. Making the **car** larger does not change the type.

Define the step.

[29] The type follows the usual approach for **rec-List**.

```
(claim step-list→vec
  (Π ((E 𝒰))
    (→ E (List E) (Σ ((ℓ Nat))
                    (Vec E ℓ))
      (Σ ((ℓ Nat))
        (Vec E ℓ)))))
```

To define ***step-list→vec***, an eliminator for Σ is needed. Do **car** and **cdr** eliminate Σ, too?

Yes. If p is a

$(\Sigma\ ((x\ A))$
$\quad D)$,

then (**car** p) is an A.

[30] That is just like (Pair A D).

But **cdr** is slightly different.

If p is a

$(\Sigma\ ((x\ A))$
$\quad D)$,

then (**cdr** p)'s type is D where every x has been consistently replaced with (**car** p).

[31] If there is no x in D, then isn't this the way Pair from chapter 1 works?

Indeed.

If p is a

$(\Sigma\ ((\ell\ \mathsf{Nat}))$
$\quad (\mathsf{Vec}\ \mathsf{Atom}\ \ell))$,

then what is (**car** p)'s type?

[32] (**car** p) is a Nat.

If p is a

$(\Sigma\ ((\ell\ \mathsf{Nat}))$
$\quad (\mathsf{Vec}\ \mathsf{Atom}\ \ell))$,

then what is (**cdr** p)'s type?

[33] (**cdr** p) is a (Vec Atom (**car** p)).

So Σ is another way to construct a dependent type.

Here is ***step-list→vec***.

```
(define step-list→vec
  (λ (E)
    (λ (e es list→vec_es)
      (cons (add1 (car list→vec_es))
        (vec:: e (cdr list→vec_es))))))
```

Please explain it.

[34] Here goes.

1. The body of the inner λ-expression has cons at the top because it must construct a Σ.

2. The **car** of the inner λ-expression's body is
   (add1 (**car** *list→vec*$_{es}$))
   because ***step-list→vec*** builds a Vec with one more entry than
   (**cdr** *list→vec*$_{es}$).

3. The **cdr** of the inner λ-expression's body has one more entry than the **cdr** of *list→vec*$_{es}$, namely *e*. vec:: adds this new entry.

Now, give a complete definition of ***list→vec***.

[35] The box is filled with (***step-list→vec*** *E*).

```
(define list→vec
  (λ (E)
    (λ (es)
      (rec-List es
        (cons 0 vecnil)
        (step-list→vec E)))))
```

How might this version of ***list→vec*** be summarized?

[36] This ***list→vec*** converts a list into a pair where the **car** is the length of the list and the **cdr** is a Vec with that many entries.

For nil, the length is 0 and the Vec is vecnil. For ::, the length is one greater than the length of the converted rest of the list, and vec:: adds the same entry that :: added.

What is the value of

```
(list→vec Atom
  (:: 'beans
    (:: 'tomato nil)))?
```

[37] Let's see.

1.
```
(list→vec Atom
  (:: 'beans
    (:: 'tomato nil)))
```
2.
```
(rec-List (:: 'beans
            (:: 'tomato nil))
  (cons 0 vecnil)
  (step-list→vec Atom))
```
3.
```
(step-list→vec Atom
  'beans
  (:: 'tomato nil)
  (rec-List (:: 'tomato nil)
    (cons 0 vecnil)
    (step-list→vec Atom)))
```
4.
```
(cons
  (add1
    (car
      (rec-List (:: 'tomato nil)
        (cons 0 vecnil)
        (step-list→vec Atom))))
  (vec:: 'beans
    (cdr
      (rec-List (:: 'tomato nil)
        (cons 0 vecnil)
        (step-list→vec Atom)))))
```

What is the normal form? The "same-as" chart can be skipped.

[38] The normal form is

```
(cons 2
  (vec:: 'beans
    (vec:: 'tomato vecnil))).
```

The definition of *list→vec* is in a dashed box.

Why?

[39] That means that there is something the matter with it?

The type given for ***list→vec*** is not specific enough.

The whole point of Vec is to keep track of how many entries are in a list, but wrapping it in a Σ hides this information. In chapter 7, specific types were used to make functions total. But specific types can also rule out foolish definitions.

[40] But this definition is correct, isn't it? The starting expression

```
(:: 'beans
  (:: 'tomato nil))
```

appears to be the expected normal form. Here it is with its length:

```
(cons 2
  (vec:: 'beans
    (vec:: 'tomato vecnil))).
```

Use a Specific Type for Correctness

Specific types can rule out foolish definitions.

Here is a foolish definition that the type of ***list→vec*** permits.

```
(define list→vec
  (λ (E)
    (λ (es)
      (cons 0 vecnil))))
```

[41] Applying this ***list→vec*** to any type and any list yields (cons 0 vecnil).

That's correct.

What might another incorrect, yet still type-correct, definition be?

[42] ***list→vec*** could be a function that always produces a Vec with 52 entries.

Almost.

Can it produce 52 entries, each of which has type E, when *es* is nil?

[43] We don't know ahead of time which $\mathcal{U}$ is to be the E that is the argument to the λ-expression. So there is no way to find an entry with that type when *es* is nil.

list→vec could be a function that produces a Vec with 52 entries when *es* has :: at the top, or 0 entries when *es* is nil, right?

Yes, it could.

Writing vec:: 52 times would be tiring, though.

[44] A definition similar to ***peas*** would help with that.

Good idea. Call it ***replicate***. Just as with ***peas***, the definition of ***replicate*** requires the use of **ind-Nat**.

Why?

[45] The definition of ***replicate*** requires the use of **ind-Nat** because, in ***replicate***'s type, the Nat ℓ is the target.

```
(claim replicate
  (Π ((E U)
      (ℓ Nat))
    (→ E
      (Vec E ℓ))))
```

The body of the Π-expression *depends on* ℓ, and **ind-Nat** is used when a type depends on the target.

Even though it is now time for breakfast, chapter 7 was not spent in vain!

What is the base?

[46] The base is a

(Vec E 0),

so it must be

vecnil.

Here is ***mot-replicate***'s type.

```
(claim mot-replicate
  (→ 𝒰 Nat
    𝒰))
```

Now define ***mot-replicate***.

[47] The definition of ***mot-replicate*** follows a familiar approach, abstracting over zero as in frame 7:66.

```
(define mot-replicate
  (λ (E k)
    (Vec E k)))
```

The next step is to define ***step-replicate***.

[48] At each step, ***step-replicate*** should add an entry to the list.

Where does that entry come from?

Just as E is an argument to ***mot-replicate***, both E and e are arguments to ***step-replicate***.

This is similar to the way ***step-∗*** is applied to j in frame 3:66.

[49] Here is ***step-replicate***'s definition.

```
(claim step-replicate
  (Π ((E 𝒰)
      (e E)
      (ℓ-1 Nat))
    (→ (mot-replicate E ℓ-1)
      (mot-replicate E (add1 ℓ-1)))))
(define step-replicate
  (λ (E e ℓ-1)
    (λ (step-replicate_ℓ-1)
      (vec:: e step-replicate_ℓ-1))))
```

Now define ***replicate*** using the motive, the base, and the step.

[50] The components are all available.

```
(define replicate
  (λ (E ℓ)
    (λ (e)
      (ind-Nat ℓ
        (mot-replicate E)
        vecnil
        (step-replicate E e)))))
```

In frame 49, ***mot-replicate*** is applied to two arguments, but here, it is applied to one. Also, ***step-replicate*** is applied to four arguments, but here, it is applied to only two.

Why?

[51] Every motive for **ind-Nat** has type

(→ Nat
  $\mathcal{U}$).

Because of Currying, (***mot-replicate*** *E*) has that type.

Similarly, every step for **ind-Nat** is applied to two arguments. Because of Currying, applying the first two arguments to the four-argument ***step-replicate*** produces the expected two-argument function.

---

replicate is intended to help write an alternative definition of ***list→vec*** that produces a Vec with 52 entries when *es* has :: at the top, or 0 entries when *es* is nil.

[52] Here, cons in the definition of ***copy-52-times*** is the constructor of Σ, used to associate the length with the Vec.

```
(claim copy-52-times
  (Π ((E U))
    (→ E
       (List E)
       (Σ ((ℓ Nat))
         (Vec E ℓ))
       (Σ ((ℓ Nat))
         (Vec E ℓ)))))
(define copy-52-times
  (λ (E)
    (λ (e es copy-52-times_es)
      (cons 52 (replicate E 52 e)))))

(define list→vec
  (λ (E)
    (λ (es)
      (rec-List es
        (cons 0 vecnil)
        (copy-52-times E)))))
```

The type can be made more specific by making clear the relationship between the List and the number of entries in the Vec.

What is that relationship?

[53] The number of entries in the Vec is the length of the List.

Exactly. Here is a more specific type.

```
(claim list→vec
  (Π ((E U)
      (es (List E)))
    (Vec E (length E es))))
```

[54] How can ***list→vec*** be defined?

Some of it should be predictable.

[55] Yes, the type of ***list→vec*** predicts some of ***list→vec***'s definition.

```
(define list→vec
  (λ (E es)
    ... but what goes here? ))
```

What is the type of the box?

[56] The type of the box is the body of the Π-expression in the type of ***list→vec***, which is

(Vec *E* (***length*** *E es*)).

If *es* were a Nat, then **ind-Nat** would work. But *es* is a (List *E*).

Is there an **ind-List**?

Good thinking.

ind-Nat requires one more argument than **rec-Nat**, the motive.

[57] Does **ind-List** also need a motive?

---

ind-List requires one more argument than **rec-List**, and this argument is also a motive:

(**ind-List** *target*
mot
base
step).

[58] What is that expression's type?

First, *target* is a (List E).

[59] Of course.

Otherwise, **ind-List** would not be induction on List.

Just as in **ind-Nat**, *mot* explains the reason for doing induction. In other words, it explains the manner in which the type of the **ind-List**-expression depends on *target*.

What type should *mot* have?

[60] *mot* finds a type when applied to a list, so it is an

($\rightarrow$ (List E)
$\mathcal{U}$).

What type should *base* have?

[61] *base* is a (*mot* nil) because nil plays the same role as zero.

The constructor :: plays a role similar to add1, except :: has two arguments: an entry and a list.

[62] Does the step for **ind-List** have a type that is similar to the step for **ind-Nat**?

Just like the step for **ind-Nat** transforms an almost-answer for n into an answer for (add1 n), the step for **ind-List** takes an almost-answer for some list *es* and constructs an answer for (:: *e es*).

step's type is

```
(Π ((e E)
    (es (List E)))
  (→ (mot es)
    (mot (:: e es)))).
```

[63] Here, adding an entry *e* to *es* with :: is like adding one with add1 in **ind-Nat**.

The Law of ind-List

If *target* **is a** (List E)**,**
mot **is an**

```
  (→ (List E)
    U),
```

base **is a** (*mot* nil)**, and** *step* **is a**

```
  (Π ((e E)
      (es (List E)))
    (→ (mot es)
      (mot (:: e es))))
```

then

```
  (ind-List target
    mot
    base
    step)
```

is a (*mot target*)**.**

The First Commandment of ind-List

The ind-List-expression

(ind-List nil
mot
base
step)

is the same (*mot* nil) **as** *base*.

The Second Commandment of ind-List

The ind-List-expression

(ind-List (:: *e es*)
mot
base
step)

is the same (*mot* (:: *e es*)) **as**

(*step e es*
(ind-List *es*
mot
base
step)).

Nat and List are closely related.

[64] As expected.

Thus, an ind-List-expression's type is
(*mot target*).

The box in frame 55 should be filled by an **ind-List**-expression.

[65] The target is *es*.

```
(define list→vec
  (λ (E es)
    (ind-List es
      mot-list→vec
      base-list→vec
      step-list→vec)))
```

Could → have been used to write the Π-expression in the type of ***list→vec*** in frame 54?

[66] No, because the type

(Vec *E* (***length*** *E es*))

depends on both *E* and *es*.

What is ***base-list→vec***'s type?

[67] When *es* is nil,

(Vec *E* (***length*** *E es*))

and

(Vec *E* 0) are the same type.

What is the base, then?

[68] The only (Vec *E* 0) is vecnil, so there is no point in defining ***base-list→vec***.

```
(define list→vec
  (λ (E es)
    (ind-List es
      mot-list→vec
      vecnil
      step-list→vec)))
```

Now, working backwards from the type of the base, what is the motive?

[69] Abstracting over the zero in the base does not *immediately* work because the argument to the motive is a (List E), not a Nat.

But ***length*** transforms Lists into Nats, and appears in the body of the Π-expression in ***list→vec***'s type in frame 54.

That is well-spotted. Abstracting over constants often works, but in this case, it requires a little fine-tuning with ***length***.

Here is ***mot-list→vec***'s type.

```
(claim mot-list→vec
  (Π ((E U))
    (→ (List E)
      U)))
```

Now define ***mot-list→vec***.

[70] Here is the definition of ***mot-list→vec***.

```
(define mot-list→vec
  (λ (E)
    (λ (es)
      (Vec E (length E es)))))
```

For example, the value of

(***mot-list→vec*** Atom nil)

is

(Vec Atom 0),

as expected.

What is ***step-list→vec***'s type?

[71] No surprises here.

```
(claim step-list→vec
  (Π ((E U)
      (e E)
      (es (List E)))
    (→ (mot-list→vec E es)
      (mot-list→vec E (:: e es)))))
```

Now define ***step-list→vec***.

[72] Here it is.

```
(define step-list→vec
  (λ (E e es)
    (λ (list→vec_es)
      (vec:: e list→vec_es))))
```

What is the almost-answer *list→vec*$_{es}$'s type?

[73] It is

(***mot-list→vec*** *E es*).

Also,

(***mot-list→vec*** *E es*)

and

(Vec *E* (***length*** *E es*))

are the same type.

(***length*** *E es*) is a Nat, even though it is neither zero nor does it have add1 at the top.

[74] The normal form of (***length*** *E es*) must be neutral because the target of **rec-List** in ***length*** is *es*, which is a variable.

What is the type of

(vec:: *e list→vec*$_{es}$)?

[75] *list→vec*$_{es}$'s type is

(Vec *E* (***length*** *E es*))

so the type of

(vec:: *e list→vec*$_{es}$)

is

(Vec *E* (add1 (***length*** *E es*))).

Why are

(Vec *E* (add1 (***length*** *E es*)))

and

(***mot-list→vec*** *E* (:: *e es*))

the same type?

[76] Because all these expressions are the same type.

1. (***mot-list→vec*** *E* (:: *e es*))
2. (Vec *E* (***length*** *E* (:: *e es*)))
3. (Vec *E* (add1 (***length*** *E es*)))

Now define ***list→vec***.

77 ***list→vec*** finally deserves a solid box.

```
(define list→vec
  (λ (E es)
    (ind-List es
      (mot-list→vec E)
      vecnil
      (step-list→vec E))))
```

This more specific type rules out our two foolish definitions.

Unfortunately, there are still foolish definitions that have this type.

78 Oh no!

What is the first foolish definition that the new type rules out?

79 The first foolish definition, in frame 41, always produces

(cons 0 vecnil).

What is the other?

80 The foolish definition in frame 52 makes 52 copies of the first entry in the list. The new type demands the correct length, so it rules out this foolish definition.

What other foolishness is possible?

Here is a possible, yet foolish, step. Would the definition of ***list→vec*** need to be different to use this step?

```
(define step-list→vec
  (λ (E e es)
    (λ (list→vec_es)
      (replicate E (length E (:: e es))
        e))))
```

81 No, the same definition would work.

```
(define list→vec
  (λ (E es)
    (ind-List es
      (mot-list→vec E)
      vecnil
      (step-list→vec E))))
```

Using this foolish definition, what is the normal form of

```
(list→vec Atom
  (:: 'bowl-of-porridge
    (:: 'banana
      (:: 'nuts nil))))?
```

[82] The name *list→vec*$_{es}$ is dim, so the definition is not actually recursive.

The normal form is three bowls of porridge,

```
(vec:: 'bowl-of-porridge
  (vec:: 'bowl-of-porridge
    (vec:: 'bowl-of-porridge vecnil))).
```

The first is too hot, the second is too cold, but the third is just right.†

Nevertheless, the definition is foolish—'banana and 'nuts make a breakfast more nutritious.

†Thank you, Robert Southey (1774–1843).

[83] Is there an even more specific type that rules out all of the foolish definitions?

Yes, there is.

[84] And what about appending Vecs?

Coming right up! But finish your breakfast first—you need energy for what's next.

[85] Can't wait!

Go have toast with jam and a cup of tea.

Also, just one bowl of porridge with a banana and nuts.

11

All Lists Are Created Equal

After all that porridge, it's time for an afternoon coffee break with Swedish treats!

[1] Yes! *Fika.*

Here is a list of treats for our *fika.*

```
(claim treats
  (Vec Atom 3))
(define treats†
  (vec:: 'kanelbullar
    (vec:: 'plättar
      (vec:: 'prinsesstårta vecnil))))
```

†*Kanelbullar* are cinnamon rolls, *plättar* are small pancakes topped with berries, and a *prinsesstårta* is a cake with layers of sponge cake, jam, and custard under a green marzipan surface.

[2] Sounds great! But how can **treats** be combined with **drinks**?

```
(claim drinks
  (List Atom))
(define drinks
  (:: 'coffee
    (:: 'cocoa nil)))
```

That's right—there are some loose ends from the preceding chapter. One loose end is a version of **append** for Vec, and the other is ruling out more foolish definitions of **list→vec**.

[3] Okay.

If *es* has ℓ entries and *end* has j entries, then how many entries do they have together?

[4] Surely they have $(+\ \ell\ j)$ entries together.

That's right.

```
(claim vec-append
  (Π ((E U)
      (ℓ Nat)
      (j Nat))
    (→ (Vec E ℓ) (Vec E j)
      (Vec E (+ ℓ j)))))
```

[5] This looks very much like **append**'s type.

How does ***vec-append***'s type differ from ***append***'s type?

[6] This more specific type makes clear how many entries are in each list.

Exactly.

To define ***vec-append***, what is missing?

[7] An eliminator for Vec.

Actually, it is possible to define ***vec-append*** in the same style as ***first***, ***rest***, ***last***, and ***drop-last***, using **ind-Nat**, **head**, and **tail**.

The definition that uses **ind-Vec**, however, expresses its intent more directly.

[8] Can every operation on Vec that can be written using **ind-Nat** also be written using **head** and **tail**?

No.

In all of the definitions that can be written using **head** and **tail**, the type depends only on the length, which is a Nat. Sometimes, though, a type depends on a Vec, and then **ind-Vec** is necessary.

[9] Is **ind-Vec** like **ind-List**?

Yes, **ind-Vec** is much like **ind-List**. An **ind-Vec**-expression

(**ind-Vec** *n es*
mot
base
step)

has two targets:

1. *n*, which is a Nat,
2. and *es*, which is a (Vec *E n*).

[10] So *n* is the number of entries in *es*.

Are there any other differences between **ind-List** and **ind-Vec**?

Each part of the **ind-Vec**-expression must account for the number of entries in *es*.

mot's type is

```
(Π ((k Nat))
  (→ (Vec E k)
    U))
```

because it explains why *any* target Nat and Vec are eliminated.

[11] Why isn't E also an argument in the Π-expression?

Excellent question. This is because the type of entries in a list plays a very different role from the number of entries.

In any individual list, the type of entries is the same throughout, but the number of entries in the **tail** of a list is *different* from the number of entries in the list.

[12] Why does that matter?

The entry type E is determined once, and it is the same for the entire elimination. But the number of entries changes with each of **ind-Vec**'s steps.

How is a motive used for the type of a step?

[13] The type of a step uses the motive in the type of the almost-answer and the type of the answer.

This means that the motive is used for different numbers of entries. That is why the number of entries is an argument to the motive.

These two varieties of arguments to a type constructor, that either vary or do not vary, have special names. Those that do not vary, such as the entry type in Vec and List, are called *parameters*, and those that do vary are called *indices*.

[14] So the number of entries in a Vec is an index.

The Law of ind-Vec

If n **is a** Nat**,** *target* **is a** (Vec E n)**,** *mot* **is a**

```
(Π ((k Nat))
  (→ (Vec E k)
    𝒰))
```

base **is a** (*mot* zero vecnil)**, and** *step* **is a**

```
(Π ((k Nat)
    (h E)
    (t (Vec E k)))
  (→ (mot k t)
    (mot (add1 k) (vec:: h t))))
```

then

```
(ind-Vec n target
  mot
  base
  step)
```

is a (*mot* n *target*)**.**

Yes, it is.[†]

Whenever a type constructor has an index, the index shows up in the motive for its eliminator, and therefore also in the step.

[†]A family of types whose argument is an index is sometimes called "an indexed family."

[15] What is *base*'s type in ind-Vec?

---

base's type is

(*mot* zero vecnil).

In ind-Vec, *mot* receives two arguments, rather than one.

[16] Doesn't ***mot-replicate*** in frame 10:47 receive two arguments as well?

No, though it does appear to.

Remember that ***mot-replicate*** is Curried. Applying ***mot-replicate*** to its first argument, which is the entry type, constructs a one-argument motive to be used with **ind-Nat**.

[17] What is *step*'s type?

---

step transforms an almost-answer for some list t into an answer for (vec:: h t), so it is a

```
(Π ((k Nat)
    (h E)
    (t (Vec E k)))
  (→ (mot k t)
    (mot (add1 k) (vec:: h t)))).
```

Why is *mot* applied to (add1 k) as its first argument in the answer type?

[18] The step transforms the almost-answer for t into the answer for (vec:: h t), which has one more entry than t.

Why are the **head** and **tail** called h and t, rather than the usual e and es?

The name *es* is already taken to refer to the second target.

Now it is time to use **ind-Vec** to define ***vec-append***. Please start the definition.

[19] Just like ***append***, the base is *end*.

```
(define vec-append
  (λ (E ℓ j)
    (λ (es end)
      (ind-Vec ℓ es
        mot-vec-append
        end
        step-vec-append))))
```

Why is *end*'s type
(Vec E (+ ℓ j))?

[20] In the base, *es* is vecnil. This means that the number of entries ℓ in *es* is zero, and (+ zero j) is the same Nat as j.

1. (Vec E (+ zero j))
2. (Vec E j)

end's type is (Vec E j), which is exactly what we need.

Now define ***mot-vec-append***.

[21] The definition can be found by abstracting over the number of entries and the list in the base's type.

```
(claim mot-vec-append
  (Π ((E U)
      (k Nat)
      (j Nat))
    (→ (Vec E k)
      U)))
(define mot-vec-append
  (λ (E k j)
    (λ (es)
      (Vec E (+ k j)))))
```

With ***mot-vec-append*** in frame 21, ***vec-append*** would need a λ-expression as its motive. Why?

```
(define vec-append
  (λ (E ℓ j es end)
    (ind-Vec ℓ es
      (λ (k)
        (mot-vec-append E k j))
      end
      step-vec-append)))
```

[22] Because the two arguments to the motive are the two targets, ℓ and *es*. But the last two arguments to ***mot-vec-append*** do not match, so the λ-expression swaps *k* and *j*.

The First Commandment of ind-Vec

The ind-Vec-expression

```
(ind-Vec zero vecnil
  mot
  base
  step)
```

is the same (*mot* zero vecnil) **as** *base*.

The Second Commandment of ind-Vec

The ind-Vec-expression

```
(ind-Vec (add1 n) (vec:: e es)
  mot
  base
  step)
```

is the same `(mot (add1 n) (vec:: e es))` **as**

```
(step n e es
  (ind-Vec n es
    mot
    base
    step)).
```

Consider this definition of ***mot-vec-append***, instead.

```
(claim mot-vec-append
  (Π ((E U)
      (j Nat)
      (k Nat))
    (→ (Vec E k)
      U)))
(define mot-vec-append
  (λ (E j k)
    (λ (es)
      (Vec E (+ k j)))))
```

How does this change ***vec-append***?

[23] The λ-expression for the motive is no longer necessary.

```
(define vec-append
  (λ (E ℓ j)
    (λ (es end)
      (ind-Vec ℓ es
        (mot-vec-append E j)
        end
        step-vec-append))))
```

When writing a Curried motive, base, or step, it pays to carefully consider the order of arguments.

[24] It's certainly easier to re-order ***mot-vec-append***'s arguments than it is to write an extra λ-expression.

Now define ***step-vec-append***.

What is ***step-vec-append***'s type?

[25] This time, j is before k in the arguments.

```
(claim step-vec-append
  (Π ((E U)
      (j Nat)
      (k Nat)
      (e E)
      (es (Vec E k)))
    (→ (mot-vec-append E j
         k es)
      (mot-vec-append E j
        (add1 k) (vec:: e es)))))
```

Keen observation.

What is the definition?

[26]

```
(define step-vec-append
  (λ (E j ℓ-1 e es)
    (λ (vec-append_es)
      (vec:: e vec-append_es))))
```

This use of vec:: is justified because

(+ (add1 ℓ-1) j)

is the same Nat as

(add1 (+ ℓ-1 j)).

This relies on the observation on page 189.

All of the pieces of ***vec-append*** are ready.

[27] Here is the definition, in a well-earned solid box.

```
(define vec-append
  (λ (E ℓ j)
    (λ (es end)
      (ind-Vec ℓ es
        (mot-vec-append E j)
        end
        (step-vec-append E j)))))
```

The first loose end has been tied up.

What is a good name for

(***vec-append*** Atom 3 2 ***treats*** ***drinks***)?

[28] That expression is not described by a type because ***drinks*** is a (List Atom).

But how about ***fika*** for this version?

```
(claim fika
  (Vec Atom 5))
(define fika
  (vec-append Atom 3 2
    treats
    (list→vec Atom drinks)))
```

This ***fika*** is foolish if ***list→vec*** is foolish. In frame 10:81, a ***list→vec*** is defined that is foolish, but this foolish definition has the right type.

```
(define step-list→vec
  (λ (E e es)
    (λ (list→vec_es)
      (replicate E (length es) e))))

(define list→vec
  (λ (E es)
    (ind-List es
      mot-list→vec
      vecnil
      (step-list→vec E))))
```

[29] Using this definition, the normal form of

(***list→vec*** Atom ***drinks***)

is

```
(vec:: 'coffee
  (vec:: 'coffee vecnil)),
```

but some prefer 'cocoa to 'coffee.

How can we rule out this foolishness?

Thus far, we have used more specific types to rule out foolish definitions. Another way to rule out foolish definitions is to *prove* that they are not foolish.†

[30] What is an example of such a proof?

†Sometimes, using a more specific type is called an *intrinsic* proof. Similarly, using a separate proof is called *extrinsic*.

One way to rule out foolish definitions of ***list→vec*** is to prove that transforming the Vec back into a List results in an equal List.

This requires ***vec→list***. Here is the motive.

```
(claim mot-vec→list
  (Π ((E 𝒰)
      (ℓ Nat))
    (→ (Vec E ℓ)
      𝒰)))
(define mot-vec→list
  (λ (E ℓ)
    (λ (es)
      (List E))))
```

What is the step?

[31] The step replaces each vec:: with a :: constructor, just as ***step-list→vec*** replaces each :: with a vec:: constructor.

```
(claim step-vec→list
  (Π ((E 𝒰)
      (ℓ-1 Nat)
      (e E)
      (es (Vec E ℓ-1)))
    (→ (mot-vec→list E
         ℓ-1 es)
      (mot-vec→list E
        (add1 ℓ-1) (vec:: e es)))))
(define step-vec→list
  (λ (E ℓ-1 e es)
    (λ (vec→list_es)
      (:: e vec→list_es))))
```

The definition of ***vec→list*** is also very similar to the definition of ***list→vec***.

```
(claim vec→list
  (Π ((E 𝒰)
      (ℓ Nat))
    (→ (Vec E ℓ)
      (List E))))
(define vec→list
  (λ (E ℓ)
    (λ (es)
      (ind-Vec ℓ es
        (mot-vec→list E)
        nil
        (step-vec→list E)))))
```

What is the normal form of

(***vec→list*** Atom 3 ***treats***)?

[32] It is

```
(:: 'kanelbullar
  (:: 'plättar
    (:: 'prinsesstårta nil))).
```

So is it clear how to find the value of an **ind-Vec**-expression?

[33] Yes, it is just like finding the value of an **ind-List**-expression, except the step is applied to both targets.

How can the statement,

"For every List, transforming it into a Vec and back to a List yields a list that is equal to the starting list."

be written as a type?

[34] The term *every* implies that there should be a Π. How about this type?

```
(claim list→vec→list=
  (Π ((E 𝒰)
      (es (List E)))
    (= (List E)
      es
      (vec→list E
        (list→vec E es)))))
```

That is very close, but the second argument to ***vec→list*** is the number of entries in the Vec.

How many entries does

(***list→vec*** *E es*)

have?

[35] Oh, right, can't forget the ***length***.

```
(claim list→vec→list=
  (Π ((E 𝒰)
      (es (List E)))
    (= (List E)
      es
      (vec→list E
        (length E es)
        (list→vec E es)))))
```

What is an appropriate target for induction?

[36] The target of induction is *es*. The definition has the usual suspects: a motive, a base, and a step.

```
(define list→vec→list=
  (λ (E es)
    (ind-List es
      (mot-list→vec→list= E)
      (base-list→vec→list= E)
      (step-list→vec→list= E))))
```

What is the base?

[37] The base's type is

```
(= (List E)
  nil
  (vec→list E
    (length E nil)
    (list→vec E nil)))
```

also known as

```
(= (List E) nil nil).
```

That is the base's type.

But what is the base?

[38] (same nil), of course.

Once again, there's no need to define

base-list→vec→list=.

Here is the motive's type.

```
(claim mot-list→vec→list=
  (Π ((E U))
    (→ (List E)
      U)))
```

Define *mot-list→vec→list=*.

[39] Abstracting over nil in the base's type in frame 37 leads directly to the definition.

```
(define mot-list→vec→list=
  (λ (E es)
    (= (List E)
      es
      (vec→list E
        (length E es)
        (list→vec E es)))))
```

The only thing left is the step.

What is an appropriate type for the step?

[40] Follow the Law of ind-List.

```
(claim step-list→vec→list=
  (Π ((E U)
      (e E)
      (es (List E)))
    (→ (mot-list→vec→list= E
         es)
      (mot-list→vec→list= E
        (:: e es)))))
```

Here is the beginning of a definition.

```
(define step-list→vec→list=
  (λ (E e es)
    (λ (list→vec→list=es)
      [ list→vec→list=es ])))
```

What can be put in the box to transform the almost-proof for *es* into a proof for (:: *e es*)?

[41] The almost-proof, *list→vec→list=es*, is an

```
(= (List E)
  es
  (vec→list E
    (length E es)
    (list→vec E es))).
```

This is an opportunity to use our old friend **cong** from chapter 8 to eliminate *list→vec→list=es*.

Remember, **cong** expresses that every function produces equal values from equal arguments.

[42] Equal in, equal out!

How would we use **cong** here?

What is the type of

```
(cong (same 'plättar)
  (snoc Atom (:: 'kanelbullar nil)))?
```

[43] *snoc* does not yet have the new entry to be placed at the end of the list. Because

```
(same 'plättar)
```

is an

```
(= Atom 'plättar 'plättar),
```

and that new entry will be 'plättar, so the type is

```
(= (List Atom)
  (:: 'kanelbullar
    (:: 'plättar nil))
  (:: 'kanelbullar
    (:: 'plättar nil))).
```

Prove that

"consing 'plättar onto two equal lists of treats produces equal lists of treats."

[44] This proof can be used in the box.

First, how can the statement be written as a type?

[45] "Two equal lists of treats" can be written as a Π-expression with two (List Atom) arguments and a proof that they are equal.

```
(claim Treat-Statement
  𝒰)
(define Treat-Statement
  (Π ((some-treats (List Atom))
      (more-treats (List Atom)))
    (→ (= (List Atom)
          some-treats
          more-treats)
       (= (List Atom)
          (:: 'plättar some-treats)
          (:: 'plättar more-treats)))))
```

Proving this statement is easier with this definition.

```
(claim ::-plättar
  (→ (List Atom)
     (List Atom)))
(define ::-plättar
  (λ (tasty-treats)
    (:: 'plättar tasty-treats)))
```

Use this with **cong** to prove *Treat-Statement*.

[46] Here is the definition of *treat-proof*.

```
(claim treat-proof
  Treat-Statement)
(define treat-proof
  (λ (some-treats more-treats)
    (λ (treats=)
      (cong treats= ::-plättar))))
```

Great!

What can be said about the lengths of equal lists?

[47] Every two equal lists have equal lengths.

Now prove that

"Every two equal treat lists have equal lengths."

using **cong**.

[48] ***length-treats=*** is similar to ***treat-proof***.

```
(claim length-treats=
  (Π ((some-treats (List Atom))
      (more-treats (List Atom)))
    (→ (= (List Atom)
          some-treats
          more-treats)
       (= Nat
          (length Atom some-treats)
          (length Atom more-treats)))))
(define length-treats=
  (λ (some-treats more-treats)
    (λ (treats=)
      (cong treats= (length Atom)))))
```

Returning to the matter at hand, it is now possible to fill the box in frame 41 with a **cong**-expression.

The almost-proof, $list{\to}vec{\to}list{=}_{es}$, is an

```
(= (List E)
  es
  (vec→list E
    (length E es)
    (list→vec E es))).
```

What is the box's type in frame 41?

[49] The box's type is

```
(= (List E)
  (:: e es)
  (vec→list E
    (length E (:: e es))
    (list→vec E (:: e es)))).
```

Now it is time for an observation about ***list→vec***, similar to the observation about **+** on page 210.

What is the value of

1.
```
(vec→list E
  (length E (:: e es))
  (list→vec E (:: e es)))?
```

[50] Let's see.

2.
```
(vec→list E
  (add1 (length E es))
  (vec:: e (list→vec E es)))
```

3.
```
(:: e
  (vec→list E
    (length E es)
    (list→vec E es)))
```

When in Doubt, Evaluate

Gain insight by finding the values of expressions in types and working out examples in "same-as" charts.

How is this new observation similar to the observation about **+**?

[51] The preceding observation is that we can pull out an

```
add1
```

from **+**'s first argument and put the `add1` around the whole expression.

This new observation is that we can similarly pull out a

```
::
```

from ***list→vec***'s second argument, putting a `vec::` around the whole expression.

When using **cong**, the same function is applied to both the FROM and the TO of an **=**-expression.

What function transforms

```
es
```

into

```
(:: e es)
```

and

```
(vec→list E
  (length E es)
  (list→vec E es))
```

into

```
(:: e
  (vec→list E
    (length E es)
    (list→vec E es)))
```

?

[52] (:: *e*), right?

That is very close. But the constructor of functions is λ. Other constructors construct different types.

[53] Here is a function that does the trick.

```
(claim ::-fun
  (Π ((E 𝒰))
    (→ E (List E)
      (List E))))
(define ::-fun
  (λ (E)
    (λ (e es)
      (:: e es))))
```

Now complete the box in frame 41 to define ***step-list→vec→list=***.

[54] Here it is.

```
(define step-list→vec→list=
  (λ (E e es)
    (λ (list→vec→list=es)
      (cong list→vec→list=es
        (::-fun E e)))))
```

It's time to put the pieces together, using the motive, the base, and the step. Remember the **claim** in frame 35 on page 255.

[55] Here is another well-built solid box.

```
(define list→vec→list=
  (λ (E es)
    (ind-List es
      (mot-list→vec→list= E)
      (same nil)
      (step-list→vec→list= E))))
```

This proof rules out the foolish definition from frame 29 on page 253.

Why?

[56] Because, using the foolish definition,

```
(vec→list Atom
  (length Atom drinks)
  (list→vec Atom drinks)),
```

is not equal to ***drinks***.

Why not?

[57] Because

```
(:: 'coffee
  (:: 'coffee nil))
```

is not equal to

```
(:: 'coffee
  (:: 'cocoa nil)).
```

Where would the proof go wrong?

[58] It would go wrong in frame 54 because the new observation in frame 50 would no longer be the case.

Exactly. This proof has ruled out many foolish definitions.

[59] Many?

At some point, it becomes necessary to trust that enough specific types have been used to avoid the foolishness one might be prone to. This requires hard-won self-knowledge.

If ***vec→list*** could remove the foolishness introduced by ***list→vec***, then it would remain undetected.

[60] How could that be?

Imagine that ***vec→list*** and ***list→vec*** both reversed the order of the list.

[61] Coffee and cake are good for the imagination.

In this imaginary world, the proof would work, but both ***vec→list*** and ***list→vec*** would be foolish.

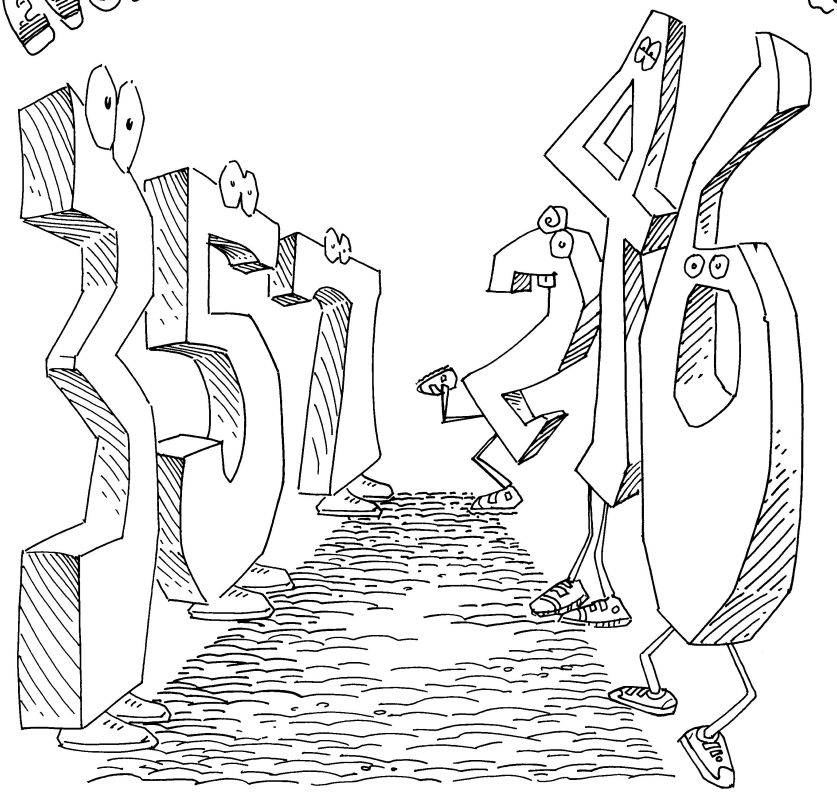
[62] If they also reversed lists, then that should have been part of their names!

Now, go and enjoy a cozy *fika*

with either an even or an odd number of friends.

12

Even Numbers Can Be Odd



What is an even number?

[1] It is a number that can be split into two equal halves.

What does it mean for a number to be split into two equal halves?

[2] It means that,

"There is some number that, added to itself, yields the original number."

How can that definition be written as a type?

[3] According to frame 10:18, a Σ-expression does the trick.

A "there is" statement has two important parts: the type of the thing that exists, and a property that it has.

Here, the type of thing that exists is Nat, and its property is being half of the even Nat. These are the respective **car** and **cdr** of the evidence for a "there is" statement.

[4] What does ***Even*** look like?

The definition of evenness can be written as a function that returns a type.

```
(claim Even
  (→ Nat
    𝒰))
(define Even
  (λ (n)
    (Σ ((half Nat))
      (= Nat n (double half)))))
```

What is the value of (***Even*** 10)?

[5] The value of

(***Even*** 10)

is

(Σ ((*half* Nat))
  (= Nat 10 (***double*** *half*))).

What are the values in (***Even*** 10)?

[6] The values look like (cons a d), where

a

is a Nat and

d

is an

(= Nat 10 (***double*** a)).

Find a and d so that

(cons a d)

is an

(***Even*** 10).

[7] a is clearly 5 because 5 is half of 10.

And

(same 10)

is an

(= Nat 10 (***double*** 5)).

This is what is needed to prove that 10 is even.

What is the proof?

[8] The proof is

```
(cons 5
  (same 10)).
```

That's right. What about 0?

[9] Half of 0 is 0.

```
(claim zero-is-even
  (Even 0))
(define zero-is-even
  (cons 0
    (same 0)))
```

What is another way that ***Even*** could have been defined?

[10] Wouldn't + do the trick?

```
(define Even
  (λ (n)
    (Σ ((half Nat))
      (= Nat n (+ half half)))))
```

That would certainly work, because

"For all n, (***twice*** n) equals (***double*** n)."

Although two functions always return the same answer, sometimes one of them is easier to use because it more quickly becomes a value. In particular, **+** and thus ***twice*** leave an `add1` on the second argument, while ***double*** puts both `add1`s at the top immediately.

[11] As seen in the proof of ***twice=double*** in frame 9:52.

How can the statement,

"Two greater than every even number is even."

be written as a type?

[12] Good question.

It can be useful to use more descriptive prose when translating a statement into a type.

Here's another way to say the same thing:

"For every natural number n, if n is even, then $2 + n$ is even."

[13] "Every" sounds like Π.

```
(claim +two-even
  (Π ((n Nat))
    (→ (Even n)
      (Even (+ 2 n)))))
```

Now prove it.

[14] Clearly, the proof uses **ind-Nat** because the type depends on a Nat.

It can actually be done without induction.

But first, how much of the definition can be written now?

[15] Here's a start ...

```
(define +two-even
  (λ (n e_n)
    ...but what goes here? ))
```

Good question.

If 5 is half of 10, then what is half of (+ 2 10)?

---

[16] (+ 2 10) is the same Nat as 12, and half of 12 is 6.

If 6 is half of 12, then what is half of (+ 2 12)?

---

[17] (+ 2 12) is the same Nat as 14, and half of 14 is 7.

There is a repeating pattern here.

Yes, there is a repeating pattern. This pattern can be used to fill the box.

If a is half of n, then what is half of (+ 2 n)?

---

[18] It is (add1 a).

But where is that a in the empty box?

It is (car e_n) because e_n is an (*Even* n).

This means

(car e_n)

is half of n, and

(cdr e_n)

proves this.

---

[19] Right, because car and cdr work with expressions described by Σ.

If p is a

(Σ ((x A))
D),

then (car p) is an A, and (cdr p)'s type is found by consistently replacing each x in D with (car p).

---

[20] That follows directly from the description in frame 10:6 on page 220.

It's possible to go a bit further with the definition of **+*two-even*** now.

[21] The body of the λ-expression has cons at the top because it must be an

(***Even*** (**+** 2 *n*)).

```
(define +two-even
  (λ (n e_n)
    (cons (add1 (car e_n))
          [          ])))
```

And the **car** is (add1 (**car** e_n)) because (**car** e_n) is half of *n*.

So far, so good.

What is (**cdr** e_n)'s type?

[22] (**cdr** e_n) is an

(**=** Nat
n
(***double*** (**car** e_n))).

There is an equality proof available, and it is *almost* correct ...

[23] How can an

(**=** Nat
n
(***double*** (**car** e_n)))

be transformed into an

(**=** Nat
(**+** 2 *n*)
(***double*** (add1 (**car** e_n))))?

This is where the choice of ***double*** over **+** shows its value, just as it did when defining ***double-Vec*** in frame 9:59.

(***double*** (add1 (**car** e_n)))

is the same Nat as

(add1
(add1
(***double*** (**car** e_n)))).

[24] And if the **cdr**'s type had been **claim**ed with **+** or ***twice***, then this Nat would have been

(add1
(**+** (**car** e_n)
(add1 (**car** e_n)))),

and more work would have been required to bring both add1s to the top.

Carefully Choose Definitions

Carefully-chosen definitions can greatly simplify later proofs.

In frame 21's empty box,

```
(cdr e_n)
```

is an

```
(= Nat
  n
  (double (car e_n))).
```

Find an expression that is an

```
(= Nat
  (+ 2 n)
  (add1
    (add1
      (double (car e_n))))).
```

That is precisely what is needed to complete the proof.

Is two even?

[25] The expression

```
(cong (cdr e_n) (+ 2))
```

has that type because

```
(+ 2 n)
```

is the same Nat as

```
(add1
  (add1 n)).
```

---

[26] Thanks for the hints.

```
(define +two-even
  (λ (n e_n)
    (cons (add1 (car e_n))
      (cong (cdr e_n) (+ 2)))))
```

---

[27] Yes, it is.

Prove it, using **+*two-even***.

[28] To use **+*two-even***, we need evidence that 0 is even. This evidence is in frame 26.

```
(claim two-is-even
  (Even 2))
(define two-is-even
  (+two-even 0 zero-is-even))
```

Here is the value of ***two-is-even***.

1. ***two-is-even***
2. (**+*two-even*** 0 ***zero-is-even***)
3. (cons (add1 (**car** ***zero-is-even***))
     (**cong** (**cdr** ***zero-is-even***) (**+** 2)))

Now find the normal form.

[29] The normal form takes just a few more steps.

4. (cons (add1 zero)
     (**cong** (same zero) (**+** 2)))
5. (cons 1
     (same (**+** 2 zero)))
6. (cons 1
     (same 2))

What is an odd number?

[30] An odd number is not even.

Is there a more explicit way to say that?

[31] Odd numbers cannot be split into two equal parts. There is always an add1 remaining.

How can that description be written as a type?

Hint: use the definition of ***Even*** as a guide.

[32] Isn't this an odd definition?

```
(claim Odd
  (→ Nat
    𝒰))
(define Odd
  (λ (n)
    (Σ ((haf Nat))
      (= Nat n (add1 (double haf))))))
```

No, but it is an ***Odd*** definition.

What does *haf* mean?

[33] It is pretty close to *half*. It is half of the even number that is one smaller than *n*.

Here is a claim that 1 is odd.

```
(claim one-is-odd
  (Odd 1))
```

Prove it.

[34] Here is the proof:

```
(define one-is-odd
  (cons 0
    (same 1))).
```

Here, the **cdr** is

(same 1)

because

(same 1)

is an

(= Nat 1 (add1 (***double*** 0))).

Now prove that,

"13 is odd."

[35] "Haf" of a baker's dozen is 6.

```
(claim thirteen-is-odd
  (Odd 13))
(define thirteen-is-odd
  (cons 6
    (same 13)))
```

If n is even, what can be said about (add1 n)?

[36] Would this statement do the trick?

"If n is even, then (add1 n) is odd."

Yes.

How can that be written as a type?

[37] It uses a Π-expression and an $\rightarrow$-expression, because the n means "for every n," and if-then statements are translated to $\rightarrow$-expressions.

Now translate the statement.

[38] Here it is.

```
(claim add1-even→odd
  (Π ((n Nat))
    (→ (Even n)
      (Odd (add1 n)))))
```

Is that claim true.

[39] Yes.

What is the evidence? Remember, truth *is the same as* having evidence, yet no evidence has been provided.

[40] So the statement is false?

No.

There is neither evidence that the statement is true, nor evidence that the statement is false. For now, it is a mystery.

[41] Better solve that mystery then.

To solve the mystery, think about the relationship between half of n and "haf" of (add1 n) when n is even.

[42] They are the same Nat.

Why are they the same Nat?

[43] Because the extra add1 is "used up" in the TO side of the equality in the definition of *Odd*.

Now use this important fact to prove the mystery statement and make it true.

[44] *Et voilà!*

```
(define add1-even→odd
  (λ (n e_n)
    (cons (car e_n)
      (cong (cdr e_n) (+ 1)))))
```

Definitions should be written to be understood.

Why is this definition correct?

[45] In the body of the λ-expression, there is a cons-expression. This expression is the proof of (***Odd*** (add1 *n*)) because the value of (***Odd*** (add1 *n*)) has Σ at the top.

What about the **car** of the proof?

[46] The **car** is (**car** e_n) because "haf" of an odd number is half of the even number that is one smaller.

And what about the **cdr** of the proof?

[47] The **cdr** is built with **cong**, because

(**cdr** e_n)

is an

(**=** Nat
n
(***double*** (**car** e_n))),

but the definition of (***Odd*** (add1 *n*)) demands that the **cdr** be an

(**=** Nat
(add1 *n*)
(add1 (***double*** (**car** e_n)))).

The statement is now true. Take a bow.

If *n* is odd, what can be said about (add1 *n*)?

[48] Clearly,

"If *n* is odd, then (add1 *n*) is even."

That's quite the claim ...

[49] Indeed.

```
(claim add1-odd→even
  (Π ((n Nat))
    (→ (Odd n)
      (Even (add1 n)))))
```

Now it's time to make that claim true.

What is "haf" of 25?

[50] It is 12 because

(add1 (***double*** 12))

is the same Nat as 25.

What is half of 26?

[51] It is 13 because

(***double*** 13)

is the same Nat as 26.

Following this template, what is the relationship between "haf" of some odd number n and half of (add1 n)?

[52] If a is "haf" of the odd number n, then half of the even (add1 n) is (add1 a).

Now start the definition, using this "haf."

[53] Here it is.

```
(define add1-odd→even
  (λ (n o_n)
    (cons (add1 (car o_n))
          [          ])))
```

The box needs an

```
(= Nat
  (add1 n)
  (double (add1 (car o_n)))).
```

Where did that type come from?

[54] It came from the definition of ***Even*** combined with the Commandment of **car**.

What type does (**cdr** o_n) have?

[55] In the box,

(**cdr** o_n)

is an

(**=** Nat
  n
  (add1 (***double*** (**car** o_n)))).

How can (**cdr** o_n) be used to construct evidence that

(**=** Nat
  (add1 *n*)
  (***double*** (add1 (**car** o_n))))?

[56] **cong** does the trick, because

(***double*** (add1 (**car** o_n)))

is the same Nat as

(add1
  (add1
    (***double*** (**car** o_n)))).

```
(define add1-odd→even
  (λ (n o_n)
    (cons (add1 (car o_n))
      (cong (cdr o_n) (+ 1)))))
```

That definition deserves a solid box.

[57] Whew! It's time for another *fika*.

Go eat a haf a baker's dozen muffins

and get ready to divide by two.

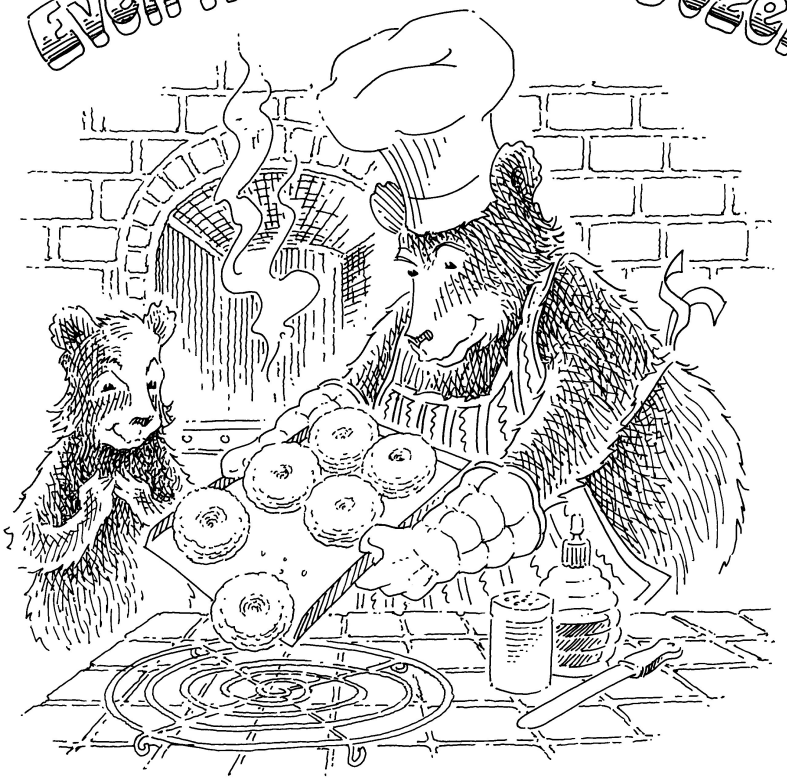
Behold! Ackermann!

```
(claim repeat
  (→ (→ Nat
        Nat)
     Nat
     Nat))
(define repeat
  (λ (f n)
    (iter-Nat n
      (f 1)
      (λ (iter_f,n-1)
        (f iter_f,n-1)))))

(claim ackermann
  (→ Nat Nat
     Nat))
(define ackermann
  (λ (n)
    (iter-Nat n
      (+ 1)
      (λ (ackermann_n-1)
        (repeat ackermann_n-1)))))
```

13

Even Half a Baker's Dozen



Is every natural number either even or odd?

[1] They might be.

But where's the evidence?

Writing

"Every natural number is either even or odd."

as a type requires a new type constructor: Either, which is used to write "or" as a type.

[2] That seems reasonable.

When does Either construct a type?

(Either L R) is a type if L is a type and R is a type.

[3] What are the values of (Either L R)?

The Law of Either

(Either L R) is a type if L is a type and R is a type.

There are two constructors. If lt is an L, then (left lt) is an (Either L R). If rt is an R, then (right rt) is an (Either L R).

When are two (Either L R) values the same?

[4] Here's a guess based on earlier types.

(left lt_1) and (left lt_2) are the same (Either L R) if lt_1 and lt_2 are the same L.

So far, so good. Anything to add?

[5] Yes, one more thing. (right rt_1) and (right rt_2) are the same (Either L R) if rt_1 and rt_2 are the same R.

The Law of left

(left *lt*) is an (Either *L R*) if *lt* is an *L*.

The Law of right

(right *rt*) is an (Either *L R*) if *rt* is an *R*.

Any other possibilities?

[6] Probably not.

That is indeed all of the possibilities.

The eliminator for Either is called **ind-Either**.

[7] That's not a surprise.

---

ind-Either has two bases, but no step.

Why is that?

[8] It is because there are two ways to construct an (Either *L R*), but neither left nor right has an (Either *L R*) as an argument.

So can **ind-Either** introduce recursion?

[9] No, because neither left nor right, Either's two constructors, are recursive.

In an **ind-Either**-expression

(**ind-Either** *target*
mot
base-left
base-right),

target is an (Either *L R*).

[10] Does *mot* explain why *target* is being eliminated?

As usual, it does. *mot*'s type is

(→ (Either L R)
  $\mathcal{U}$).

base-left explains *how* the motive is fulfilled for every left. That is, *base-left*'s type is

(Π ((x L))
  (*mot* (left x))).

[11] Is *base-right*'s type built the same way?

Yes, it is. *base-right* explains *how* the motive is fulfilled for every right.

What is *base-right*'s type?

[12] *base-right*'s type is

(Π ((y R))
  (*mot* (right y)))

because "every" becomes a Π-expression when written as a type.

What is the value of

(**ind-Either** (left x)
  mot
  base-left
  base-right)?

[13] It is the value of (*base-left* x), which is the only available expression with the correct type.

What is the value of

(**ind-Either** (right y)
  mot
  base-left
  base-right)?

[14] It is the value of (*base-right* y), for the same reason.

The Law of ind-Either

If *target* **is an** (Either L R)**,** *mot* **is an**
  ($\rightarrow$ (Either L R)
    $\mathcal{U}$),
base-left **is a**
  (Π ((x L))
    (*mot* (left x))),
and *base-right* **is a**
  (Π ((y R))
    (*mot* (right y)))
then
  (ind-Either *target*
    mot
    base-left
    base-right)
is a (*mot target*)**.**

The First Commandment of ind-Either

(ind-Either (left x)
  mot
  base-left
  base-right)
is the same (*mot* (left x)) **as** (*base-left* x)**.**

The Second Commandment of ind-Either

(ind-Either (right y)
  mot
  base-left
  base-right)
is the same (*mot* (right y)) **as** (*base-right* y)**.**

Now we know how to write,

"Every natural number is even or odd."

as a type.

```
(claim even-or-odd
  (Π ((n Nat))
    (Either (Even n) (Odd n))))
```

---

[15] This is a claim about all Nats. Does the proof use ind-Nat?

Yes, it does.

mot-even-or-odd describes the purpose of the elimination. Try to define it without finding the base first.

```
(claim mot-even-or-odd
  (→ Nat
    𝒰))
```

---

[16] Abstracting over n in frame 15 does it.

```
(define mot-even-or-odd
  (λ (k)
    (Either (Even k) (Odd k))))
```

Good choice.

What is the base?

---

[17] The base is an

(Either (Even zero) (Odd zero))

and zero happens to be even.

Sound familiar?

[18] Yes, it does.

The base is
(left ***zero-is-even***).

It is.

What is the type of the step?

[19] The type of the step is found using the motive.

```
(claim step-even-or-odd
  (Π ((n-1 Nat))
    (→ (mot-even-or-odd n-1)
      (mot-even-or-odd (add1 n-1)))))
```

Now define ***step-even-or-odd***.

[20] Here's a start ...

```
(define step-even-or-odd
  (λ (n-1)
    (λ (e-or-o_n-1)
      ... but what goes here?)))
```

What is $e\text{-}or\text{-}o_{n\text{-}1}$'s type?

[21] The type comes from the step's claim.

1. (***mot-even-or-odd*** *n-1*)
2. (Either (***Even*** *n-1*) (***Odd*** *n-1*))

What is the eliminator for Either?

[22] **ind-Either**, of course.

So eliminate it.

[23] Here's a version with empty boxes in it, at least.

```
(define step-even-or-odd
  (λ (n-1)
    (λ (e-or-o_n-1)
      (ind-Either e-or-o_n-1
        [          ]
        [          ]
        [          ])))))
```

Good start.

What is the motive?

[24] According to ***step-even-or-odd***'s claim, the elimination produces a

(***mot-even-or-odd*** (add1 *n-1*)).

Instead of defining a separate motive, try writing a λ-expression this time. The argument to the motive is the target, but this elimination is not producing a type that depends on the target. So the motive's argument can be dim.

[25] That's a lot shorter than defining it separately.

```
(define step-even-or-odd
  (λ (n-1)
    (λ (e-or-o_n-1)
      (ind-Either e-or-o_n-1
        (λ (e-or-o)
          (mot-even-or-odd (add1 n-1)))
        [          ]
        [          ])))))
```

Yes, it is shorter. But shorter is not always easier to read. Compare the two styles and decide which is easier to understand in each case.

When *n-1* is even, what is the evidence that (add1 *n-1*) is odd?

[26] The evidence can be constructed with ***add1-even→odd***.

The first empty box in frame 25 is an

```
(→ (Even n-1)
  (Either
    (Even (add1 n-1))
    (Odd (add1 n-1)))).
```

[27] The Law of ind-Either states that the base for left is a

```
(Π ((x L))
  (mot (left x))),
```

so why doesn't the empty box's type have Π at the top?

The type has Π at the top. Because → is another way of writing Π when its argument name is not used, → is sufficient, as seen in frame 6:40.

[28] Because (add1 *n-1*) is odd, the expression uses right:

```
(λ (e_{n-1})
  (right
    (add1-even→odd n-1 e_{n-1}))).
```

That's right.

What about the last box?

[29] In that box, *n-1* is odd. Thus, (add1 *n-1*) is even and the expression uses left:

```
(λ (o_{n-1})
  (left
    (add1-odd→even n-1 o_{n-1}))).
```

Now assemble the definition of *step-even-or-odd*.

[30] The boxes are filled in.

```
(define step-even-or-odd
  (λ (n-1)
    (λ (e-or-o_{n-1})
      (ind-Either e-or-o_{n-1}
        (λ (e-or-o_{n-1})
          (mot-even-or-odd
            (add1 n-1)))
        (λ (e_{n-1})
          (right
            (add1-even→odd
              n-1 e_{n-1})))
        (λ (o_{n-1})
          (left
            (add1-odd→even
              n-1 o_{n-1})))))))
```

Now, define ***even-or-odd***.

[31] The pieces are ready.

```
(define even-or-odd
  (λ (n)
    (ind-Nat n
      mot-even-or-odd
      (left zero-is-even)
      step-even-or-odd)))
```

even-or-odd is a proof that

"Every natural number is even or odd."

But it is more than just a proof—it is a λ-expression that produces a value when it gets an argument.

[32] It always produces a value because all functions are total.

Is this value interesting?

Let's find out.

What is the value of (***even-or-odd*** 2)?

[33] That's an interesting question.

Get ready for a long "same-as" chart. Here's the beginning.

1. (***even-or-odd*** 2)
2. ((λ (*n*)
     (**ind-Nat** *n*
       mot-even-or-odd
       (left ***zero-is-even***)
       step-even-or-odd))
   2)
3. (**ind-Nat** 2 . . .)
4. (***step-even-or-odd***
     1
     (**ind-Nat** 1 . . .))

In this chart, . . . , an ellipsis, stands for the arguments to **ind-Nat** or **ind-Either** that don't change at all.

[34] Here's the next one.

5. ((λ (*n-1*)
     (λ (*e-or-o*$_{n-1}$)
       (**ind-Either** *e-or-o*$_{n-1}$
         (λ (*e-or-o*$_{n-1}$)
           (***mot-even-or-odd***
             (add1 *n-1*)))
         (λ (*e*$_{n-1}$)
           (right
             (***add1-even→odd***
               n-1 *e*$_{n-1}$)))
         (λ (*o*$_{n-1}$)
           (left
             (***add1-odd→even***
               n-1 *o*$_{n-1}$))))))
   1 (**ind-Nat** 1 . . .))

At each step, look for the parts of expressions that change and those that don't.

Try to identify motives, bases, and steps that appear multiple times.

[35] What about targets?

Targets are rarely repeated, but worth watching.

6.
```
((λ (e-or-o_{n-1})
   (ind-Either e-or-o_{n-1}
     (λ (e-or-o_{n-1})
       (mot-even-or-odd 2))
     (λ (e_{n-1})
       (right
         (add1-even→odd 1 e_{n-1})))
     (λ (o_{n-1})
       (left
         (add1-odd→even 1 o_{n-1})))))
 (ind-Nat 1 ...))
```

7.
```
(ind-Either (ind-Nat 1 ...)
  (λ (e-or-o_{n-1})
    (mot-even-or-odd 2))
  (λ (e_{n-1})
    (right
      (add1-even→odd 1 e_{n-1})))
  (λ (o_{n-1})
    (left
      (add1-odd→even 1 o_{n-1}))))
```

[36] Ah, because as soon as a target's value is found, a base or step is chosen.

8.
```
(ind-Either
  (step-even-or-odd
    0
    (ind-Nat 0 ...))
  (λ (e-or-o_{n-1})
    (mot-even-or-odd 2))
  (λ (e_{n-1})
    (right
      (add1-even→odd 1 e_{n-1})))
  (λ (o_{n-1})
    (left
      (add1-odd→even 1 o_{n-1}))))
```

9.
```
(ind-Either
  ((λ (n-1)
     (λ (e-or-o_{n-1})
       (ind-Either e-or-o_{n-1} ...)))
   0 (ind-Nat 0 ...))
  (λ (e-or-o_{n-1})
    (mot-even-or-odd 2))
  (λ (e_{n-1})
    (right
      (add1-even→odd 1 e_{n-1})))
  (λ (o_{n-1})
    (left
      (add1-odd→even 1 o_{n-1}))))
```

10.
```
(ind-Either
  ((λ (e-or-o_{n-1})
     (ind-Either e-or-o_{n-1}
       (λ (e-or-o_{n-1})
         (mot-even-or-odd 1))
       (λ (e_{n-1})
         (right
           (add1-even→odd 0 e_{n-1})))
       (λ (o_{n-1})
         (left
           (add1-odd→even 0 o_{n-1})))))
   (ind-Nat 0 ...))
  (λ (e-or-o_{n-1})
    (mot-even-or-odd 2))
  (λ (e_{n-1})
    (right
      (add1-even→odd 1 e_{n-1})))
  (λ (o_{n-1})
    (left
      (add1-odd→even 1 o_{n-1}))))
```

37

11.
```
(ind-Either
  ((λ (e_{n-1})
     (right
       (add1-even→odd 0 e_{n-1})))
   zero-is-even)
  (λ (e-or-o_{n-1})
    (mot-even-or-odd 2))
  (λ (e_{n-1})
    (right
      (add1-even→odd 1 e_{n-1})))
  (λ (o_{n-1})
    (left
      (add1-odd→even 1 o_{n-1}))))
```

12.
```
(ind-Either
  (right
    (add1-even→odd 0 zero-is-even))
  (λ (e-or-o_{n-1})
    (mot-even-or-odd 2))
  (λ (e_{n-1})
    (right
      (add1-even→odd 1 e_{n-1})))
  (λ (o_{n-1})
    (left
      (add1-odd→even 1 o_{n-1}))))
```

13.
```
((λ (o_{n-1})
   (left
     (add1-odd→even 1 o_{n-1})))
 (add1-even→odd 0 zero-is-even))
```

38

14.
```
(left
  (add1-odd→even
    1
    (add1-even→odd
      0
      zero-is-even)))
```

The last expression in the chart is a value.

Whew!

Indeed,

```
(left
  (add1-odd→even
    1
    (add1-even→odd
      0
      zero-is-even)))
```

is a value.

What can we learn from this value?

[39] From this value, it is clear that 2 is even, because the value has left at the top.

In this case, there is still more to be learned.

Find the normal form of

```
(left
  (add1-odd→even
    1
    (add1-even→odd
      0
      zero-is-even))).
```

[40] The first step in finding the normal form is to replace ***add1-odd→even*** with its definition.

That's right.

15.
```
(left
  ((λ (n o_n)
     (cons (add1 (car o_n))
       (cong (cdr o_n) (+ 1))))
   1
   (add1-even→odd
     0
     zero-is-even)))
```

What is next?

[41] The next step is to replace n with 1 and ***add1-even→odd*** with its definition.

16.
```
(left
  ((λ (o_n)
     (cons (add1 (car o_n))
       (cong (cdr o_n) (+ 1))))
   ((λ (n e_n)
      (cons (car e_n)
        (cong (cdr e_n) (+ 1))))
    0
    zero-is-even)))
```

The next step is to drop in the definition of *zero-is-even*.

```
17. (left
      ((λ (o_n)
         (cons (add1 (car o_n))
           (cong (cdr o_n) (+ 1))))
       ((λ (e_n)
          (cons (car e_n)
            (cong (cdr e_n) (+ 1))))
        zero-is-even)))
18. (left
      ((λ (o_n)
         (cons (add1 (car o_n))
           (cong (cdr o_n) (+ 1))))
       ((λ (e_n)
          (cons (car e_n)
            (cong (cdr e_n) (+ 1))))
        (cons 0 (same 0)))))
```

What's next?

Here's the next step.

```
20. (left
      ((λ (o_n)
         (cons (add1 (car o_n))
           (cong (cdr o_n) (+ 1))))
       (cons 0
         (same 1))))
```

What remains?

What can be learned from this normal form?

Each step in the "same as" chart is the same as the previous step, so the value also contains the proof.

---

[42] Next, find the **car** and **cdr** of e_n.

```
19. (left
      ((λ (o_n)
         (cons (add1 (car o_n))
           (cong (cdr o_n) (+ 1))))
       (cons 0
         (cong (same 0) (+ 1)))))
```

It looks like the next step is to find the value of

(**cong** (same 0) (**+** 1)),

and by the Commandment of **cong**, that value is

(same 1).

---

[43] There is one more **cong**-expression that can be made more direct.

```
21. (left
      (cons 1
        (cong (same 1) (+ 1))))
22. (left
      (cons 1
        (same 2)))
```

---

[44] From the value, we see that 2 is even. The normal form also has the *proof* that 2 is even tucked under left.

---

[45] Normal forms, however, are often easier to understand, and this one is no exception.

What can be learned from such a proof?

[46] Not only that 2 is even, but also that 1 is half of 2.

Definitions like ***even-or-odd*** play two roles. In the first role, ***even-or-odd*** is a *proof* that every Nat is either even or odd.

[47] What is the other role?

In the second role, ***even-or-odd*** is a *function* that can determine *whether* a Nat is even or odd. To do so, it finds either half or "haf" of the Nat.

[48] ***even-or-odd*** is interesting *both* as evidence for a statement *and* for the results that it finds.

Exactly. Now, it's time to go for a nice relaxing walk in the woods.

[49] Sounds good.

Every number is even or odd,

and some are smaller than others.

Get ready.

14
There's Safety in Numbers

Please select a dish from this menu:

- ratatouille,
- kartoffelmad,
- hero sandwich, or
- prinsesstårta.

[1] The fourteenth, please.

There are only four dishes on the menu, so you don't get anything.

[2] That's unfortunate.

In order to pick a specific entry from a list, we must know what to do when there are not enough entries.

[3] One might say that there may be an entry, but there also may not be.

To represent the case when there is no entry, we need a new type, called Trivial.[†]

[4] What is Trivial?

[†]Sometimes called the *unit* type.

Trivial is a type, and sole is a Trivial.

Every Trivial expression is the same Trivial as sole.

[5] What about neutral Trivial expressions?

Yes, neutral Trivial expressions are the same as sole. And that's all there is to say about Trivial.

[6] This type is appropriately named.

The Law of Trivial

Trivial **is a type.**

The Law of sole

sole **is a** Trivial**.**

The Commandment of sole

If e **is a** Trivial**, then** e **is the same as** sole**.**

That an entry may or may not be in a list can be represented using ***Maybe***.

```
(claim Maybe
  (→ U
    U))
```

[7] How can ***Maybe*** represent presence or absence?

There is either an X or a Trivial.

```
(define Maybe
  (λ (X)
    (Either X Trivial)))
```

[8] Okay.

Absence is indicated using (right sole).

```
(claim nothing
  (Π ((E U))
    (Maybe E)))
(define nothing
  (λ (E)
    (right sole)))
```

[9] Presumably, presence uses left.

Indeed it does. Here is the claim.

```
(claim just
  (Π ((E U))
    (→ E
      (Maybe E))))
```

[10] In order to use left, an E is necessary.

```
(define just
  (λ (E e)
    (left e)))
```

Using ***Maybe***, it is possible to write a total version of **head** for List.

```
(claim maybe-head
  (Π ((E U))
    (→ (List E)
      (Maybe E))))
```

[11] Following the type, the definition begins with λ.

```
(define maybe-head
  (λ (E es)
    [          ]))
```

What should we expect from (***maybe-head*** Atom nil)?

[12] It should be (***nothing*** Atom) because the empty list has no head.

What should we expect from

```
(maybe-head Atom
  (:: 'ratatouille
    (:: 'kartoffelmad
      (:: (sandwich 'hero)
        (:: 'prinsesstårta nil)))))?
```

[13] It should be (***just*** Atom 'ratatouille).

This is enough information to find the base and step for **rec-List** in the empty box.

[14] Plenty of information.

```
(define maybe-head
  (λ (E es)
    (rec-List es
      (nothing E)
      (λ (hd tl head_tl)
        (just E hd)))))
```

What type should ***maybe-tail*** have?

[15] It is similar to ***maybe-head***, except that it (maybe) finds a list.

```
(claim maybe-tail
  (Π ((E 𝒰))
    (→ (List E)
      (Maybe (List E)))))
```

The definition of ***maybe-tail*** is also very similar to the definition of ***maybe-head***. Only the type in the base and the step's type and value need to change.

[16]

```
(define maybe-tail
  (λ (E es)
    (rec-List es
      (nothing (List E))
      (λ (hd tl tail_tl)
        (just (List E) tl)))))
```

maybe-head and ***maybe-tail*** can be used to define ***list-ref***, which either finds or does not find a specific entry in a list.

[17] What is ***list-ref***'s type?

list-ref accepts an entry type, a Nat, and a list. It may or may not find an entry.

```
(claim list-ref
  (Π ((E 𝒰))
    (→ Nat (List E)
      (Maybe E))))
```

What should we expect from

(***list-ref*** Atom 0 nil)?

---

[18] We should expect nothing, because nil has no entries.

Or, rather, we should expect

(***nothing*** Atom).

What about

```
(list-ref Atom
  zero
  (:: 'ratatouille
    (:: 'kartoffelmad
      (:: (sandwich 'hero)
        (:: 'prinsesstårta nil)))))?
```

---

[19] That's just 'ratatouille, or rather,

(***just*** Atom 'ratatouille).

In other words, when the Nat is zero, ***list-ref*** acts just like ***maybe-head***.

```
(define list-ref
  (λ (E n)
    (rec-Nat n
      (maybe-head E)
      [          ])))
```

---

[20] That's why ***maybe-head*** is the base.

What is the step?

The base is an

(→ (List *E*)
  (***Maybe*** *E*)).

What is the step's type?

---

[21] It should work for any entry type *E*.

```
(claim step-list-ref
  (Π ((E 𝒰))
    (→ Nat
      (→ (List E)
        (Maybe E))
      (→ (List E)
        (Maybe E)))))
```

The step takes as its argument a ***list-ref*** for some smaller Nat, which is *almost* a ***list-ref*** for *this* Nat.

How can a ***list-ref*** for *n-1* be transformed into a ***list-ref*** for *n*?

[22] The ***list-ref*** for *n-1* can be applied to the tail of the list.

Complete this definition.

```
(define step-list-ref
  (λ (E)
    (λ (n-1 list-ref_n-1)
      (λ (es)
        [          ]))))
```

[23] *list-ref*$_{n\text{-}1}$ can be used when ***maybe-tail*** of *es* finds a (List *E*). When ***maybe-tail*** finds ***nothing***, the step finds ***nothing***.

```
(define step-list-ref
  (λ (E)
    (λ (n-1 list-ref_n-1)
      (λ (es)
        (ind-Either (maybe-tail E es)
          (λ (maybe_tl)
            (Maybe E))
          (λ (tl)
            (list-ref_n-1 tl))
          (λ (empty)
            (nothing E)))))))
```

Now define ***list-ref***.

[24] Here it is.

```
(define list-ref
  (λ (E n)
    (rec-Nat n
      (maybe-head E)
      (step-list-ref E))))
```

Take a short break, and maybe eat some delicious ratatouille.

Please select a dish from *this* menu:

1. ratatouille,
2. kartoffelmad,
3. hero sandwich, or
4. prinsesstårta.

[25] The fourteenth, please.

What does "fourteenth" mean?

[26] Ah, there are *precisely* four entries.

What is the difference between a Vec and a List?

[27] In a Vec, the type states how many entries there are. This second menu must be a Vec.

That's right.

```
(claim menu
  (Vec Atom 4))
(define menu
  (vec:: 'ratatouille
    (vec:: 'kartoffelmad
      (vec:: (sandwich 'hero)
        (vec:: 'prinsesstårta vecnil)))))
```

[28] That's one 'delicious hero sandwich.

To define *vec-ref*, a new type is needed: one that represents only numbers smaller than the length of the Vec.

This type is called (*Fin* ℓ), where ℓ is the length.

[29] Why is it called *Fin*?

Fin is a very finite way of writing "finite."

[30] Another abbreviation.

How many natural numbers are smaller than zero?

[31] There are no such numbers.

This requires a new type constructor, named Absurd.[†]

[†]Absurd is sometimes referred to as the *empty type*.

[32] That's an absurd name.

When is Absurd a type?

Absurd is always a type, just like Atom, Nat, $\mathcal{U}$, and Trivial are always types.

[33] That's easy enough.

What are the values of Absurd?

The Law of Absurd

Absurd **is a type.**

There are none, but all of them are the same.

[34] If there are no values of Absurd, how can they be the same?

In fact, *every* expression that is an Absurd is the same as every other expression that is an Absurd.

[35] But there are no Absurd values.

If there are no values, then there is no way to tell any of them apart.

[36] If there are no Absurd values, then how can there be expressions of type Absurd?

Neutral expressions can have type Absurd.

What is the type of x in the body of ***similarly-absurd***'s definition?

```
(claim similarly-absurd
  (→ Absurd
    Absurd))
(define similarly-absurd
  (λ (x)
    x))
```

[37] x is an Absurd.

The Commandment of Absurdities

Every expression of type Absurd **is neutral, and all of them are the same.**

Even though there is no way to construct an Absurd value, there is an eliminator for Absurd.

One way to view an eliminator is as a means of exposing the information inside a constructor. Another way to view it is as a way of picking some new expression for each of a type's values.

[38] ***length*** picks a new Nat for each List, and ***peas*** picks a (Vec Atom ℓ) for each Nat ℓ.

By picking a new expression for each value, the eliminator expression itself has a type given by the motive.

To use the eliminator for Absurd, provide a new expression for each Absurd value.

[39] There are no Absurd values.

Precisely.

The eliminator for Absurd, called **ind-Absurd**, has neither bases nor steps because there are no Absurd values.

[40] How can that be?

There is only a target and a motive.

The expression

(**ind-Absurd** *target*
mot)

is a *mot* when *target* is an Absurd and and when *mot* is a $\mathcal{U}$.

[41] Why isn't *mot* a function?

There are no Absurd values to provide as targets to the motive.

Other eliminators' motives take arguments so that the type of the eliminator expression can mention the target. This is not necessary because there never will be a target.

[42] If *target* can never be a value, what use is **ind-Absurd**?

The Law of ind-Absurd

The expression

(**ind-Absurd** *target*
mot)

is a *mot* **if** *target* **is an** Absurd **and** *mot* **is a** $\mathcal{U}$**.**

It is used to express that some expressions can never be evaluated, or in other words, that the expression is permanently neutral.

[43] And neutral expressions cannot yet be evaluated because the values of their variables are not yet known.

For each Nat n, (***Fin*** n) should be a type with n values.

```
(claim Fin
  (→ Nat
    𝒰))
```

What should the value of (***Fin*** zero) be?

[44] (***Fin*** zero) should have zero values, so Absurd is appropriate.

Here is the beginning of ***Fin***'s definition.

```
(define Fin
  (λ (n)
    (iter-Nat n
      Absurd
      [          ])))
```

What goes in the empty box?

[45] The step for ***Fin***, which goes in the empty box, should transform a type with *n-1* values into a type with n values.

How many values have the type

(***Maybe*** Absurd)?

[46] There is just one,

(***nothing*** Absurd),

which has the normal form

(right sole).

What about Either's constructor left?

[47] That would require an Absurd value, and there are no Absurd values.

How many values have the type

```
(Maybe
  (Maybe Absurd))?
```

[48] There are two possibilities:

(***nothing*** (***Maybe*** Absurd)),

and

```
(just (Maybe Absurd)
  (nothing Absurd)).
```

Based on these examples, if a type X has n values, how many values does

(Maybe X)

have?

[49] It has (add1 n) values because ***Maybe*** adds one value, which is (***nothing*** X).

This means that ***Maybe*** is a suitable step for ***Fin***.

Indeed it is.

[50] Here is the definition.

```
(define Fin
  (λ (n)
    (iter-Nat n
      Absurd
      Maybe)))
```

What is the normal form of (***Fin*** 1)?

[51]

1. (***Fin*** 1)
2. (***Maybe*** Absurd)
3. (Either Absurd Trivial)

This type has 1 value.

What is the normal form of (***Fin*** 2)?

[52] It is

```
(Maybe
  (Maybe Absurd)),
```

better known as

```
(Either (Either Absurd
                Trivial)
  Trivial),
```

which has 2 values.

To use ***Fin*** to pick out entries in a Vec, it is necessary to determine which ***Fin*** points at which entry.

The first entry in a Vec is found using (***fzero*** *n*) when the Vec has (add1 *n*) entries.

```
(claim fzero
  (Π ((n Nat))
    (Fin (add1 n))))
```

---

[53] This is because there are no entries when the length is zero.

Take another look at the definition of ***Fin*** in frame 50. What is another way of writing ***fzero***'s type?

---

[54] **iter-Nat** applies the step when the target has add1 at the top, so ***fzero***'s type and

```
(Π ((n Nat))
  (Maybe (Fin n)))
```

are the same type.

In that type, what are (***Fin*** *n*)'s values?

---

[55] That depends on *n*'s values.

This means that a good choice for ***fzero***'s definition is ...

---

[56] ... (***nothing*** (***Fin*** *n*)), even though it is something rather than nothing.

Good choice. Now define ***fzero***.

---

[57] Here it is.

```
(define fzero
  (λ (n)
    (nothing (Fin n))))
```

Just as (***fzero*** n) points at the head of a (Vec X (add1 n)), ***fadd1*** points somewhere in its tail.

```
(claim fadd1
  (Π ((n Nat))
    (→ (Fin n)
      (Fin (add1 n)))))
```

[58] Why do the two ***Fin***s have different arguments?

Take a look at frame 48. There are two values for (***Fin*** 2). The first is

(***nothing*** (***Maybe*** Absurd)),

also known as (***fzero*** 1).

What is the other?

[59] The other is

(***just*** (***Maybe*** Absurd)
  (***nothing*** Absurd)).

For each layer of ***Maybe*** in the type, there is a choice between either stopping with ***fzero*** (also known as ***nothing***) and continuing with ***just*** a value from the smaller type.

Now define ***fadd1***.

[60] It adds the extra ***just***.

```
(define fadd1
  (λ (n)
    (λ (i-1)
      (just (Fin n) i-1))))
```

Now it's time to define ***vec-ref***, so that there's always something to eat from the menu.

```
(claim vec-ref
  (Π ((E 𝒰)
      (ℓ Nat))
    (→ (Fin ℓ) (Vec E ℓ)
      E)))
```

[61] Here, there is no ***Maybe***.

There are three possibilities:

1. the length ℓ is zero,
2. the length ℓ has add1 at the top and the ***Fin*** is ***fzero***, or
3. the length ℓ has add1 at the top and the ***Fin*** is ***fadd1***.

[62] It depends first and foremost on ℓ. The motive is built by abstracting the rest of the type over ℓ.

```
(define vec-ref
  (λ (E ℓ)
    (ind-Nat ℓ
      (λ (k)
        (→ (Fin k) (Vec E k)
          E))
      [                    ]
      [                    ])))
```

Good start. What is the base's type?

[63] Apply the motive to zero.

```
(claim base-vec-ref
  (Π ((E 𝒰))
    (→ (Fin zero) (Vec E zero)
      E)))
```

The only constructor for (Vec E zero) is vecnil, but vecnil does not contain any Es.

What is the value of (***Fin*** zero)?

[64] The value of (***Fin*** zero) is Absurd.

Because there are no Absurd values, the base can never be applied to its second argument's value.

Use **ind-Absurd** to take advantage of this fact.

[65] Okay.

```
(define base-vec-ref
  (λ (E)
    (λ (no-value-ever es)
      (ind-Absurd no-value-ever
        E))))
```

Now it is time to define the step.

What is the step's type?

[66] Once again, it is found by the Law of **ind-Nat**.

```
(claim step-vec-ref
  (Π ((E 𝒰)
      (ℓ-1 Nat))
    (→ (→ (Fin ℓ-1)
          (Vec E ℓ-1)
          E)
       (→ (Fin (add1 ℓ-1))
          (Vec E (add1 ℓ-1))
          E))))
```

There are now two possibilities remaining:

1. the ***Fin*** is ***fzero***, or
2. the ***Fin*** is ***fadd1***.

What is the value of (***Fin*** (add1 *ℓ-1*))?

[67] The value has Either at the top.

1. (***Fin*** (add1 *ℓ-1*))
2. (***Maybe*** (***Fin*** *ℓ-1*))
3. (Either (***Fin*** *ℓ-1*) Trivial)

What can be used to check which Either it is?

[68] The only eliminator for Either is **ind-Either**.

If the ***Fin*** is ***fzero***, then it has right at the top, and if it is ***fadd1***, then it has left at the top.

If the ***Fin*** has left at the top, then there should be recursion to check the Vec's tail. If it has right at the top, then find the Vec's **head**.

[69] **ind-Either** is used to distinguish between Either's constructors.

Indeed it is. Define the step.

[70] Here goes.

```
(define step-vec-ref
  (λ (E ℓ-1)
    (λ (vec-ref_ℓ-1)
      (λ (i es)
        (ind-Either i
          (λ (i)
            E)
          (λ (i-1)
            (vec-ref_ℓ-1
              i-1 (tail es)))
          (λ (triv)
            (head es)))))))
```

Now define ***vec-ref***.

[71] The boxes are all filled.

```
(define vec-ref
  (λ (E ℓ)
    (ind-Nat ℓ
      (λ (k)
        (→ (Fin k) (Vec E k)
          E))
      (base-vec-ref E)
      (step-vec-ref E))))
```

Now that it's clear how to find entries in ***menu***, which one do you want?

[72] The second one.

The second one?

[73] Pardon me.

The

```
  (fadd1 3
    (fzero 2))
```

nd one, please.

Let's find it. Here's the first few steps.

1.
```
(vec-ref Atom 4
  (fadd1 3
    (fzero 2))
  menu)
```
2.
```
((λ (E ℓ)
   (ind-Nat ℓ
     (λ (k)
       (→ (Fin k) (Vec E k)
         E))
     (base-vec-ref E)
     (step-vec-ref E)))
 Atom (add1 3)
 (fadd1 3
   (fzero 2))
 menu)
```
3.
```
((ind-Nat (add1 3)
   (λ (k)
     (→ (Fin k) (Vec Atom k)
       Atom))
   (base-vec-ref Atom)
   (step-vec-ref Atom))
 (fadd1 3
   (fzero 2))
 menu)
```
4.
```
((step-vec-ref Atom (add1 2)
   (ind-Nat (add1 2)
     (λ (k)
       (→ (Fin k) (Vec Atom k)
         Atom))
     (base-vec-ref Atom)
     (step-vec-ref Atom)))
 (fadd1 3
   (fzero 2))
 menu)
```

[74] The motive, base, and step in the **ind-Nat**-expression do not change, so they are replaced with an ellipsis, just like in frame 13:34.

5.
```
(((λ (E ℓ-1)
    (λ (vec-ref_ℓ-1)
      (λ (f es)
        (ind-Either f
          (λ (i)
            E)
          (λ (i-1)
            (vec-ref_ℓ-1
              i-1 (tail es)))
          (λ (triv)
            (head es))))))
  Atom (add1 2)
  (ind-Nat (add1 2) ...))
 (fadd1 3
   (fzero 2))
 menu)
```
6.
```
(((λ (vec-ref_ℓ-1)
    (λ (f es)
      (ind-Either f
        (λ (i)
          Atom)
        (λ (i-1)
          (vec-ref_ℓ-1
            i-1 (tail es)))
        (λ (triv)
          (head es)))))
  (ind-Nat (add1 2) ...))
 (fadd1 3
   (fzero 2))
 menu)
```

That's a good start.

```
7.  ((λ (f es)
       (ind-Either f
         (λ (i)
           Atom)
         (λ (i-1)
           ((ind-Nat (add1 2) ...)
            i-1 (tail es)))
         (λ (triv)
           (head es))))
     (fadd1 3
       (fzero 2))
     menu)
8.  (ind-Either (fadd1 3
                       (fzero 2))
       (λ (i)
         Atom)
       (λ (i-1)
         ((ind-Nat (add1 2) ...)
          i-1 (tail menu)))
       (λ (triv)
         (head menu)))
9.  (ind-Either (left (fzero 2))
       (λ (i)
         Atom)
       (λ (i-1)
         ((ind-Nat (add1 2) ...)
          i-1 (tail menu)))
       (λ (triv)
         (head menu)))
10. ((ind-Nat (add1 2) ...)
     (fzero 2) (tail menu))
```

75

```
11. (step-vec-ref Atom (add1 1)
      (ind-Nat (add1 1) ...)
      (fzero 2)
      (tail menu))
12. ((λ (E ℓ-1)
       (λ (vec-ref_ℓ-1)
         (λ (f es)
           (ind-Either f
             (λ (i)
               E)
             (λ (i-1)
               (vec-ref_ℓ-1
                 i-1 (tail es)))
             (λ (triv)
               (head es))))))
     Atom
     (add1 1)
     (ind-Nat (add1 1) ...)
     (fzero 2)
     (tail menu))
13. ((λ (vec-ref_ℓ-1)
       (λ (f es)
         (ind-Either f
           (λ (i)
             Atom)
           (λ (i-1)
             (vec-ref_ℓ-1
               i-1 (tail es)))
           (λ (triv)
             (head es)))))
     (ind-Nat (add1 1) ...)
     (fzero 2)
     (tail menu))
```

Almost there!

```
14. ((λ (f es)
       (ind-Either f
         (λ (i)
           Atom)
         (λ (i-1)
           ((ind-Nat (add1 1) ...)
            i-1 (tail es)))
         (λ (triv)
           (head es))))
     (fzero 2)
     (tail menu))
15. ((λ (es)
       (ind-Either (fzero 2)
         (λ (i)
           Atom)
         (λ (i-1)
           ((ind-Nat (add1 1) ...)
            i-1 (tail es)))
         (λ (triv)
           (head es))))
     (tail menu))
```

[76]

```
16. (ind-Either (fzero 2)
      (λ (i)
        Atom)
      (λ (i-1)
        ((ind-Nat (add1 1) ...)
         i-1 (tail (tail menu))))
      (λ (triv)
        (head (tail menu))))
17. (head (tail menu))
18. 'kartoffelmad
```

Finally, my 'kartoffelmad is here.

Enjoy your smørrebrød

things are about to get subtle.

Turner's Teaser

Define a function that determines whether another function that accepts any number of Eithers always returns left. Some say that this can be difficult with types.[†] Perhaps they are right; perhaps not.

```
(claim Two
  𝒰)
(define Two
  (Either Trivial Trivial))

(claim Two-Fun
  (→ Nat
    𝒰))
(define Two-Fun
  (λ (n)
    (iter-Nat n
      Two
      (λ (type)
        (→ Two
          type)))))

(claim both-left
  (→ Two Two
    Two))
(define both-left
  (λ (a b)
    (ind-Either a
      (λ (c)
        Two)
      (λ (left-sole)
        b)
      (λ (right-sole)
        (right sole)))))

(claim step-taut
  (Π ((n-1 Nat))
    (→ (→ (Two-Fun n-1)
          Two)
      (→ (Two-Fun (add1 n-1))
        Two))))
(define step-taut
  (λ (n-1 taut_n-1)
    (λ (f)
      (both-left
        (taut_n-1
          (f (left sole)))
        (taut_n-1
          (f (right sole)))))))

(claim taut
  (Π ((n Nat))
    (→ (Two-Fun n)
      Two)))
(define taut
  (λ (n)
    (ind-Nat n
      (λ (k)
        (→ (Two-Fun k)
          Two))
      (λ (x)
        x)
      step-taut)))
```

[†]Thanks, David A. Turner (1946–).

15
Imagine That . . .

We have proved that many different expressions are equal.

1 That's right.

Not every pair of expressions are equal, however. Clearly,

"39 is not equal to 117."

2 Can that statement also be written as a type?

A statement is true when we have evidence that it is true. False statements have no evidence at all.

3 This sounds like Absurd.

It does.

The eliminator **ind-Absurd** corresponds to a principle of thought.

4 What principle is that?

If a false statement were true, then we might as well say anything at all.

5 Sounds reasonable enough.

That principle† is induction for Absurd.

Here is the type that captures the meaning of the statement from frame 2.

```
(→ (= Nat 39 117)
   Absurd)
```

6 Why does

```
(→ X
   Absurd)
```

capture the meaning of "not X?"

†Also known as *the Principle of Explosion* or *ex falso quodlibet*, which means "from false, anything."

It says,

"If there were a proof that 39 equals 117, then there would be a proof of Absurd."

7 Providing evidence that 39 equals 117 as an argument to the function, whose type is in the preceding frame, would result in a proof of Absurd. And we know that no such proof exists.

There is no proof of Absurd, so there can't be a proof of (= Nat 39 117).

[8] But if there are no Absurd values, then how can a "not" statement have a proof?

What could be in the body of the λ-expression?

The key is to carefully avoid having to write the body of the λ-expression.

[9] How can that be achieved?

With attention to detail and an open mind.

First, we define what the consequences are of the fact that two Nats are equal.

[10] What are the consequences?

It depends on which Nats they are.

```
(claim =consequence
  (→ Nat Nat
    𝒰))
```

[11] Okay. What is the definition of *=consequence*?

If zero equals zero, nothing interesting is learned. This can be represented using Trivial.

[12] What does Trivial mean as a statement?

To understand Trivial as a *statement*, consider how to prove it.

[13] There is sole.

That's the proof.

[14] Rather trivial.

Is there an eliminator for Trivial?

There could be, but it would be pointless.

There is only one Trivial value, so nothing is to be learned from eliminating it.

[15] So Trivial is a boring statement that can always be proved.

Just like Π and Σ, normal expressions with type Trivial are always values.

[16] This is because every expression with type Trivial is the same as sole.

The fact that Trivial is not an interesting statement makes it a perfect type to represent that nothing is learned from

(= Nat zero zero).

[17] What else can be learned if two Nats are equal?

If

(= Nat (add1 *n-1*) zero)

or

(= Nat zero (add1 *j-1*))

is true, then anything at all can be true. So the consequence is Absurd.

[18] That makes sense.

Finally, if

(= Nat (add1 *n-1*) (add1 *j-1*))

is true, what is the consequence?

[19] It must be that *n-1* and *j-1* are equal Nats.

This table represents the four possibilities.

	zero	(add1 *j-1*)
zero	Trivial	Absurd
(add1 *n-1*)	Absurd	(= Nat *n-1 j-1*)

[20] The function is not recursive, so **which-Nat** is enough.

Here is *=consequence*'s definition.

Check that each part of the table matches.

```
(define =consequence
  (λ (n j)
    (which-Nat n
      (which-Nat j
        Trivial
        (λ (j-1)
          Absurd))
      (λ (n-1)
        (which-Nat j
          Absurd
          (λ (j-1)
            (= Nat n-1 j-1)))))))
```

[21] Each does.

If *=consequence* tells us it is true about two equal Nats, then it should certainly be true when the Nats are the same.

How can this goal be written as a type?

[22] *n* is clearly the same Nat as *n*.

```
(claim =consequence-same
  (Π ((n Nat))
    (=consequence n n)))
```

That's right.

[23] Here's the start of a proof.

The motive is built by abstracting the **ind-Nat**-expression's type over *n*.

```
(define =consequence-same
  (λ (n)
    (ind-Nat n
      (λ (k)
        (=consequence k k))
      [          ]
      [          ])))
```

What is the base's type?

[24] As usual, the base's type is the motive applied to zero, which is

Trivial.

So the base is

sole.

What about the step?

[25] The step's type is

```
(Π ((n-1 Nat))
  (→ (=consequence n-1 n-1)
    (=consequence
      (add1 n-1)
      (add1 n-1)))),
```

which is also found by applying the motive.

The step has λ at the top. What type is expected in the empty box?

```
(λ (n-1 almost)
  [          ])
```

[26] The value of

```
(=consequence (add1 n-1) (add1 n-1))
```

is

```
(= Nat n-1 n-1).
```

n-1 and *n-1* are the same Nat, so (same *n-1*) fits in this box.

Now fill in the boxes and define *=consequence-same*.

[27] Like ***zerop***, the step's second argument is dim because the function is not recursive.

```
(define =consequence-same
  (λ (n)
    (ind-Nat n
      (λ (k)
        (=consequence k k))
      sole
      (λ (n-1 =consequence_n-1)
        (same n-1)))))
```

Could *=consequence-same* have been defined with **which-Nat**?

[28] No. Because the type of the **ind-Nat**-expression depends on the target, **ind-Nat** is needed.

Now comes the tricky part.

The proof of *=consequence* for Nats that are the same can be used to prove *=consequence* for any two *equal* Nats.

[29] If two Nats are equal, aren't they the same?

Not necessarily. Using types, it is possible to *assume* things that may or may not be true, and then see what can be concluded from these assumptions.

How can the type

```
(→ (= Nat 0 6)
   (= Atom 'powdered 'glazed))
```

be read as a statement?

[30] That type can be read,

"If zero equals six, then powdered donuts are glazed donuts."

Because there is no evidence that

"Zero equals six,"

there are no suitable arguments for this function.

This is fortunate for those who have discriminating taste in desserts.

[31] Donuts can be part of a mid-afternoon *fika* as well as dessert.

Imagine That ...

Using types, it is possible to *assume* things that may or may not be true, and then see what can be concluded from these assumptions.

Sameness is not a type, it is a judgment, as seen in frame 8:21.

Either two expressions are the same, or they are not the same, but there is no way to provide evidence of this sameness. Types, such as =-expressions, *can* have evidence.

[32] Are more things equal than are the same?

Indeed. There is a good reason that more things are equal than are the same. The fact that any two expressions either are or are not the same means that we are freed from the obligation to provide a proof because sameness can be determined by following the Laws and Commandments.

Equality requires *proof*, and therefore is more expressive. Recognizing a proof requires only the Laws and Commandments, but constructing a proof may require creativity, ingenuity, and hard work.

[33] How is this expressive power useful?

Types can occur in other types. It is possible to assume that two Nats are equal, and then use that assumption to *prove* the consequences from frame 20.

[34] Even the Absurd consequences?

Sameness versus Equality

Either two expressions are the same, or they are not. It is impossible to prove that they are the same because sameness is a judgment, not a type, and a proof is an expression with a specific type.

Even the Absurd consequences.

It is not possible to prove Absurd, but it is possible to exclude those two Absurd cases using the equality assumption.

[35] So the statement to be proved is

"If n and j are equal Nats, then the consequences from frame 20 follow."

Here is that statement as a type that explains how a proof that n and j are equal can be used.

```
(claim use-Nat=
  (Π ((n Nat)
      (j Nat))
    (→ (= Nat n j)
      (=consequence n j))))
```

[36] The proof definitely has λs at the top.

```
(define use-Nat=
  (λ (n j)
    (λ (n=j)
      [          ])))
```

Here comes the trick.

replace can make n the same as j, which allows *=consequence-same* to prove *use-Nat=*.

[37] But what if they are not the same?

Then there is no need to worry.

If there is no evidence that n equals j, then there are no suitable arguments.

[38] Here is the definition with **replace** in the box.

```
(define use-Nat=
  (λ (n j)
    (λ (n=j)
      (replace n=j
        [   ]
        (=consequence-same [   ])))))
```

The target is n=j.

Should n or j be the argument to *=consequence-same*?

What is the FROM and what is the TO in *n=j*'s type?

39 The FROM is *n* and the TO is *j*. This means that the base's type is the motive applied to *n*.

What about the entire **replace**-expression's type?

40 It is the motive applied to *j*.

The base must be

(***=consequence-same*** *n*)

because the FROM is *n*.

What should the motive be?

41 The whole **replace**-expression is an

(***=consequence*** *n j*),

so the motive should abstract over *j*.

The *n* in the base's type is replaced by *j*.

Now finish the definition.

Remember that

(***=consequence-same*** *n*)

is an

(***=consequence*** *n n*).

42 That was quite the trick!

```
(define use-Nat=
  (λ (n j)
    (λ (n=j)
      (replace n=j
        (λ (k)
          (=consequence n k))
        (=consequence-same n)))))
```

Is ***use-Nat=*** useful?

It can be used to prove

"If zero equals six, then powdered donuts equal glazed donuts."

43 How does that proof work?

The first step is to prove

"zero does not equal any Nat that has add1 at the top."†

†This statement is sometimes called *no confusion* or *disjointness*.

44 That statement can be written as a type.

```
(claim zero-not-add1
  (Π ((n Nat))
    (→ (= Nat zero (add1 n))
      Absurd)))
```

Use the table in frame 20 to find the consequences of zero being equal to (add1 *n*).

45 That's Absurd.

What happens if **use-Nat=** is applied to zero and (add1 *n*)?

46 The type of

```
  (use-Nat= zero (add1 n))
```

is

```
  (→ (= Nat zero (add1 n))
    (=consequence zero (add1 n))),
```

and

```
    (=consequence zero (add1 n))
```

and

```
    Absurd
```

are the same type.

Voilà! The proof.

47 Oh, it is.

```
(define zero-not-add1
  (λ (n)
    (use-Nat= zero (add1 n))))
```

Now prove ***donut-absurdity***.

```
(claim donut-absurdity
  (→ (= Nat 0 6)
    (= Atom 'powdered 'glazed)))
```

[48] If there were evidence that

"0 equals 6,"

then there would be evidence for anything at all, including strange facts about donuts.

```
(define donut-absurdity
  (λ (zero=six)
    (ind-Absurd
      (zero-not-add1 5 zero=six)
      (= Atom 'powdered 'glazed))))
```

What are the consequences if two Nats with add1 at the top are equal?

[49] According to the table, the Nats tucked under the add1s are also equal.

This means that,

"If

73 equals 73,

then

72 equals 72."

Prove this statement:

"For every two Nats n and j,

if

(add1 n) equals (add1 j),

then

n equals j."

[50] It is called ***sub1=***. Because it is part of the table, ***use-Nat=*** is enough!

```
(claim sub1=
  (Π ((n Nat)
      (j Nat))
    (→ (= Nat (add1 n) (add1 j))
      (= Nat n j))))
(define sub1=
  (λ (n j)
    (use-Nat= (add1 n) (add1 j))))
```

Now prove that 1 does not equal 6.

```
(claim one-not-six
  (→ (= Nat 1 6)
    Absurd))
```

---

[51] Does the proof use induction?

No.

Induction is used to prove something for *every* Nat. For these specific Nats, it is not necessary. Just use ***zero-not-add1*** and ***sub1=***.

[52] ***sub1=*** can be used to show that,

"If 1 equals 6, then 0 equals 5."

zero-not-add1 can be used to show that 0 does not equal 5.

That's a good strategy.

Define ***one-not-six***.

[53] Here it is.

```
(define one-not-six
  (λ (one=six)
    (zero-not-add1 4
      (sub1= 0 5 one=six))))
```

Absurd is useful for more than just statements involving "not."

Just as **ind-List** can do anything that **rec-List** can do, **ind-Vec** can do anything that **head** can do. Sometimes, however, Absurd is a necessary part of such definitions.

[54] If that's the case, then a function that behaves very much like **head** can be defined using **ind-Vec**.

```
(claim front
  (Π ((E 𝒰)
      (n Nat))
    (→ (Vec E (add1 n))
      E)))
```

The direct approach used in previous invocations of **ind-Vec** does not work here.

What is the type of the expression that could fill the box?

```
(define front
  (λ (E ℓ es)
    (ind-Vec (add1 ℓ) es
      (λ (k xs)
        E)
      [          ]
      (λ (k h t front_ys)
        h))))
```

[55] It would be E, but no E is available because vecnil is empty.

There is no way to fill this box, but this bad definition of ***front*** provides no evidence of that fact in the base.

The solution is to change the motive so that the base's type contains this evidence.

[56] So the motive isn't boring, is it?

---

ind-Vec can eliminate *any* Vec, but ***front*** only works on Vecs whose length has add1 at the top. Because **ind-Vec** is *too powerful* for this task, it must be restricted to rule out the need for a base. This is done by carefully choosing the motive.

[57] What motive can be used here?

What is the purpose of the motive in **ind-Vec**?

[58] The motive explains how the type of the **ind-Vec**-expression depends on the two targets.

---

mot-front has a type like any other motive.

```
(claim mot-front
  (Π ((E U)
      (k Nat))
    (→ (Vec E k)
      U)))
```

[59] This is no different from other uses of **ind-Vec**.

That's right.

The definition of ***mot-front***, however, is quite different.

```
(define mot-front
  (λ (E)
    (λ (k es)
      (Π ((j Nat))
        (→ (= Nat k (add1 j))
          E)))))
```

[60] Please explain that definition.

The argument k is a target of **ind-Vec**. Both the base and the step now have two extra arguments: a Nat called j and a proof that k is (add1 j).

In the base, k is zero. Thus, there is no such j.

[61] If there were such a j, then zero would equal (add1 j). But ***zero-not-add1*** proves that this is impossible.

Exactly.

zero-not-add1 can be used with **ind-Absurd** to show that no value is needed for the base.

[62] What about the step?

What is the step's type?

63 The step's type follows the Law of **ind-Vec**.

```
(claim step-front
  (Π ((E U)
      (ℓ Nat)
      (e E)
      (es (Vec E ℓ)))
    (→ (mot-front E
         ℓ
         es)
       (mot-front E
         (add1 ℓ)
         (vec:: e es)))))
```

Here is the start of ***step-front***. What belongs in the box?

```
(define step-front
  (λ (E ℓ e es)
    (λ (front_es)
      [          ])))
```

64 The box is a

```
(mot-front E
  (add1 ℓ)
  (vec:: e es)).
```

What is the purpose of the expression that goes in the box?

65 ***front*** is not really recursive—like ***zerop***, the answer is determined by the top constructor of **ind-Vec**'s target. The answer is *e*, which is the first entry under vec:: in the list.

What is the normal form of the empty box's type

```
(mot-front E
  (add1 ℓ)
  (vec:: e es))?
```

66 The normal form is

```
(Π ((j Nat))
  (→ (= Nat (add1 ℓ) (add1 j))
    E)).
```

What does this type mean?

[67] It means that the step builds a function that takes a Nat called j and evidence that (add1 ℓ) is (add1 j), and then produces an E. The only E here is e.

In the base's type, the motive requires that zero has add1 at the top, so no base is needed. A step, however, can be written because (add1 ℓ) does have add1 at the top.

What is the purpose of ***step-front***?

[68] ***step-front*** finds the value of ***front*** for non-empty Vecs, which is the first entry in the Vec.

Define ***step-front***.

[69] Because ***front*** is not recursive, $front_{es}$ is dim. Similarly, the specific length is not important, because it is not zero. The empty box is filled with a function that ignores its arguments, resulting in e.

```
(define step-front
  (λ (E ℓ e es)
    (λ (front_es)
      (λ (j eq)
        e))))
```

Because the value of ***mot-front*** is a Π-expression, the ind-Vec-expression in ***front*** has a function type.

[70] That's not right. ***front*** should find the first entry in a Vec, not a function.

The function type found by ***mot-front*** expects two arguments: a new Nat called j and evidence that the length of the Vec is (add1 j).

[71] How does that help?

According to ***front***'s type, the Vec's length already has add1 at the top.

[72] So the new Nat is ℓ because the length of the Vec is (add1 ℓ).

Yes.

And, proving that

"(add1 ℓ) equals (add1 ℓ)"

does not require a complicated argument.

[73] Right.

Because (same (add1 ℓ)) does it.

Now, define ***front***.

[74] Because the **ind-Vec**-expression's type is a Π-expression, it can be applied to ℓ and (same (add1 ℓ)).

```
(define front
  (λ (E ℓ es)
    ((ind-Vec (add1 ℓ) es
       (mot-front E)
       (λ (j eq)
         (ind-Absurd
           (zero-not-add1 j eq)
           E))
       (step-front E))
     ℓ (same (add1 ℓ)))))
```

Congratulations!

Being able to design appropriate motives for definitions such as ***front*** is very important. A similar technique is used to write ***drop-last*** or ***rest*** using **ind-Vec**.

[75] This sounds like a valuable skill.

Finding values is a valuable skill as well. What is the value of

```
(front Atom 2
  (vec:: 'sprinkles
    (vec:: 'chocolate
      (vec:: 'maple vecnil))))?
```

[76] The first step is to apply ***front*** to its arguments.

```
((ind-Vec (add1 2)
          (vec:: 'sprinkles
            (vec:: 'chocolate
              (vec:: 'maple vecnil)))
   (mot-front Atom)
   (λ (j eq)
     (ind-Absurd
       (zero-not-add1 j eq)
       Atom))
   (step-front Atom))
 2 (same (add1 2)))
```

What's next?

[77] **ind-Vec**'s targets have add1 and vec:: at the top, so ***step-front*** is next.

```
((step-front E 2
   'sprinkles
   (vec:: 'chocolate
     (vec:: 'maple vecnil))
   (ind-Vec 2 (vec:: 'chocolate
                (vec:: 'maple vecnil))
     (mot-front Atom)
     (λ (j eq)
       (ind-Absurd
         (zero-not-add1 j eq)
         Atom))
     (step-front Atom)))
 2 (same (add1 2)))
```

```
((λ (j eq)
   'sprinkles)
 2 (same (add1 2)))
```

```
'sprinkles
```

Take a cozy break for *fika* if you feel the need.

[78] See you in half an hour.

How was the coffee and donuts?

79 *Läckert!*

Is every statement true or false?

80 Clearly.

"Every statement is true or false." is called *the Principle of the Excluded Middle.*†

†Sometimes, the Principle of the Excluded Middle is called the "Law" of the Excluded Middle. It is also sometimes written *tertium non datur*, which means "there is no third choice."

81 Let's prove it.

Write the statement

"*Every* statement is true or false."

as a type.

82 Statements are types. How can "is false" be written as a type?

If a statement is false, it has no evidence. This can be written as an "if-then" statement.

"X is false" is written

```
(→ X
   Absurd).
```

83 "Every statement is true or false." is a Π-expression.

```
(claim pem
  (Π ((X 𝒰))
    (Either X
      (→ X
        Absurd))))
```

There is no evidence for ***pem***.

84 Why not?

What would count as evidence for ***pem***?

85 Evidence for ***pem*** would be a function that determines the truth or falsity of *every statement that can be written as a type.*

Every single statement?

[86] Every single statement, because Π means "every."

This would mean that there are no unsolved problems.

[87] Great! No more problems.

Life would be boring if we had no problems left to solve.

[88] So ***pem*** isn't true, but it can't possibly be false!

That's right.

[89] It's not true, but it can't be false?

That's right. It can't possibly be false.

Write

"'Every statement is true or false' can't possibly be false."

as a type.

[90] In other words,

"'"Every statement is true or false" is false' is false."

```
(claim pem-not-false
  (Π ((X U))
    (→ (→ (Either X
              (→ X
                 Absurd))
          Absurd)
       Absurd)))
```

That's right. Now prove ***pem-not-false***.

[91] How?

What counts as evidence for a Π-expression?

[92] A λ-expression.

```
(define pem-not-false
  (λ (X)
    [          ]))
```

What is the empty box's type?

93 The empty box is an

```
(→ (→ (Either X
          (→ X
              Absurd))
        Absurd)
    Absurd),
```

so it should also be filled with a λ-expression.

That's right.

Continue the proof.

94 This new λ-expression accepts evidence that the Principle of the Excluded Middle is false for X as its argument.

```
(define pem-not-false
  (λ (X)
    (λ (pem-false)
      [          ])))
```

What can be done with *pem-false*?

95 *pem-false*'s type has → at the top, so it can be eliminated by applying it. The empty box's type is Absurd, and *pem-false* would produce evidence of Absurd if it were applied to a suitable argument.

What is the type of suitable arguments?

```
(define pem-not-false
  (λ (X)
    (λ (pem-false)
      (pem-false [          ]))))
```

96 The box's type is

```
(Either X
  (→ X
    Absurd)).
```

There are two ways to construct one of those.

Is left relevant?

[97] No, left is not relevant because there is no evidence for X available.

What about right?

```
(define pem-not-false
  (λ (X)
    (λ (pem-false)
      (pem-false
        (right
          [          ])))))
```

What is the empty box's type now?

[98] The box's type is

$(\rightarrow X$
Absurd).

What is evidence for an $\rightarrow$?

[99] A λ-expression. This new box's type is Absurd.

```
(define pem-not-false
  (λ (X)
    (λ (pem-false)
      (pem-false
        (right
          (λ (x)
            [          ]))))))
```

What can be used to make an Absurd?

[100] *pem-false* can.

Give it a try.

101 Again? Okay.

```
(define pem-not-false
  (λ (X)
    (λ (pem-false)
      (pem-false
        (right
          (λ (x)
            (pem-false □)))))))
```

This box's type is

```
(Either X
  (→ X
    Absurd)).
```

102 Isn't this getting a bit repetitive?

The difference is that there is now an X available.

103 This means that left can be used.

```
(define pem-not-false
  (λ (X)
    (λ (pem-false)
      (pem-false
        (right
          (λ (x)
            (pem-false
              (left x))))))))
```

Nice proof.

104 But if the Principle of the Excluded Middle is not false, why isn't it true?

Very funny.

If ***pem*** were true, then we would have evidence: a magical total function that solves every problem that we can write as a type.

105 So evidence that a statement is not false is less interesting than evidence that it is true?

Exactly.

There are, however, some statements that *are* either true or false. These statements are called *decidable* because there is a function that decides whether they are true or false.

[106] Can "X is decidable" be written as a type?

It certainly can.

```
(claim Dec
  (→ U
    U))
(define Dec
  (λ (X)
    (Either X
      (→ X
        Absurd))))
```

[107] That looks a lot like ***pem***.

Another way to phrase ***pem*** is

"All statements are decidable."

[108] So ***pem***'s claim could have been written using ***Dec***.

```
(claim pem
  (Π ((X U))
    (Dec X)))
```

Some statements are decidable, even though not all statements are decidable.

[109] How about deciding that this has been enough for today?

Sure. Tomorrow, we encounter a decidable statement.

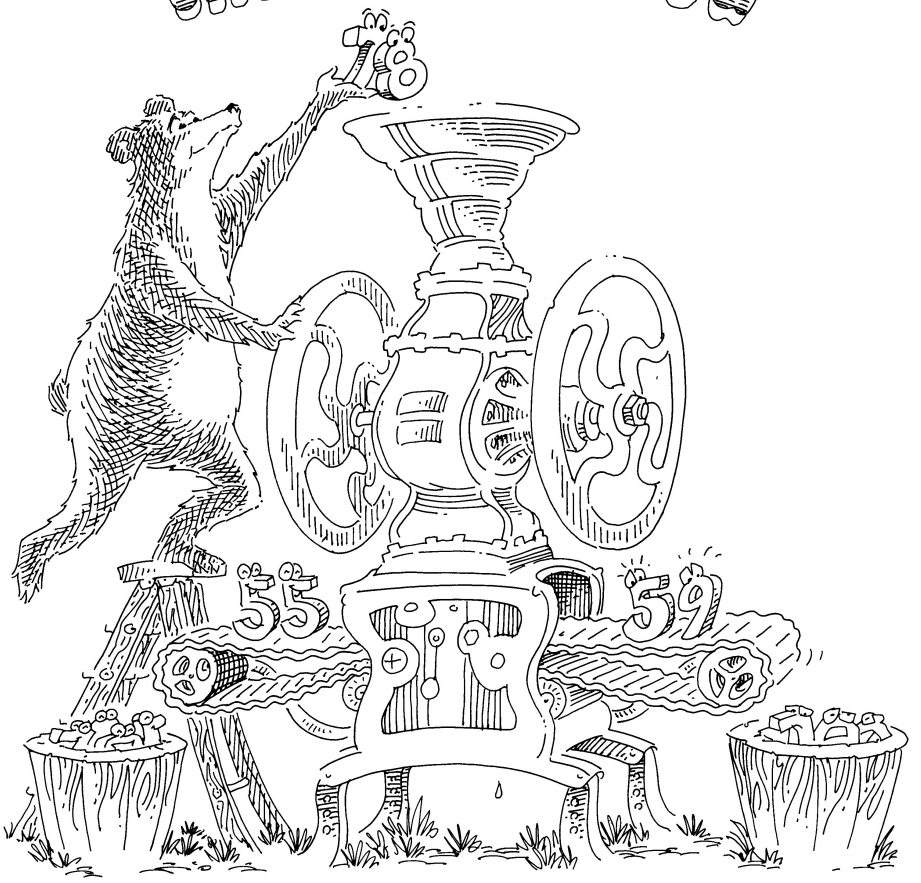
[110] It's a good thing there are more donuts.

Enjoy your donuts

you'll need your energy for tomorrow's decisions.

This page is not unintentionally left blank.

16
If It's All the Same to You



Remember ***zerop*** from chapter 3?

[1] Refresh my memory.

If n is a Nat, then (***zerop*** n) is an Atom.

[2] Which Atom is it?

Good question.

The type

(→ Nat
  Atom)

is not particularly specific.

[3] No, it isn't. But, based on frame 3:43 on page 80,

(***zerop*** n)

is 't when n is zero, and 'nil otherwise.

The specific type that describes checking for zero can be written using ***Dec***.

```
(claim zero?
  (Π ((j Nat))
    (Dec
      (= Nat zero j))))
```

[4] That type says,

"For every Nat j, it is decidable whether j equals zero."

What would count as evidence for that statement?

[5] A function that, given some j, decides whether j equals zero.

If the value of (***zero?*** n) has left at the top, what is tucked under left?

[6] Evidence that n equals zero, because

(***Dec***
  (= Nat zero n))

and

(Either
  (= Nat zero n)
  (→ (= Nat zero n)
    Absurd))

are the same type.

If the value of (*zero?* *n*) has right at the top, what is tucked under right?

[7] Evidence that *n* is not equal to zero.

In other words, the type of *zero?* says that not only does it determine whether a number is zero, it also constructs evidence that it is the correct choice.

[8] *zero?* is a function, so it has λ at the top.

```
(define zero?
  (λ (j)
    [          ]))
```

The empty box's type is

```
(Dec
  (= Nat zero j)).
```

Should it be filled with left or right?

[9] That depends on *j*. Because the empty box's type mentions the target *j*, **ind-Nat** must be used.

What about the motive?

[10] The motive can be found by abstracting over the target.

```
(define zero?
  (λ (j)
    (ind-Nat j
      (λ (k)
        (Dec
          (= Nat zero k)))
      [          ]
      [          ])))
```

Why not abstract over zero in the base's type?

[11] There are two zeros in the base's type, but only one of them is the target.

What is the base? Its type is

```
(Dec
  (= Nat zero zero)).
```

[12] So,

```
(left
  (same zero))
```

does the trick.

What about the step?

[13] The step's type is

```
(Π ((j-1 Nat))
  (→ (Dec
       (= Nat zero j-1))
     (Dec
       (= Nat zero (add1 j-1))))).
```

Is zero ever equal to a Nat with add1 at the top?

[14] No.

Prove that

"zero is not equal to (add1 *j-1*)."

[15] The proof is (***zero-not-add1*** *j-1*).

That's right.

Use this to define the step.

[16] ***zero?*** is not really recursive, so $zero?_{n-1}$ is dim. The proof that (add1 *j-1*) is not equal to zero is tucked under a right because ***Dec*** is defined to mean Either.

```
(define zero?
  (λ (j)
    (ind-Nat j
      (λ (k)
        (Dec
          (= Nat zero k)))
      (left
        (same zero))
      (λ (j-1 zero?_{n-1})
        (right
          (zero-not-add1 j-1))))))
```

---

zero? is both a *proof* that equality with zero is either true or false and a *function* that decides whether any given Nat is equal to zero.

In fact,

"For *every* two natural numbers n and j, it is decidable whether n equals j."

[17] That's a bold claim.

```
(claim nat=?
  (Π ((n Nat)
      (j Nat))
    (Dec
      (= Nat n j))))
```

A claim requires a proof.

```
(define nat=?
  (λ (n j)
    ((ind-Nat n
       [          ]
       [          ]
       [          ])
     j)))
```

[18] That's a strange way to start the proof.

What is the reason that the ind-Nat-expression is applied to j?

The definition of ***front*** uses a more informative motive to make apparent that the base is unnecessary.

[19] Is ***nat=?***'s base also unnecessary?

No, ***nat=?*** needs a base because it makes sense to apply it to zero as an argument.

But a more informative motive is needed here in order to write the step, or else the almost-proof does not prove a strong enough statement.

[20] Please point out where this is necessary.

Gladly.

What is the motive's type?

[21] Every motive used with **ind-Nat** has the same type.

```
(claim mot-nat=?
  (→ Nat
    𝒰))
```

The more informative motive, read as a statement, says

"For every Nat j, it is decidable whether j is equal to the target."

Write this as a function from the target to a type.

[22] The *every* means that there is a Π, and the target is the argument to the motive.

```
(define mot-nat=?
  (λ (k)
    (Π ((j Nat))
      (Dec
        (= Nat k j)))))
```

Compare ***mot-nat=?*** to ***mot-front*** in frame 15:60 on page 330.

[23] Both of them give Π-expressions, so the base and step accept arguments.

These arguments, however, serve different purposes.

The extra arguments in ***front*** are used to make the types more specific to rule out the base. On the other hand, the extra argument in ***nat=?*** is used to make the type more general so that the almost-proofs can decide equalities with *every* Nat, instead of only the second argument to ***nat=?***.

[24] Neither motive is found by just abstracting over some constant, though.

Sometimes, the "motive" is more complicated than just "what" the base is.

Speaking of the base, what is its type?

[25] The base is a

```
(Π ((j Nat))
  (Dec
    (= Nat zero j))).
```

What has that type?

[26] ***zero?*** has that type.

```
(define nat=?
  (λ (n j)
    ((ind-Nat n
       mot-nat=?
       zero?
       [          ])
     j)))
```

The step is still an empty box. What is its type?

[27] For **ind-Nat**, the type of the step is found using the motive.

```
(claim step-nat=?
  (Π ((n-1 Nat))
    (→ (mot-nat=? n-1)
      (mot-nat=? (add1 n-1)))))
```

The types of the step and the motive are determined by the Law of **ind-Nat**. Their definitions, however, may both require insight.

Define ***step-nat=?***.

[28] Here's a start.

```
(define step-nat=?
  (λ (n-1)
    (λ (nat=?n-1)
      (λ (j)
        [          ]))))
```

---

step-nat=?'s type has a Π and an →, but that definition has three λs.

Why is the innermost λ present?

[29] The innermost λ-expression is there because

```
(mot-nat=? (add1 n-1))
```

and

```
(Π ((j Nat))
  (Dec
    (= Nat (add1 n-1) j)))
```

are the same type.

Now it is time to decide whether
(add1 *n-1*) equals *j*.

[30] If *j* is zero, then they are certainly not equal.

Checking whether *j* is zero requires an eliminator.

[31] **ind-Nat** is the only eliminator for Nat that allows the type to depend on the target, and *j* is in the type.

```
(define step-nat=?
  (λ (n-1 nat=?n-1 j)
    (ind-Nat j
      (λ (k)
        (Dec
          (= Nat (add1 n-1) k)))
      [                      ]
      [                      ])))
```

In this definition, the base is much easier than the step. What is the base's type?

[32] The base's type is

(*Dec*
(= Nat (add1 *n-1*) zero)).

The base has right at the top because (add1 *n-1*) certainly does not equal zero.

Prove it.

[33] Prove what?

Prove that
"(add1 *n-1*) does not equal zero."

[34] Again?

---

zero-not-add1 is not a proof that "(add1 *n-1*) does not equal zero."

[35] Ah.

```
(claim add1-not-zero
  (Π ((n Nat))
    (→ (= Nat (add1 n) zero)
      Absurd)))
(define add1-not-zero
  (λ (n)
    (use-Nat= (add1 n) zero)))
```

What is the base?

[36] The base has right at the top, and uses ***add1-not-zero***.

```
(define step-nat=?
  (λ (n-1 nat=?n-1 j)
    (ind-Nat j
      (λ (k)
        (Dec
          (= Nat (add1 n-1) k)))
      (right
        (add1-not-zero n-1))
      [          ])))
```

What is the step's type?

[37] The step is a

```
(Π ((j-1 Nat))
  (→ (Dec
       (= Nat (add1 n-1) j-1))
    (Dec
      (= Nat (add1 n-1) (add1 j-1))))).
```

If ***mot-nat=?*** didn't produce a Π-expression, we would be unable to write the step.

[38] Why is that?

In order to decide whether

(add1 *n-1*) equals (add1 *j-1*),

is it useful to know whether

(add1 *n-1*) equals *j-1*?

[39] 4 does not equal 3, but 4 certainly equals (add1 3).

On the other hand, 4 does not equal 9, but 4 also does not equal (add1 9).

This means that the second argument to this step is dim.

[40] How can the decision be made?

```
(define step-nat=?
  (λ (n-1 nat=?n-1 j)
    (ind-Nat j
      (λ (k)
        (Dec
          (= Nat (add1 n-1) k)))
      (right
        (add1-not-zero n-1))
      (λ (j-1 nat=?n-1)
        [          ]))))
```

---

nat=?$_{n\text{-}1}$ is able to decide whether *n-1* is equal to *any* Nat.

[41] And that is the reason why **mot-nat=?** must have a Π-expression in its body. Otherwise, *nat=?*$_{n\text{-}1}$ would just be a statement about *j* that is unrelated to (add1 *j-1*).

What type does

(*nat=?*$_{n\text{-}1}$ *j-1*)

have?

[42] It is a

(**Dec**
  (= Nat *n-1 j-1*))

but the empty box needs a

(**Dec**
  (= Nat (add1 *n-1*) (add1 *j-1*))).

If we can decide whether

n-1 and *j-1* are equal,

then we can also decide whether

(add1 *n-1*) and (add1 *j-1*) are equal.

```
(claim dec-add1=
  (Π ((n-1 Nat)
      (j-1 Nat))
    (→ (Dec
         (= Nat
           n-1
           j-1))
       (Dec
         (= Nat
           (add1 n-1)
           (add1 j-1))))))
```

[43] If *n-1* equals *j-1*, then **cong** can make (add1 *n-1*) equal (add1 *j-1*). And if they are not equal, then working backwards with ***sub1=*** is enough to be Absurd.

Checking both cases means **ind-Either**.

Start the definition.

[44] The motive in **ind-Either** ignores its argument because the type does not depend on the target.

```
(define dec-add1=
  (λ (n-1 j-1 eq-or-not)
    (ind-Either
      (λ (target)
        (Dec
          (= Nat
            (add1 n-1)
            (add1 j-1))))
      [                    ]
      [                    ])))
```

What goes in the first empty box?

[45] The first empty box needs an

```
(→ (= Nat n-1 j-1)
  (Dec
    (= Nat (add1 n-1) (add1 j-1)))).
```

That's the type.

What about the contents of the box?

[46] The left is used because the answer is still, "Yes, they're equal." And **cong** with (**+** 1) transforms evidence that

"*n-1* equals *j-1*"

into evidence that

"(add1 *n-1*) equals (add1 *j-1*)."

The box should contain

```
(λ (yes)
  (left
    (cong yes (+ 1)))).
```

Indeed. What goes in the second box?

[47] The second box's type is

```
(→ (→ (= Nat n-1 j-1)
      Absurd)
   (Dec
     (= Nat (add1 n-1) (add1 j-1)))).
```

The contents of that second box will have right at the top. Why?

[48] Because if *n-1* and *j-1* are not equal, then (add1 *n-1*) and (add1 *j-1*) are also not equal.

In that box, right requires an

```
(→ (= Nat (add1 n-1) (add1 j-1))
   Absurd).
```

That box has a variable available named *no*, with type

```
(→ (= Nat n-1 j-1)
   Absurd).
```

[49] *no* proves Absurd when its argument is an

```
(= Nat n-1 j-1),
```

which can be found using ***sub1=*** like this:

```
(λ (n=j)
  (no (sub1= n-1 j-1 n=j))).
```

Now complete the definition.

[50] ***dec-add1=*** is a bit long.

```
(define dec-add1=
  (λ (n-1 j-1 eq-or-not)
    (ind-Either eq-or-not
      (λ (target)
        (Dec
          (= Nat (add1 n-1) (add1 j-1))))
      (λ (yes)
        (left
          (cong yes (+ 1))))
      (λ (no)
        (right
          (λ (n=j)
            (no
              (sub1= n-1 j-1
                n=j)))))))))
```

Finish ***step-nat=?***.

[51] Here it is.

```
(define step-nat=?
  (λ (n-1 nat=?n-1 j)
    (ind-Nat j
      (λ (k)
        (Dec
          (= Nat (add1 n-1) k)))
      (right
        (add1-not-zero n-1))
      (λ (j-1 nat=?n-1)
        (dec-add1= n-1 j-1
          (nat=?n-1 j-1))))))
```

Now that the motive, the base, and the step are completed for ***nat=?***, it can be given a solid box.

[52] It is decidable whether two natural numbers are equal.

```
(define nat=?
  (λ (n j)
    ((ind-Nat n
       mot-nat=?
       zero?
       step-nat=?)
     j)))
```

Just like ***even-or-odd***, ***nat=?*** is both a *proof* that makes a statement true and a *function* that determines whether any two numbers are equal. Because ***nat=?*** is total and because it provides *evidence*, there is no way that it can find the wrong value.

[53] Why was there no food in this chapter?

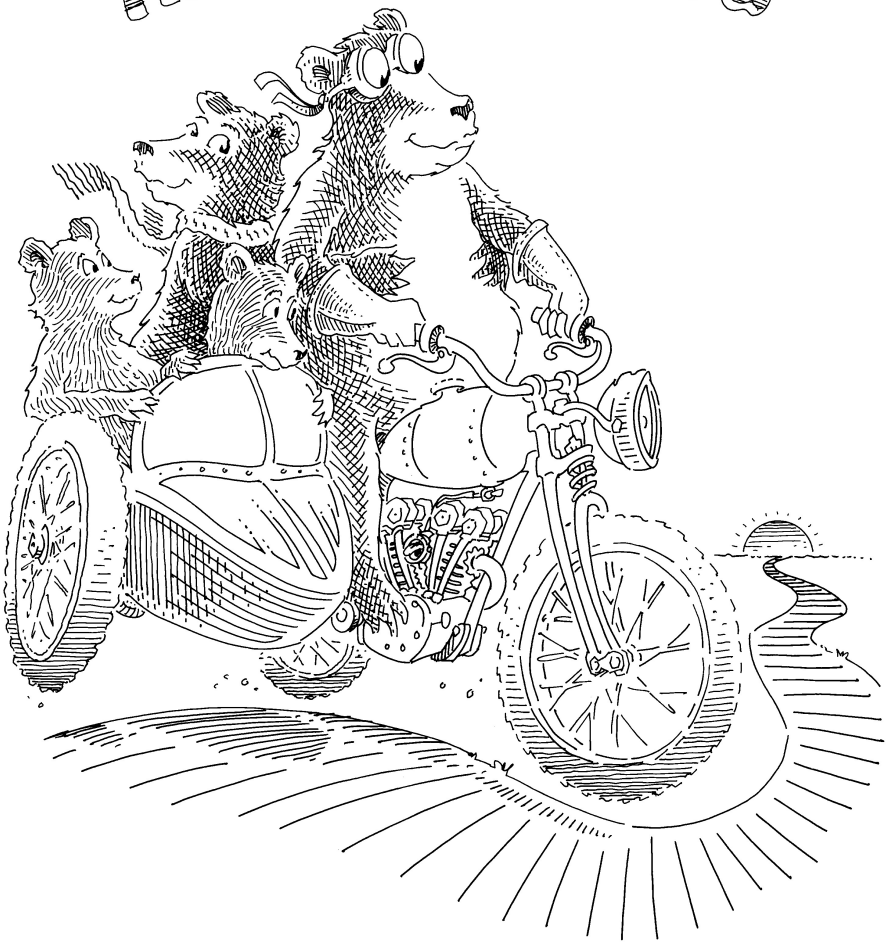
Numbers nourish our minds, not our bodies.

[54] But a weak body leads to a weak mind.

Go enjoy a banquet

you've earned it!

The Way Forward



Pie is a small language—small enough to be understood completely. Now, it may be time to continue with more sophisticated dependently typed languages.[1]

In addition to type constructors like Π and Σ, these languages include five extensions: infinitely many universes, the ability to define new type constructors and their associated data constructors, the ability to define functions through pattern matching, the ability to leave out expressions that the language can find on its own, and tactics for automating proof construction.

A Universe Hierarchy

In Pie, there is a single universe type, called $\mathcal{U}$. While $\mathcal{U}$ is a type, $\mathcal{U}$ does not describe itself nor any type that can contain a $\mathcal{U}$, such as (List $\mathcal{U}$). While more universes are not needed for any of the examples in this book, it is sometimes necessary to have a type that describes $\mathcal{U}$ (and sometimes even a type that describes the type that describes $\mathcal{U}$). By including infinitely many universes, each describing the previous ones, more sophisticated languages ensure that there are always sufficient universes to solve each problem.

Inductive Datatypes

Some types that one might propose do not make sense. Restricting Pie to a fixed collection of types ensures that no type can undermine the system as a whole. Some problems, however, cannot be easily expressed using the tools in this book. More sophisticated languages allow for adding new datatype type constructors.[2] These new types are called *inductive datatypes* because their eliminators express the mathematical idea of induction.

If Pie did not already feature lists, then adding them could require the following declaration: if E is a $\mathcal{U}$, then (List E) is a $\mathcal{U}$. In addition, there are two constructors: nil, which is a (List E), and ::, which needs two arguments. The name for an eliminator is also needed. The Laws and Commandments for the eliminator are based on the provided constructors.

```
(data List ((E U)) ()
  (nil ()
    (List E))
  (:: ((e E) (es (List E)))
    (List E))
  ind-List)
```

These new inductive datatypes might have both parameters, which do not vary between the constructors, and indices, which can vary between them (as discussed in frame 11:14). For Vec, the first argument is a parameter, while the length varies between vec:: and vecnil.

```
(data Less-Than () ((j Nat) (k Nat))
  (zero-smallest ((n Nat))
    (Less-Than zero (add1 n)))
  (add1-smaller ((j Nat)
                 (k Nat)
                 (j<k (Less-Than j k)))
    (Less-Than (add1 j) (add1 k)))
  ind-Less-Than)
```

As an example of an indexed family, the datatype Less-Than is evidence that one number is smaller than another. Because the constructors impose different values on

[1] Examples include Coq, Agda, Idris, and Lean.
[2] Thanks, Peter Dybjer (1953–).

each Nat, the Nats are indices. The Law of **ind-Less-Than** follows a pattern that should be familiar from other types: if *target* is a

(Less-Than *j k*),

mot is a

(Π ((*j* Nat)
(*k* Nat))
(→ (Less-Than *j k*) *𝒰*)),

base is a

(Π ((*k* Nat)
(*lt* (Less-Than zero (add1 *k*))))
(*mot* zero *k lt*)),

and *step* is a

(Π ((*j* Nat)
(*k* Nat)
(*j*<*k* (Less-Than *j k*)))
(→ (*mot j k j*<*k*)
(*mot* (add1 *j*) (add1 *k*)
(add1-smaller *j k j*<*k*)))),

then (**ind-Less-Than** *target mot base step*) is a (*mot j k target*).

The ability to define new datatypes makes it much more convenient to do complicated things in these other languages. Furthermore, using eliminators directly, as we have in Pie, is not particularly convenient for larger problems.

Recursive Functions with Pattern Matching

The basic principle of eliminators is that for each constructor, we need to explain what must be done to satisfy the motive using the information inside the constructor. Recursion is made safe by having each eliminator be responsible for ensuring that recursive computation is performed only on smaller values.

[3]Thanks, Thierry Coquand (1961–).

An alternative way to define functions is with *pattern matching* and a safe form of recursion.[3] More sophisticated languages also allow recursive functions to be defined by directly explaining what action to take with each possible value. For instance, ***length*** could have been written as follows:

```
(claim length
  (Π ((E Nat))
    (→ (List E) Nat)))
(define length
  (λ (E es)
    (match es
      (nil zero)
      ((:: x xs) (add1 (length xs))))))
```

While recursion is not an option in Pie, sophisticated languages have additional checks to ensure that recursion is only used safely, and can thus allow it.

While ***front***'s definition in frame 15:74 requires a more informative motive to rule out the vecnil case, as well as extra arguments to satisfy the motive, a definition with pattern matching is more direct. Not only does it work, but it is also more understandable and more compact.

```
(claim front
  (Π ((E 𝒰)
      (n Nat))
    (→ (Vec E (add1 n)) E)))
(define front
  (λ (E n es)
    (match es
      ((vec:: x xs) x))))
```

Sometimes, we only care *that* we have evidence for a statement, not *which* evidence it is. In such situations, writing the evidence out explicitly is not always appealing—especially when that evidence

consumes many pages. Truly verbose evidence can even require a whole bookshelf, while being repetitive and tedious rather than pithy and interesting.

Implicit Arguments

Programs written with dependent types have a tendency to grow quickly. For instance, ***length*** requires not only a list, but also the type of entries in that list, and ***vec-append*** requires the type of entries *and* the respective lengths of the vectors being appended. This information, however, is already available in the types of later arguments, so it would be convenient to be able to omit some of it.

More sophisticated languages provide a mechanism called *implicit* or *hidden* arguments.[4] These arguments are to be discovered by the system, rather than the responsibility of the user.

Pie could be extended with implicit arguments. One way to do this would be to add three new syntactic forms:

1. an implicit Π, say Π∗, that works just like the ordinary Π, except that its arguments are marked implicit,

2. an implicit λ, say λ∗, that works just like the ordinary λ, except that its arguments are marked implicit, and

3. an implicit function application, say **implicitly**, that marks its arguments as filling an implicit rather than explicit role.

With these features, ***length*** could be written so that the type of entries is hidden, and automatically discovered.

```
(claim length
  (Π* ((E U))
    (→ (List E) Nat)))
(define length
  (λ* (E)
    (λ (es)
      (rec-List es
        0
        (λ (e es ℓ)
          (add1 ℓ))))))
```

Then, the expression

```
(length (:: 'potato (:: 'gravy nil)))
```

would be the equivalent of having written

```
(length Atom (:: 'potato (:: 'gravy nil)))
```

in Pie using the definition of ***length*** from chapter 5. Similarly,

```
(implicitly length Atom)
```

would be an

```
(→ (List Atom) Nat).
```

Implicit arguments allow definitions to be just as concise as the built-in constructors and eliminators.

Proof Tactics

Here is another way to define ***even-or-odd***. Instead of directly constructing the evidence that every natural number is either even or odd, this version uses *proof tactics*[5] to automate the definition.

A tactic is a program in a special language that is provided with a desired type (called a *goal*) that either succeeds with zero or more new goals or fails. Further tactics can then be deployed to solve these new goals until all tactics have succeeded with no remaining goals. Then, evidence for the original goal is the result of the

[4] Thanks, Randy Pollack (1947–).

[5] Thank you, Robin Milner (1934–2010).

tactic program. If Pie had tactics, then evidence for ***even-or-odd*** could be constructed with a tactic script instead of being written as an expression.

```
(claim even-or-odd
  (Π ((n Nat))
    (Either (Even n) (Odd n))))
(define-tactically even-or-odd
  (intro n)
  (elim n)
  (apply zero-is-even)
  (intro n-1 e-or-o_n-1)
  (elim e-or-o_n-1)
  (then
    right
    (apply add1-even→odd)
    auto)
  (then
    left
    (apply add1-odd→even)
    auto))
```

Here, intro is a tactic that succeeds when the goal type has Π at the top, binding the name given as an Atom using λ. elim uses an appropriate eliminator, here **ind-Nat** and **ind-Either**, respectively. apply uses an expression to solve the goal, but leaves behind new goals for each argument needed by the expression. then causes each tactic in sequence to be used in all of the new goals from the preceding tactic. When used as tactics, right and left succeed when the goal has Either at the top, and provide Either's respective argument types as new goals. auto takes care of simple evidence completely on its own. The result of these tactics is the same as the ***even-or-odd*** defined in chapter 13.

Tactics can be combined to create new tactics, which allows even very complicated and tedious evidence to be constructed using very small programs. Furthermore, it is possible to write one tactic that can solve many different goals, allowing it to be used again and again.

Each sophisticated language for programming and proving has some mix of the useful, yet more complicated, features described here. Do not be concerned—while these languages have features that make programs easier to write, the underlying ideas are the familiar ideas from Pie. We wish you the best in your further exploration of dependent types.

Some Books You May Love

Flatland: A Romance of Many Dimensions
by Edwin A. Abbott. Seeley & Co. of London, 1884.

Gödel's Proof
by Ernest Nagel and James R. Newman. NYU Press, 1958.

Grooks
by Piet Hein. MIT Press, 1966.

Gödel, Escher, Bach: An Eternal Golden Braid
by Douglas R. Hofstadter. Basic Books, 1979.

To Mock a Mockingbird and Other Puzzles
by Raymond Smullyan. Knopf, 1985.

Sophie's World: A Novel About the History of Philosophy
by Jostein Gaarder. Farrar Straus Giroux, 1995.

Logicomix
by Apostolos Doxiadis, Christos H. Papadimitriou, Alecos Papadatos, and Annie Di Donna. Bloomsbury USA, 2009.

Computation, Proof, Machine: Mathematics Enters a New Age
by Gilles Dowek. Cambridge University Press, 2015.

Rules Are Made to Be Spoken

This appendix is for those who have some background in the theory of programming languages who want to compare Pie to other languages or who want to implement Pie from scratch. Three good books that can be used to get this background are Harper's *Practical Foundations for Programming Languages*, Pierce's *Types and Programming Languages*, and Felleisen, Findler, and Flatt's *Semantics Engineering with PLT Redex*.

When implementing dependent types, there are two questions to be answered: *when* to check for sameness, and *how* to check for sameness. Our implementation of Pie uses bidirectional type checking (described in the section **Forms of Judgment**) to decide when, and normalization by evaluation (described in the section **Normalization**) as the technique for checking sameness.

Forms of Judgment

While Pie as described in the preceding chapters is a system for guiding human judgment, Pie can also be implemented in a language like Scheme. In an implementation, each form of judgment corresponds to a function that determines whether a particular judgment is believable by the Laws and Commandments. To make this process more straightforward, implementations of Pie have additional forms of judgment.

Although chapter 1 describes four forms of judgment, this appendix has additional details in order to precisely describe Pie's implementation. In the implementation, expressions written in the language described in the preceding chapters are simultaneously checked for validity and translated into a simpler core language. Elaboration into Core Pie can be seen as similar to macro expansion of Scheme programs.

Only the simpler core language is ever checked for sameness. The complete grammars of Pie and Core Pie are at the end of the appendix, on pages 392 and 393. When the distinction between them is important, e is used to stand for expressions written in Pie and c is used to stand for expressions written in Core Pie.

The forms of judgment for implementations of Pie are listed in figure B.1. When a form of judgment includes the bent arrow $\rightsquigarrow$, that means that the expression following the arrow is output from the elaboration algorithm. All contexts and expressions that precede the arrow are input to the elaboration algorithm, while those after the arrow

Γ **ctx**	Γ is a context.
$\Gamma \vdash$ **fresh** $\rightsquigarrow x$	Γ does not bind x.
$\Gamma \vdash x$ **lookup** $\rightsquigarrow c_t$	Looking up x in Γ yields the type c_t.
$\Gamma \vdash e_t$ **type** $\rightsquigarrow c_t$	e_t represents the type c_t.
$\Gamma \vdash c_1 \equiv c_2$ **type**	c_1 and c_2 are the same type.
$\Gamma \vdash e \in c_t \rightsquigarrow c_e$	Checking that e can have type c_t results in c_e.
$\Gamma \vdash e$ **synth** $\rightsquigarrow$ (`the` c_t c_e)	From e, the type c_t can be synthesized, resulting in c_e.
$\Gamma \vdash c_1 \equiv c_2 : c_t$	c_1 is the same c_t as c_2.

Figure B.1: Forms of Judgment

are output. When there is no $\rightsquigarrow$ in a form of judgment, then there is no interesting output, and the judgment's program can only succeed or fail.

When a form of judgment includes a turnstile $\vdash$, the position before the turnstile is a *context*. Contexts assign types to free variables. In Pie, the order of the variables listed in a context matters because a type may itself refer to variables from earlier in the context. Contexts are represented by the variable Γ,[1] and are described by the following grammar:

$$\begin{array}{rcll} \Gamma & ::= & \bullet & \text{Empty context} \\ & | & \Gamma, x : c_t & \text{Context extension} \end{array}$$

In Scheme, contexts can be represented by association lists that pair variables with their types.

Forms of judgment occur within *inference rules*. An inference rule consists of a horizontal line. Below the line is a *conclusion*, and above the line is any number of *premises*. The premises are either written next to each other or on top of each other, as in figure B.2. The meaning of the rule is that, if one believes in the premises, then one should also believe in the conclusion. Because the same conclusion can occur in multiple rules, belief in the premises cannot be derived from belief in the conclusion. Each rule has a name, written in Small Caps to the right of the rule.

$$\frac{\text{premise}_0 \quad \ldots \quad \text{premise}_n}{\text{conclusion}}\ [\textsc{Name}] \qquad\qquad \frac{\begin{array}{c}\text{premise}_0 \\ \vdots \\ \text{premise}_n\end{array}}{\text{conclusion}}\ [\textsc{Name}]$$

Figure B.2: Inference Rules

When reading the rules as an algorithm, each form of judgment should be implemented as a function. When an expression occurs in input position in the conclusion of an inference rule, it should be read as a *pattern* to be matched against the input. When it is in output position, it should be read as *constructing* the result of the algorithm. When an expression occurs in an input position in a premise, it represents input being constructed for a recursive call, and when it occurs in the output position in a premise, it represents a pattern to be matched against the result returned from the recursive call. Italic variables in patterns are bound when a pattern matches, and italic variables in a construction are occurrences bound by patterns, in a manner similar to quasiquotation in Scheme. If any of the patterns do not match, type checking should *fail* because the rule is not relevant. If all the patterns match, type checking should *succeed*, returning the constructed result after the bent arrow. If there is no bent arrow, then type checking should indicate success by returning a trivial value, such as the empty list in Scheme or the element of the unit type in some other language.

[1] Γ is pronounced "gamma."

Γ **ctx**	None
$\Gamma \vdash$ **fresh** $\rightsquigarrow x$	Γ is a context.
$\Gamma \vdash x$ **lookup** $\rightsquigarrow c_t$	Γ is a context.
$\Gamma \vdash e_t$ **type** $\rightsquigarrow c_t$	Γ is a context.
$\Gamma \vdash c_1 \equiv c_2$ **type**	Γ is a context, and c_1 and c_2 are both types.
$\Gamma \vdash e \in c_t \rightsquigarrow c_e$	Γ is a context and c_t is a type.
$\Gamma \vdash e$ **synth** $\rightsquigarrow$ (`the` c_t c_e)	Γ is a context.
$\Gamma \vdash c_1 \equiv c_2 : c_t$	Γ is a context, c_t is a type, c_1 is a c_t, and c_2 is a c_t.

Figure B.3: Presuppositions

Each form of judgment has presuppositions that must be believed before it makes sense to entertain a judgment. In a type checking algorithm, presuppositions are aspects of expressions that should have already been checked before they are provided as arguments to the type checking functions. The presuppositions of each form of judgment are in figure B.3.

When matching against a concrete expression in a rule, the algorithm must reduce the expression enough so that if it doesn't match, further reduction cannot make it match. Finding a neutral expression or a value that is the same as the expression being examined is sufficient. A concrete implementation can do this by matching against the values used in normalization rather than against syntax that represents these values. This also provides a convenient way to implement substitution by instantiating the variable from a closure instead of manually implementing capture-avoiding substitution.

	Input	Output
Conclusion	Pattern	Construction
Premise	Construction	Pattern

There are two putative rules that govern Γ **ctx**: EmptyCtx and ExtCtx.

$$\frac{}{\bullet \textbf{ ctx}}\ [\textsc{EmptyCtx}] \qquad \frac{\Gamma \textbf{ ctx} \qquad \Gamma \vdash c_t \equiv c_t \textbf{ type}}{\Gamma, x : c_t \textbf{ ctx}}\ [\textsc{ExtCtx}]$$

Rather than repeatedly checking that all contexts are valid, however, the rest of the rules are designed so that they never add a variable and its type to the context unless the type actually is a type in that context. This maintains the invariant that contexts contain only valid types. Thus, Γ **ctx** need not have a corresponding function in an implementation.

From time to time, elaboration must construct a variable that does not conflict with any other variable that is currently bound. This is referred to as finding a *fresh* variable and is represented as a form of judgment $\Gamma \vdash$ **fresh** $\rightsquigarrow x$. This form of judgment can either be implemented using a side-effect such as Lisp's `gensym` or by repeatedly modifying a name until it is no longer bound in Γ.

Because the algorithmic system Pie is defined using elaboration that translates Pie into Core Pie, it does not make sense to ask whether a Core Pie expression is a type

or has a particular type. This is because the translation from Pie to Core Pie happens as part of checking the original Pie expression, so the input to the elaboration process is Pie rather than Core Pie.[2] The rules of sameness have been designed such that only expressions that are described by a type are considered the same, and only types are considered to be the same type. This means that sameness judgments can be used to express that one expression describes another, or that an expression is a type. An example of this approach can be seen in ExtCtx, where c_t being a type under Γ is expressed by requiring that it be the same type as itself under Γ.

Normalization

The process of checking whether the judgments $\Gamma \vdash c_1 \equiv c_2$ **type** and $\Gamma \vdash c_1 \equiv c_2 : c_t$ are believable is called *conversion checking*. To check for conversion, the Pie implementation uses a technique called *normalization by evaluation*,[3] or NbE for short. The essence of NbE is to define a notion of *value* that represents only the normal forms of the language, and then write an interpreter from Core Pie syntax into these values. This process resembles writing a Scheme interpreter, as is done in chapter 10 of *The Little Schemer*. Then, the value's type is analyzed to determine what the normal form should look like, and the value itself is converted back into syntax. Converting a value into its normal form is called *reading back* the normal form from the value.

The notion of value used in NbE is related to the notion of value introduced in chapter 1, but it is not the same. In NbE, values are mathematical objects apart from the expressions of Pie or Core Pie, where the results of computation cannot be distinguished from incomplete computations. Examples of suitable values include the untyped λ-calculus, Scheme functions and data, or explicit closures.

Evaluation and reading back are arranged to always find normal forms. This means that the equality judgments can be decided by first normalizing the expressions being compared and then comparing them for α-equivalence. While the typing rules are written as though they use only the syntax of the surface and core languages, with capture-avoiding substitution to instantiate variables, an actual implementation can maintain closures to represent expressions with free variables, and then match directly on the values of types rather than substituting and normalizing.

Here, we do not specify the precise forms of values, nor the full normalization procedure. Indeed, any conversion-checking technique that respects the Commandments for each type, including the η-rules, is sufficient. Additionally, there are ways of comparing expressions for sameness that do not involve finding normal forms and comparing them. The Commandments are given here as a specification that the conversion algorithm should fulfill. See Andreas Abel's habilitation thesis *Normalization by Evaluation: Dependent Types and Impredicativity* for a complete description of NbE.

[2] It would be possible to write a separate type checker for Core Pie, but this is not necessary.

[3] Thanks, Ulrich Berger (1956–), Helmut Schwichtenberg (1942–), and Andreas Abel (1974–).

The Rules

The rules use *italic* letters to stand for arbitrary expressions, and letters are consistently assigned based on the role played by the expression that the letter stands for. Letters that stand for other expressions are called *metavariables*. Please consult figure B.1 to see which positions are written in which language, and figure B.4 to see what each metavariable stands for.

When one metavariable stands for the result of elaborating another expression, the result has a lower-case letter o (short for *output*) as a superscript. So b^o is the result of elaborating an expression b. When the same metavariable occurs multiple times in a rule, each occurrence stands for identical expressions; if there are multiple metavariables that play the same role, then they are distinguished via subscripts. Sometimes, subscripts indicate a sequence such as $x_1 \ldots x_n$. Otherwise, the subscripts 1 and 2 or 3 and 4 are used for expressions that are expected to be the same. Even though two metavariables have different subscripts, they may nevertheless refer to the same expression; the subscripts allow them to be different but do not require them to be different.

The most basic rules are those governing the interactions between checking and synthesis. Changing from checking to synthesis requires an equality comparison, while changing from synthesis to checking requires an annotation to check against.[4] Annotations are the same as the annotated expression.

$$\frac{\Gamma \vdash X \ \textbf{type} \rightsquigarrow X^o \qquad \Gamma \vdash \mathit{expr} \in X^o \rightsquigarrow \mathit{expr}^o}{\Gamma \vdash (\texttt{the}\ X\ \mathit{expr}) \ \textbf{synth} \rightsquigarrow (\texttt{the}\ X^o\ \mathit{expr}^o)}\ [\textsc{The}]$$

$$\frac{\Gamma \vdash \mathit{expr} \ \textbf{synth} \rightsquigarrow (\texttt{the}\ X_1\ \mathit{expr}^o) \qquad \Gamma \vdash X_1 \equiv X_2 \ \textbf{type}}{\Gamma \vdash \mathit{expr} \in X_2 \rightsquigarrow \mathit{expr}^o}\ [\textsc{Switch}]$$

To read these rules aloud, take a look at the labeled copy of The below. Start below the line, in the conclusion, and identify the form of judgment. In this case, it is type synthesis. Begin at the position labeled A. If the input matches (that is, if the current task is to synthesize a type for a `the`-expression), proceed to the premises. Identify the form of judgment used in the first premise B: that X is a type. Checking that X is a type yields a Core Pie expression X^o as output, at position C. This Core Pie expression is used as input to the next premise, at position D, which checks that *expr* is an X^o, yielding an elaborated Core Pie version called expr^o at position E. Finally, having satisfied all of the premises, the result of the rule is constructed at position F.

$$\frac{\text{B}\ \Gamma \vdash X \ \textbf{type} \rightsquigarrow \text{C}\ X^o \qquad \text{D}\ \Gamma \vdash \mathit{expr} \in X^o \rightsquigarrow \text{E}\ \mathit{expr}^o}{\text{A}\ \Gamma \vdash (\texttt{the}\ X\ \mathit{expr}) \ \textbf{synth} \rightsquigarrow \text{F}\ (\texttt{the}\ X^o\ \mathit{expr}^o)}\ [\textsc{The}]$$

[4] Thanks, Benjamin C. Pierce (1963–) and David N. Turner (1968–).

A `the`-expression is the same as its second argument. Try reading this rule aloud.

$$\frac{\Gamma \vdash \mathit{expr}_1 \equiv \mathit{expr}_2 : X}{\Gamma \vdash (\texttt{the}\ X\ \mathit{expr}_1) \equiv \mathit{expr}_2 : X}\ [\textsc{TheSame}]$$

Aside from [The], [Switch], and one of the rules for $\mathcal{U}$, the rules fall into one of a few categories:

1. *formation rules*, which describe the conditions under which an expression is a type;
2. *introduction rules*, which describe the constructors for a type;
3. *elimination rules*, which describe the eliminators for a type;
4. *computation rules*, which describe the behavior of eliminators whose targets are constructors;
5. *η-rules*, which describe how to turn neutral expressions into values for some types; and
6. *other sameness rules*, which describe when sameness of subexpressions implies sameness of whole expressions.

Formation, introduction, and elimination rules correspond to the Laws, while the remaining rules correspond to the Commandments. The names of rules begin with an indication of which family of types they belong to. For instance, rules about Atom begin with Atom, and rules about functions begin with Fun. Formation, introduction, and elimination rules then have an F, I, or E, respectively. Computation rules include the letter ι (pronounced "iota") in their names, with the exception of [FunSame-β] and [TheSame]. The η-rules contain η in their names, and the other sameness rules are named after the syntactic form at the top of their expressions.

Sameness

Sameness is a partial equivalence relation; that is, it is symmetric and transitive. Additionally, the rules are arranged such that, for each type, the expressions described by that type are the same as themselves. It is important to remember that rules whose conclusions are sameness judgments are *specifications* for a normalization algorithm, rather than a description of the algorithm itself. Algorithms for checking sameness do not typically include rules such as [SameSymm] on page 370 because it could be applied an arbitrary number of times without making progress.

Meta	Role	Mnemonic
a	**car** of a pair	c*a*r
A	Type of **car** of a pair	C*A*R
arg	Argument to a function	*arg*ument
Arg	Type of argument to a function	*Arg*ument
b	Base	b is for base
B	Type of base	B is for base
d	**cdr** of a pair	c*d*r
D	Type of **cdr** of a pair	C*D*R
e	Entry in a list or vector	e is for entry
E	Type of entries in a list or vector	*E*NTRY
es	Entries in a list or vector	Plural of e
expr	Any expression	*expr* is an expression
from	FROM	
f	A function expression	*f*unction
ℓ	Length of Vec	ℓ is for length
lt	Evidence for left type in Either	*lt* is short for *left*
mid	Middle of transitivity	
m	Motive	m is for motive
n	A natural number	n is for natural
P	Left type in Either	The *P*ort is on the left
pr	A pair	*pr* is for pair
r	Result of a function	r is for result
R	Type of result of a function	R is the type of the result
rt	Evidence for right type in Either	*rt* is short for *right*
s	Step	s is for step
S	Right type in Either	The *S*tarboard is on the right
t	Target	t is for target
to	TO	
x, y	Variable names	x and y are frequently unknown
X, Y, Z	Any type	X, Y, or Z can be any type

Figure B.4: Metavariables

$$\frac{\Gamma \vdash \mathit{expr}_2 \equiv \mathit{expr}_1 : X}{\Gamma \vdash \mathit{expr}_1 \equiv \mathit{expr}_2 : X}\ [\textsc{SameSymm}]$$

$$\frac{\Gamma \vdash \mathit{expr}_1 \equiv \mathit{expr}_2 : X \qquad \Gamma \vdash \mathit{expr}_2 \equiv \mathit{expr}_3 : X}{\Gamma \vdash \mathit{expr}_1 \equiv \mathit{expr}_3 : X}\ [\textsc{SameTrans}]$$

Variables

The form of judgment with fewest rules is $\Gamma \vdash x\ \mathbf{lookup} \rightsquigarrow c_t$. It has two rules: LookupStop and LookupPop.

$$\frac{}{\Gamma, x : X \vdash x\ \mathbf{lookup} \rightsquigarrow X}\ [\textsc{LookupStop}]$$

$$\frac{x \neq y \qquad \Gamma \vdash x\ \mathbf{lookup} \rightsquigarrow X}{\Gamma, y : Y \vdash x\ \mathbf{lookup} \rightsquigarrow X}\ [\textsc{LookupPop}]$$

Read aloud, LookupStop says:

> To look up x in a context $\Gamma, x : X$, succeed with X as a result.

and LookupPop says:

> To look up x in a context $\Gamma, y : Y$, make sure that x and y are not the same name, and then recursively look up x in Γ.

Together, these rules describe looking up a name in an association list using Scheme's `assoc` to find a name-type pair. Looking up a variable is used in the rule Hypothesis, which describes how to synthesize a type for a variable.

$$\frac{\Gamma \vdash x\ \mathbf{lookup} \rightsquigarrow X}{\Gamma \vdash x\ \ \mathbf{synth} \rightsquigarrow (\texttt{the}\ X\ x)}\ [\textsc{Hypothesis}]$$

To read Hypothesis aloud, say:

> To synthesize a type for a variable x, look it up in the context Γ. If the lookup succeeds with type X, synthesis succeeds with the Core Pie expression (`the` X x).

The conclusion of HypothesisSame rule below is a judgment of sameness, so it is a specification for the normalization algorithm.

$$\frac{\Gamma \vdash x\ \mathbf{lookup} \rightsquigarrow X}{\Gamma \vdash x \equiv x : X}\ [\textsc{HypothesisSame}]$$

HypothesisSame says:

> If a variable x is given type X by the context Γ, then conversion checking must find that x is the same X as x.

As you read the rest of this appendix, remember to read the rules aloud to aid understanding them.

Atoms

In these rules, the syntax $\lceil sym \rceil$ stands for a literal Scheme symbol that satisfies the definition of atoms in chapter 1: namely, that they consist of a non-empty sequence of letters and hyphens.

$$\frac{}{\Gamma \vdash \mathsf{Atom}\ \ \mathbf{type} \rightsquigarrow \mathsf{Atom}}\ [\textsc{AtomF}] \qquad \frac{}{\Gamma \vdash \mathsf{Atom} \equiv \mathsf{Atom}\ \ \mathbf{type}}\ [\textsc{AtomSame-Atom}]$$

$$\frac{}{\Gamma \vdash \text{'}\lceil sym \rceil\ \ \mathbf{synth} \rightsquigarrow (\texttt{the}\ \mathsf{Atom}\ \text{'}\lceil sym \rceil)}\ [\textsc{AtomI}]$$

$$\frac{}{\Gamma \vdash \text{'}\lceil sym \rceil \equiv \text{'}\lceil sym \rceil : \mathsf{Atom}}\ [\textsc{AtomSame-tick}]$$

Pairs

$$\frac{\Gamma \vdash A\ \ \mathbf{type} \rightsquigarrow A^o \qquad \Gamma, x : A^o \vdash D\ \ \mathbf{type} \rightsquigarrow D^o}{\Gamma \vdash (\Sigma\ ((x\ A))\ D)\ \ \mathbf{type} \rightsquigarrow (\Sigma\ ((x\ A^o))\ D^o)}\ [\Sigma\text{F-1}]$$

$$\frac{\begin{array}{l}\Gamma \vdash A\ \ \mathbf{type} \rightsquigarrow A^o \\ \Gamma, x : A^o \vdash (\Sigma\ ((x_1\ A_1)\ \ldots\ (x_n\ A_n))\ D)\ \ \mathbf{type} \rightsquigarrow X\end{array}}{\Gamma \vdash (\Sigma\ ((x\ A)\ (x_1\ A_1)\ \ldots\ (x_n\ A_n))\ D)\ \ \mathbf{type} \rightsquigarrow (\Sigma\ ((x\ A^o))\ X)}\ [\Sigma\text{F-2}]$$

$$\frac{\Gamma \vdash A \ \ \mathbf{type} \rightsquigarrow A^o \qquad \Gamma \vdash \mathbf{fresh} \rightsquigarrow x \qquad \Gamma, x : A^o \vdash D \ \ \mathbf{type} \rightsquigarrow D^o}{\Gamma \vdash (\mathsf{Pair}\ A\ D) \ \ \mathbf{type} \rightsquigarrow (\Sigma\ ((x\ A^o))\ D^o)}\ [\Sigma\text{F-Pair}]$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 \ \ \mathbf{type} \qquad \Gamma, x : A_1 \vdash D_1 \equiv D_2 \ \ \mathbf{type}}{\Gamma \vdash (\Sigma\ ((x\ A_1))\ D_1) \equiv (\Sigma\ ((x\ A_2))\ D_2) \ \ \mathbf{type}}\ [\Sigma\textsc{Same-}\Sigma]$$

The second premise in ΣI below contains the expression $D[a^o/x]$. The brackets mean that *capture-avoiding substitution* should be used to consistently replace every x in D with a^o. This can be implemented by using values with closures rather than explicit substitution.

$$\frac{\Gamma \vdash a \in A \rightsquigarrow a^o \qquad \Gamma \vdash d \in D[a^o/x] \rightsquigarrow d^o}{\Gamma \vdash (\mathsf{cons}\ a\ d) \in (\Sigma\ ((x\ A))\ D) \rightsquigarrow (\mathsf{cons}\ a^o\ d^o)}\ [\Sigma\text{I}]$$

Try reading ΣI aloud.

$$\frac{\Gamma \vdash a_1 \equiv a_2 : A \qquad \Gamma \vdash d_1 \equiv d_2 : D[a_1/x]}{\Gamma \vdash (\mathsf{cons}\ a_1\ d_1) \equiv (\mathsf{cons}\ a_2\ d_2) : (\Sigma\ ((x\ A))\ D)}\ [\Sigma\textsc{Same-}\mathsf{cons}]$$

$$\frac{\Gamma \vdash pr \ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (\Sigma\ ((x\ A))\ D)\ pr^o)}{\Gamma \vdash (\mathsf{car}\ pr) \ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ A\ (\mathsf{car}\ pr^o))}\ [\Sigma\text{E-1}]$$

$$\frac{\Gamma \vdash pr_1 \equiv pr_2 : (\Sigma\ ((x\ A))\ D)}{\Gamma \vdash (\mathsf{car}\ pr_1) \equiv (\mathsf{car}\ pr_2) : A}\ [\Sigma\textsc{Same-}\mathsf{car}]$$

$$\frac{\Gamma \vdash a_1 \equiv a_2 : A \qquad \Gamma, x : A \vdash d \equiv d : D}{\Gamma \vdash (\mathsf{car}\ (\mathsf{cons}\ a_1\ d)) \equiv a_2 : A}\ [\Sigma\textsc{Same-}\iota 1]$$

$$\frac{\Gamma \vdash pr \ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (\Sigma\ ((x\ A))\ D)\ pr^o)}{\Gamma \vdash (\mathsf{cdr}\ pr) \ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ D[(\mathsf{car}\ pr^o)/x]\ (\mathsf{cdr}\ pr^o))}\ [\Sigma\text{E-2}]$$

$$\frac{\Gamma \vdash pr_1 \equiv pr_2 : (\Sigma\ ((x\ A))\ D)}{\Gamma \vdash (\mathsf{cdr}\ pr_1) \equiv (\mathsf{cdr}\ pr_2) : D[(\mathsf{car}\ pr_1)/x]}\ [\Sigma\textsc{Same-}\mathsf{cdr}]$$

$$\frac{\Gamma \vdash a_1 \equiv a_2 : A \qquad \Gamma, x : A \vdash d_1 \equiv d_2 : D}{\Gamma \vdash (\mathsf{cdr}\ (\mathsf{cons}\ a_1\ d_1)) \equiv d_2 : D[a_2/x]}\ [\Sigma\textsc{Same-}\iota 2]$$

$$\frac{\Gamma \vdash pr_1 \equiv pr_2 : (\Sigma\ ((x\ A))\ D)}{\Gamma \vdash pr_1 \equiv (\mathsf{cons}\ (\mathsf{car}\ pr_2)\ (\mathsf{cdr}\ pr_2)) : (\Sigma\ ((x\ A))\ D)}\ [\Sigma\textsc{Same-}\eta]$$

Functions

$$\frac{\Gamma \vdash Arg\ \ \textbf{type} \rightsquigarrow Arg^o \qquad \Gamma, x : Arg^o \vdash R\ \ \textbf{type} \rightsquigarrow R^o}{\Gamma \vdash (\Pi\ ((x\ Arg))\ R)\ \ \textbf{type} \rightsquigarrow (\Pi\ ((x\ Arg^o))\ R^o)}\ [\textsc{FunF-1}]$$

$$\frac{\begin{array}{l}\Gamma \vdash Arg\ \ \textbf{type} \rightsquigarrow Arg^o \\ \Gamma, x : Arg^o \vdash (\Pi\ ((x_1\ Arg_1)\ \ldots\ (x_n\ Arg_n))\ R)\ \ \textbf{type} \rightsquigarrow X\end{array}}{\Gamma \vdash (\Pi\ ((x\ Arg)\ (x_1\ Arg_1)\ \ldots\ (x_n\ Arg_n))\ R)\ \ \textbf{type} \rightsquigarrow (\Pi\ ((x\ Arg^o))\ X)}\ [\textsc{FunF-2}]$$

$$\frac{\Gamma \vdash Arg\ \ \textbf{type} \rightsquigarrow Arg^o \qquad \Gamma \vdash \textbf{fresh} \rightsquigarrow x \qquad \Gamma, x : Arg^o \vdash R\ \ \textbf{type} \rightsquigarrow R^o}{\Gamma \vdash (\rightarrow\ Arg\ R)\ \ \textbf{type} \rightsquigarrow (\Pi\ ((x\ Arg^o))\ R^o)}\ [\textsc{FunF}{\rightarrow}1]$$

$$\frac{\begin{array}{l}\Gamma \vdash Arg\ \ \textbf{type} \rightsquigarrow Arg^o \\ \Gamma \vdash \textbf{fresh} \rightsquigarrow x \\ \Gamma, x : Arg^o \vdash (\rightarrow\ Arg_1\ \ldots\ Arg_n\ R)\ \ \textbf{type} \rightsquigarrow X\end{array}}{\Gamma \vdash (\rightarrow\ Arg\ Arg_1\ \ldots\ Arg_n\ R)\ \ \textbf{type} \rightsquigarrow (\Pi\ ((x\ Arg^o))\ X)}\ [\textsc{FunF}{\rightarrow}2]$$

Remember to read the rules aloud! To read FunF→2, say:

> To check that an →-expression with more than one argument type is a type, first check that the first argument type Arg is a type. Call its Core Pie expression Arg^o. Then, check that a new →-expression with the remaining argument types $Arg_1 \ldots Arg_n$ is a type, and call the resulting Core Pie expression X. Find a fresh variable name x that is not associated with any type in Γ, and then the result of elaboration is $(\Pi\ ((x\ Arg^o))\ X)$.

$$\frac{\Gamma \vdash Arg_1 \equiv Arg_2\ \ \textbf{type} \qquad \Gamma, x : Arg_1 \vdash R_1 \equiv R_2\ \ \textbf{type}}{\Gamma \vdash (\Pi\ ((x\ Arg_1))\ R_1) \equiv (\Pi\ ((x\ Arg_2))\ R_2)\ \ \textbf{type}}\ [\textsc{FunSame-}\Pi]$$

$$\frac{\Gamma, x : Arg \vdash r \in R \rightsquigarrow r^o}{\Gamma \vdash (\lambda\ (x)\ r) \in (\Pi\ ((x\ Arg))\ R) \rightsquigarrow (\lambda\ (x)\ r^o)}\ [\textsc{FunI-1}]$$

$$\frac{\Gamma, x : Arg \vdash (\lambda\ (y\ z \ldots)\ r) \in R \rightsquigarrow r^o}{\Gamma \vdash (\lambda\ (x\ y\ z \ldots)\ r) \in (\Pi\ ((x\ Arg))\ R) \rightsquigarrow (\lambda\ (x)\ r^o)}\ [\textsc{FunI-2}]$$

$$\frac{\Gamma, x : Arg \vdash r_1 \equiv r_2 : R}{\Gamma \vdash (\lambda\ (x)\ r_1) \equiv (\lambda\ (x)\ r_2) : (\Pi\ ((x\ Arg))\ R)}\ [\textsc{FunSame-}\lambda]$$

$$\frac{\Gamma \vdash f\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\Pi\ ((x\ Arg))\ R)\ f^o) \qquad \Gamma \vdash arg \in Arg \rightsquigarrow arg^o}{\Gamma \vdash (f\ arg)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ R[arg^o/x]\ (f^o\ arg^o))}\ [\textsc{FunE-1}]$$

$$\frac{\begin{array}{l}\Gamma \vdash (f\ arg\ \ldots\ arg_{n-1})\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\Pi\ ((x\ Arg))\ R)\ f^o) \\ \Gamma \vdash arg_n \in Arg \rightsquigarrow arg_n^o\end{array}}{\Gamma \vdash (f\ arg\ \ldots\ arg_{n-1}\ arg_n)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ R[arg_n^o/x]\ (f^o\ arg_n^o))}\ [\textsc{FunE-2}]$$

$$\frac{\Gamma \vdash f_1 \equiv f_2 : (\Pi\ ((x\ Arg))\ R) \qquad \Gamma \vdash arg_1 \equiv arg_2 : Arg}{\Gamma \vdash (f_1\ arg_1) \equiv (f_2\ arg_2) : R[arg_1/x]}\ [\textsc{FunSame-}\mathsf{apply}]$$

$$\frac{\Gamma, x : Arg \vdash r_1 \equiv r_2 : R \qquad \Gamma \vdash arg_1 \equiv arg_2 : Arg}{\Gamma \vdash ((\lambda\ (x)\ r_1)\ arg_1) \equiv r_2[arg_2/x] : R[arg_2/x]}\ [\textsc{FunSame-}\beta]$$

In FunSame-η, the premise $x \notin \mathrm{dom}(\Gamma)$ states that x is not bound by Γ. The reason that $\Gamma \vdash \textbf{fresh} \rightsquigarrow x$ is not used in this rule is that the rules for sameness are a specification that the conversion checking algorithm must fulfill rather than the algorithm itself. It would be inappropriate to use an algorithmic check in a non-algorithmic specification.

$$\frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash f_1 \equiv f_2 : (\Pi\ ((x\ Arg))\ R)}{\Gamma \vdash f_1 \equiv (\lambda\ (x)\ (f_2\ x)) : (\Pi\ ((x\ Arg))\ R)}\ [\textsc{FunSame-}\eta]$$

Natural Numbers

$$\frac{}{\Gamma \vdash \mathsf{Nat}\ \ \textbf{type} \rightsquigarrow \mathsf{Nat}}\ [\textsc{NatF}] \qquad \frac{}{\Gamma \vdash \mathsf{Nat} \equiv \mathsf{Nat}\ \ \textbf{type}}\ [\textsc{NatSame-Nat}]$$

$$\frac{}{\Gamma \vdash \mathsf{zero}\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathsf{Nat}\ \mathsf{zero})}\ [\textsc{NatI-1}]$$

$$\frac{}{\Gamma \vdash \mathsf{zero} \equiv \mathsf{zero} : \mathsf{Nat}}\ [\textsc{NatSame-zero}]$$

$$\frac{\Gamma \vdash n \in \mathsf{Nat} \rightsquigarrow n^o}{\Gamma \vdash (\mathsf{add1}\ n)\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ \mathsf{Nat}\ (\mathsf{add1}\ n^o))}\ [\textsc{NatI-2}]$$

In these rules, $\lceil n \rceil$ stands for a literal Scheme natural number.

$$\frac{}{\Gamma \vdash \lceil 0 \rceil\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ \mathsf{Nat}\ \mathsf{zero})}\ [\textsc{NatI-3}]$$

$$\frac{\Gamma \vdash \lceil k \rceil \in \mathsf{Nat} \rightsquigarrow n}{\Gamma \vdash \lceil k+1 \rceil\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ \mathsf{Nat}\ (\mathsf{add1}\ n))}\ [\textsc{NatI-4}]$$

$$\frac{\Gamma \vdash n_1 \equiv n_2 : \mathsf{Nat}}{\Gamma \vdash (\mathsf{add1}\ n_1) \equiv (\mathsf{add1}\ n_2) : \mathsf{Nat}}\ [\textsc{NatSame-add1}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t \in \mathsf{Nat} \rightsquigarrow t^o \\ \Gamma \vdash b\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ B\ b^o) \\ \Gamma \vdash s \in (\Pi\ ((x\ \mathsf{Nat}))\ B) \rightsquigarrow s^o\end{array}}{\Gamma \vdash (\mathsf{which\text{-}Nat}\ t\ b\ s)\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ B\ (\mathsf{which\text{-}Nat}\ t^o\ (\mathtt{the}\ B\ b^o)\ s^o))}\ [\textsc{NatE-1}]$$

In the next rule, a sameness judgment is written on multiple lines. The following two ways of writing the judgment have the same meaning:

$$\Gamma \vdash \begin{array}{c} c_1 \\ \equiv \\ c_2 \end{array} : c_3 \qquad \text{and} \qquad \Gamma \vdash c_1 \equiv c_2 : c_3$$

In addition to allowing wider expressions, this way of writing the judgment can also make it easier to visually compare the two expressions that are the same.

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : \mathsf{Nat} \\ \Gamma \vdash B_1 \equiv B_2\ \ \mathbf{type} \\ \Gamma \vdash b_1 \equiv b_2 : B_1 \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((x\ \mathsf{Nat}))\ B_1)\end{array}}{\Gamma \vdash \begin{array}{c}(\mathsf{which\text{-}Nat}\ t_1\ (\mathtt{the}\ B_1\ b_1)\ s_1) \\ \equiv \\ (\mathsf{which\text{-}Nat}\ t_2\ (\mathtt{the}\ B_2\ b_2)\ s_2)\end{array} : B_1}\ [\textsc{NatSame-w-N}]$$

$$\frac{\Gamma \vdash b_1 \equiv b_2 : B \qquad \Gamma \vdash s \equiv s : (\Pi\ ((x\ \mathsf{Nat}))\ B)}{\Gamma \vdash (\mathsf{which\text{-}Nat}\ \mathsf{zero}\ (\mathtt{the}\ B\ b_1)\ s) \equiv b_2 : B}\ [\textsc{NatSame-w-N}\iota 1]$$

$$\frac{\begin{array}{l}\Gamma \vdash n_1 \equiv n_2 : \mathsf{Nat} \\ \Gamma \vdash b \equiv b : B \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((x\ \mathsf{Nat}))\ B)\end{array}}{\Gamma \vdash (\textsf{which-Nat}\ (\mathsf{add1}\ n_1)\ (\mathtt{the}\ B\ b)\ s_1) \equiv (s_2\ n_2) : B}\ [\textsc{NatSame-w-N}\iota 2]$$

$$\frac{\begin{array}{l}\Gamma \vdash t \in \mathsf{Nat} \rightsquigarrow t^o \\ \Gamma \vdash b\ \ \textbf{synth} \rightsquigarrow (\mathtt{the}\ B\ b^o) \\ \Gamma \vdash s \in (\Pi\ ((x\ B))\ B) \rightsquigarrow s^o\end{array}}{\Gamma \vdash (\textsf{iter-Nat}\ t\ b\ s)\ \ \textbf{synth} \rightsquigarrow (\mathtt{the}\ B\ (\textsf{iter-Nat}\ t^o\ (\mathtt{the}\ B\ b^o)\ s^o))}\ [\textsc{NatE-2}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : \mathsf{Nat} \\ \Gamma \vdash B_1 \equiv B_2\ \ \textbf{type} \\ \Gamma \vdash b_1 \equiv b_2 : B_1 \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((x\ B_1))\ B_1)\end{array}}{\Gamma \vdash \begin{array}{c}(\textsf{iter-Nat}\ t_1\ (\mathtt{the}\ B_1\ b_1)\ s_1) \\ \equiv \\ (\textsf{iter-Nat}\ t_2\ (\mathtt{the}\ B_2\ b_2)\ s_2)\end{array} : B_1}\ [\textsc{NatSame-iter-Nat}]$$

$$\frac{\begin{array}{l}\Gamma \vdash b_1 \equiv b_2 : B \\ \Gamma \vdash s \equiv s : (\Pi\ ((x\ B))\ B)\end{array}}{\Gamma \vdash (\textsf{iter-Nat}\ \mathsf{zero}\ (\mathtt{the}\ B\ b_1)\ s) \equiv b_2 : B}\ [\textsc{NatSame-it-N}\iota 1]$$

$$\frac{\begin{array}{l}\Gamma \vdash n_1 \equiv n_2 : \mathsf{Nat} \\ \Gamma \vdash B_1 \equiv B_2\ \ \textbf{type} \\ \Gamma \vdash b_1 \equiv b_2 : B_1 \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((x\ B_1))\ B_1)\end{array}}{\Gamma \vdash \begin{array}{c}(\textsf{iter-Nat}\ (\mathsf{add1}\ n_1)\ (\mathtt{the}\ B_1\ b_1)\ s_1) \\ \equiv \\ (s_2\ (\textsf{iter-Nat}\ n_2\ (\mathtt{the}\ B_2\ b_2)\ s_2))\end{array} : B_1}\ [\textsc{NatSame-it-N}\iota 2]$$

Try comparing the rules for **which-Nat** and **iter-Nat** with each other, and keep them in mind when reading the rules for **rec-Nat** aloud.

$$\frac{\begin{array}{l}\Gamma \vdash t \in \mathsf{Nat} \rightsquigarrow t^o \\ \Gamma \vdash b\ \ \textbf{synth} \rightsquigarrow (\mathtt{the}\ B\ b^o) \\ \Gamma \vdash s \in (\Pi\ ((n\ \mathsf{Nat}))\ (\Pi\ ((x\ B))\ B)) \rightsquigarrow s^o\end{array}}{\Gamma \vdash (\textsf{rec-Nat}\ t\ b\ s)\ \ \textbf{synth} \rightsquigarrow (\mathtt{the}\ B\ (\textsf{rec-Nat}\ t^o\ (\mathtt{the}\ B\ b^o)\ s^o))}\ [\textsc{NatE-3}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : \mathsf{Nat} \\ \Gamma \vdash B_1 \equiv B_2 \ \ \mathbf{type} \\ \Gamma \vdash b_1 \equiv b_2 : B_1 \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((n\ \mathsf{Nat}))\ (\Pi\ ((x\ B_1))\ B_1))\end{array}}{\begin{array}{c}\Gamma \vdash \\ \end{array}\begin{array}{c}(\mathsf{rec\text{-}Nat}\ t_1\ (\mathtt{the}\ B_1\ b_1)\ s_1) \\ \equiv \\ (\mathsf{rec\text{-}Nat}\ t_2\ (\mathtt{the}\ B_2\ b_2)\ s_2)\end{array} : B_1} \ [\textsc{NatSame-rec-Nat}]$$

$$\frac{\Gamma \vdash b_1 \equiv b_2 : B \qquad \Gamma \vdash s \equiv s : (\Pi\ ((n\ \mathsf{Nat}))\ (\Pi\ ((x\ B))\ B))}{\Gamma \vdash (\mathsf{rec\text{-}Nat}\ \mathsf{zero}\ (\mathtt{the}\ B\ b_1)\ s) \equiv b_2 : B}\ [\textsc{NatSame-r-N}\iota 1]$$

$$\frac{\begin{array}{l}\Gamma \vdash n_1 \equiv n_2 : \mathsf{Nat} \\ \Gamma \vdash B_1 \equiv B_2 \ \ \mathbf{type} \\ \Gamma \vdash b_1 \equiv b_2 : B_1 \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((n\ \mathsf{Nat}))\ (\Pi\ ((x\ B_1))\ B_1))\end{array}}{\begin{array}{c}\Gamma \vdash \\ \end{array}\begin{array}{c}(\mathsf{rec\text{-}Nat}\ (\mathsf{add1}\ n_1)\ (\mathtt{the}\ B_1\ b_1)\ s_1) \\ \equiv \\ ((s_2\ n_2)\ (\mathsf{rec\text{-}Nat}\ n_2\ (\mathtt{the}\ B_2\ b_2)\ s_2))\end{array} : B_1}\ [\textsc{NatSame-r-N}\iota 2]$$

$$\frac{\begin{array}{l}\Gamma \vdash t \in \mathsf{Nat} \rightsquigarrow t^o \\ \Gamma \vdash m \in (\Pi\ ((x\ \mathsf{Nat}))\ \mathcal{U}) \rightsquigarrow m^o \\ \Gamma \vdash b \in (m^o\ \mathsf{zero}) \rightsquigarrow b^o \\ \Gamma \vdash s \in (\Pi\ ((k\ \mathsf{Nat}))\ (\Pi\ ((\mathit{almost}\ (m^o\ k)))\ (m^o\ (\mathsf{add1}\ k)))) \rightsquigarrow s^o\end{array}}{\Gamma \vdash (\mathsf{ind\text{-}Nat}\ t\ m\ b\ s)\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (m^o\ t^o)\ (\mathsf{ind\text{-}Nat}\ t^o\ m^o\ b^o\ s^o))}\ [\textsc{NatE-4}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : \mathsf{Nat} \\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((x\ \mathsf{Nat}))\ \mathcal{U}) \\ \Gamma \vdash b_1 \equiv b_2 : (m_1\ \mathsf{zero}) \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((k\ \mathsf{Nat}))\ (\Pi\ ((\mathit{almost}\ (m_1\ k)))\ (m_1\ (\mathsf{add1}\ k))))\end{array}}{\begin{array}{c}\Gamma \vdash \\ \end{array}\begin{array}{c}(\mathsf{ind\text{-}Nat}\ t_1\ m_1\ b_1\ s_1) \\ \equiv \\ (\mathsf{ind\text{-}Nat}\ t_2\ m_2\ b_2\ s_2)\end{array} : (m_1\ t_1)}\ [\textsc{NatSame-ind-Nat}]$$

$$\frac{\begin{array}{l}\Gamma \vdash m \equiv m : (\Pi\ ((x\ \mathsf{Nat}))\ \mathcal{U}) \\ \Gamma \vdash b_1 \equiv b_2 : (m\ \mathsf{zero}) \\ \Gamma \vdash s \equiv s : (\Pi\ ((k\ \mathsf{Nat})) \\ \qquad\qquad (\Pi\ ((\mathit{almost}\ (m\ k))) \\ \qquad\qquad\quad (m\ (\mathsf{add1}\ k))))\end{array}}{\Gamma \vdash (\textsf{ind-Nat}\ \mathsf{zero}\ m\ b_1\ s) \equiv b_2 : (m\ \mathsf{zero})}\ [\textsc{NatSame-ind-Nat1}]$$

$$\frac{\begin{array}{l}\Gamma \vdash n_1 \equiv n_2 : \mathsf{Nat} \\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((x\ \mathsf{Nat}))\ \mathcal{U}) \\ \Gamma \vdash b_1 \equiv b_2 : (m_1\ \mathsf{zero}) \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((k\ \mathsf{Nat})) \\ \qquad\qquad (\Pi\ ((\mathit{almost}\ (m_1\ k))) \\ \qquad\qquad\quad (m_1\ (\mathsf{add1}\ k))))\end{array}}{\begin{array}{c}(\textsf{ind-Nat}\ (\mathsf{add1}\ n_1)\ m_1\ b_1\ s_1) \\ \Gamma \vdash \qquad\qquad \equiv \qquad\qquad : (m_1\ (\mathsf{add1}\ n_1)) \\ ((s_2\ n_2)\ (\textsf{ind-Nat}\ n_2\ m_2\ b_2\ s_2))\end{array}}\ [\textsc{NatSame-ind-Nat2}]$$

Lists

$$\frac{\Gamma \vdash E\ \ \mathbf{type} \rightsquigarrow E^o}{\Gamma \vdash (\mathsf{List}\ E)\ \ \mathbf{type} \rightsquigarrow (\mathsf{List}\ E^o)}\ [\textsc{ListF}]$$

$$\frac{\Gamma \vdash E_1 \equiv E_2\ \ \mathbf{type}}{\Gamma \vdash (\mathsf{List}\ E_1) \equiv (\mathsf{List}\ E_2)\ \ \mathbf{type}}\ [\textsc{ListSame-List}]$$

$$\frac{}{\Gamma \vdash \mathsf{nil} \in (\mathsf{List}\ E) \rightsquigarrow \mathsf{nil}}\ [\textsc{ListI-1}]$$

$$\frac{}{\Gamma \vdash \mathsf{nil} \equiv \mathsf{nil} : (\mathsf{List}\ E)}\ [\textsc{ListSame-nil}]$$

$$\frac{\Gamma \vdash e\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ E\ e^o) \qquad \Gamma \vdash es \in (\mathsf{List}\ E) \rightsquigarrow es^o}{\Gamma \vdash (::\ e\ es)\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (\mathsf{List}\ E)\ (::\ e^o\ es^o))}\ [\textsc{ListI-2}]$$

$$\frac{\Gamma \vdash e_1 \equiv e_2 : E \qquad \Gamma \vdash es_1 \equiv es_2 : (\mathsf{List}\ E)}{\Gamma \vdash (::\ e_1\ es_1) \equiv (::\ e_2\ es_2) : (\mathsf{List}\ E)}\ [\textsc{ListSame-::}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t \ \ \mathbf{synth} \rightsquigarrow (\texttt{the}\ (\mathsf{List}\ E)\ t^o) \\ \Gamma \vdash b \ \ \mathbf{synth} \rightsquigarrow (\texttt{the}\ B\ b^o) \\ \Gamma \vdash s \in (\Pi\ ((x\ E))\ (\Pi\ ((xs\ (\mathsf{List}\ E)))\ (\Pi\ ((\mathit{almost}\ B))\ B))) \rightsquigarrow s^o\end{array}}{\Gamma \vdash (\textsf{rec-List}\ t\ b\ s) \ \ \mathbf{synth} \rightsquigarrow (\texttt{the}\ B\ (\textsf{rec-List}\ t^o\ (\texttt{the}\ B\ b^o)\ s^o))}\ [\textsc{ListE-1}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : (\mathsf{List}\ E) \\ \Gamma \vdash B_1 \equiv B_2 \ \ \mathbf{type} \\ \Gamma \vdash b_1 \equiv b_2 : B_1 \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((x\ E)) \\ \qquad\qquad (\Pi\ ((xs\ (\mathsf{List}\ E))) \\ \qquad\qquad\quad (\Pi\ ((\mathit{almost}\ B_1)) \\ \qquad\qquad\qquad B_1)))\end{array}}{\Gamma \vdash (\textsf{rec-List}\ t_1\ (\texttt{the}\ B_1\ b_1)\ s_1) \equiv (\textsf{rec-List}\ t_2\ (\texttt{the}\ B_2\ b_2)\ s_2) : B_1}\ [\textsc{ListSame-rec-List}]$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathsf{nil} \equiv \mathsf{nil} : (\mathsf{List}\ E) \\ \Gamma \vdash b_1 \equiv b_2 : B \\ \Gamma \vdash s \equiv s : (\Pi\ ((x\ E)) \\ \qquad\qquad (\Pi\ ((xs\ (\mathsf{List}\ E))) \\ \qquad\qquad\quad (\Pi\ ((\mathit{almost}\ B)) \\ \qquad\qquad\qquad B)))\end{array}}{\Gamma \vdash (\textsf{rec-List}\ \mathsf{nil}\ (\texttt{the}\ B\ b_1)\ s_1) \equiv b_2 : B}\ [\textsc{ListSame-r-L}\iota 1]$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \equiv e_2 : E \\ \Gamma \vdash es_1 \equiv es_2 : (\mathsf{List}\ E) \\ \Gamma \vdash B_1 \equiv B_2 \ \ \mathbf{type} \\ \Gamma \vdash b_1 \equiv b_2 : B_1 \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((x\ E)) \\ \qquad\qquad (\Pi\ ((xs\ (\mathsf{List}\ E))) \\ \qquad\qquad\quad (\Pi\ ((\mathit{almost}\ B_1)) \\ \qquad\qquad\qquad B_1)))\end{array}}{\begin{array}{c}(\textsf{rec-List}\ (::\ e_1\ es_1)\ (\texttt{the}\ B_1\ b_1)\ s_1) \\ \Gamma \vdash \qquad\qquad\qquad \equiv \qquad\qquad\qquad : B_1 \\ (((s_2\ e_2)\ es_2)\ (\textsf{rec-List}\ es_2\ (\texttt{the}\ B_2\ b_2)\ s_2))\end{array}}\ [\textsc{ListSame-r-L}\iota 2]$$

$$\frac{\begin{array}{l}\Gamma \vdash t \ \mathbf{synth} \rightsquigarrow (\texttt{the}\ (\mathsf{List}\ E)\ t^o) \\ \Gamma \vdash m \in (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E)))\ \mathcal{U}) \rightsquigarrow m^o \\ \Gamma \vdash b \in (m^o\ \mathsf{nil}) \rightsquigarrow b^o \\ \Gamma \vdash s \in (\Pi\ ((\mathit{x}\ E)) \\ \qquad (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E))) \\ \qquad\quad (\Pi\ ((\mathit{almost}\ (m^o\ \mathit{xs}))) \\ \qquad\qquad (m^o\ (::\ \mathit{x}\ \mathit{xs}))))) \quad \rightsquigarrow s^o\end{array}}{\Gamma \vdash (\textsf{ind-List}\ t\ m\ b\ s)\ \mathbf{synth} \rightsquigarrow (\texttt{the}\ (m^o\ t^o)\ (\textsf{ind-List}\ t^o\ m^o\ b^o\ s^o))}\ [\textsc{ListE-2}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : (\mathsf{List}\ E) \\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E)))\ \mathcal{U}) \\ \Gamma \vdash b_1 \equiv b_2 : (m_1\ \mathsf{nil}) \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((\mathit{x}\ E)) \\ \qquad (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E))) \\ \qquad\quad (\Pi\ ((\mathit{almost}\ (m_1\ \mathit{xs}))) \\ \qquad\qquad (m_1\ (::\ \mathit{x}\ \mathit{xs})))))\end{array}}{\Gamma \vdash (\textsf{ind-List}\ t_1\ m_1\ b_1\ s_1) \equiv (\textsf{ind-List}\ t_2\ m_2\ b_2\ s_2) : (m_1\ t_1)}\ [\textsc{ListSame-ind-List}]$$

$$\frac{\begin{array}{l}\Gamma \vdash m \equiv m : (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E)))\ \mathcal{U}) \\ \Gamma \vdash b_1 \equiv b_2 : (m\ \mathsf{nil}) \\ \Gamma \vdash s \equiv s : (\Pi\ ((\mathit{x}\ E)) \\ \qquad (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E))) \\ \qquad\quad (\Pi\ ((\mathit{almost}\ (m\ \mathit{xs}))) \\ \qquad\qquad (m\ (::\ \mathit{x}\ \mathit{xs})))))\end{array}}{\Gamma \vdash (\textsf{ind-List}\ \mathsf{nil}\ m\ b_1\ s) \equiv b_2 : (m\ \mathsf{nil})}\ [\textsc{ListSame-i-L}\iota 1]$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \equiv e_2 : E \\ \Gamma \vdash es_1 \equiv es_2 : (\mathsf{List}\ E) \\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E)))\ \mathcal{U}) \\ \Gamma \vdash b_1 \equiv b_2 : (m_1\ \mathsf{nil}) \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((\mathit{x}\ E)) \\ \qquad (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E))) \\ \qquad\quad (\Pi\ ((\mathit{almost}\ (m_1\ \mathit{xs}))) \\ \qquad\qquad (m_1\ (::\ \mathit{x}\ \mathit{xs})))))\end{array}}{\begin{array}{l}\Gamma \vdash \quad (\textsf{ind-List}\ (::\ e_1\ es_1)\ m_1\ b_1\ s_1) \\ \qquad\qquad\qquad \equiv \qquad\qquad\qquad : (m_1\ (::\ e_1\ es_1)) \\ \qquad (((s_2\ e_2)\ es_2)\ (\textsf{ind-List}\ es_2\ m_2\ b_2\ s_2))\end{array}}\ [\textsc{ListSame-i-L}\iota 2]$$

Vectors

$$\frac{\Gamma \vdash E \ \ \mathbf{type} \rightsquigarrow E^o \qquad \Gamma \vdash \ell \in \mathsf{Nat} \rightsquigarrow \ell^o}{\Gamma \vdash (\mathsf{Vec}\ E\ \ell) \ \ \mathbf{type} \rightsquigarrow (\mathsf{Vec}\ E^o\ \ell^o)}\ [\textsc{VecF}]$$

$$\frac{\Gamma \vdash E_1 \equiv E_2 \ \ \mathbf{type} \qquad \Gamma \vdash \ell_1 \equiv \ell_2 : \mathsf{Nat}}{\Gamma \vdash (\mathsf{Vec}\ E_1\ \ell_1) \equiv (\mathsf{Vec}\ E_2\ \ell_2) \ \ \mathbf{type}}\ [\textsc{VecSame-Vec}]$$

$$\frac{}{\Gamma \vdash \mathsf{vecnil} \in (\mathsf{Vec}\ E\ \mathsf{zero}) \rightsquigarrow \mathsf{vecnil}}\ [\textsc{VecI-1}]$$

$$\frac{}{\Gamma \vdash \mathsf{vecnil} \equiv \mathsf{vecnil} : (\mathsf{Vec}\ E\ \mathsf{zero})}\ [\textsc{VecSame-vecnil}]$$

$$\frac{\Gamma \vdash e \in E \rightsquigarrow e^o \qquad \Gamma \vdash es \in (\mathsf{Vec}\ E\ \ell) \rightsquigarrow es^o}{\Gamma \vdash (\mathsf{vec{::}}\ e\ es) \in (\mathsf{Vec}\ E\ (\mathsf{add1}\ \ell)) \rightsquigarrow (\mathsf{vec{::}}\ e^o\ es^o)}\ [\textsc{VecI-2}]$$

$$\frac{\Gamma \vdash e_1 \equiv e_2 : E \qquad \Gamma \vdash es_1 \equiv es_2 : (\mathsf{Vec}\ E\ \ell)}{\Gamma \vdash (\mathsf{vec{::}}\ e_1\ es_1) \equiv (\mathsf{vec{::}}\ e_2\ es_2) : (\mathsf{Vec}\ E\ (\mathsf{add1}\ \ell))}\ [\textsc{VecSame-vec::}]$$

$$\frac{\Gamma \vdash t \ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (\mathsf{Vec}\ E\ (\mathsf{add1}\ \ell))\ t^o)}{\Gamma \vdash (\mathsf{head}\ t) \ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ E\ (\mathsf{head}\ t^o))}\ [\textsc{VecE-1}]$$

$$\frac{\Gamma \vdash es_1 \equiv es_2 : (\mathsf{Vec}\ E\ (\mathsf{add1}\ \ell))}{\Gamma \vdash (\mathsf{head}\ es_1) \equiv (\mathsf{head}\ es_2) : E}\ [\textsc{VecSame-}\mathsf{head}]$$

$$\frac{\Gamma \vdash e_1 \equiv e_2 : E \qquad \Gamma \vdash es \equiv es : (\mathsf{Vec}\ E\ \ell)}{\Gamma \vdash (\mathsf{head}\ (\mathsf{vec{::}}\ e_1\ es)) \equiv e_2 : E}\ [\textsc{VecSame-}\mathsf{h\iota}]$$

$$\frac{\Gamma \vdash t \ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (\mathsf{Vec}\ E\ (\mathsf{add1}\ \ell))\ t^o)}{\Gamma \vdash (\mathsf{tail}\ t) \ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (\mathsf{Vec}\ E\ \ell)\ (\mathsf{tail}\ t^o))}\ [\textsc{VecE-2}]$$

$$\frac{\Gamma \vdash es_1 \equiv es_2 : (\mathsf{Vec}\ E\ (\mathsf{add1}\ \ell))}{\Gamma \vdash (\mathsf{tail}\ es_1) \equiv (\mathsf{tail}\ es_2) : (\mathsf{Vec}\ E\ \ell)}\ [\textsc{VecSame-}\mathsf{tail}]$$

$$\frac{\Gamma \vdash e \equiv e : E \qquad \Gamma \vdash es_1 \equiv es_2 : (\mathsf{Vec}\ E\ \ell)}{\Gamma \vdash (\mathsf{tail}\ (\mathsf{vec{::}}\ e\ es_1)) \equiv es_2 : (\mathsf{Vec}\ E\ \ell)}\ [\textsc{VecSame-}\mathsf{t\iota}]$$

In VecE-3 below, there is a premise stating that ℓ^o and n are the same Nat, rather than using the same metavariable for both lengths. This is because both Nats are

output, bound on the right of a $\rightsquigarrow$. The Core Pie Nats are independently produced by elaboration, so they must be checked for sameness in another premise. This pattern occurs in [EqI], as well.

$$\frac{\begin{array}{l}\Gamma \vdash \ell \in \mathsf{Nat} \rightsquigarrow \ell^o \\ \Gamma \vdash t\ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (\mathsf{Vec}\ E\ n)\ t^o) \\ \Gamma \vdash \ell^o \equiv n : \mathsf{Nat} \\ \Gamma \vdash m \in (\Pi\ ((k\ \mathsf{Nat}))\ (\Pi\ ((es\ (\mathsf{Vec}\ E\ k)))\ \mathcal{U})) \rightsquigarrow m^o \\ \Gamma \vdash b \in ((m^o\ \mathsf{zero})\ \mathsf{vecnil}) \rightsquigarrow b^o \\ \Gamma \vdash s \in (\Pi\ ((k\ \mathsf{Nat})) \\ \qquad (\Pi\ ((e\ E)) \\ \qquad\quad (\Pi\ ((es\ (\mathsf{Vec}\ E\ k))) \\ \qquad\qquad (\Pi\ ((almost\ ((m^o\ k)\ es))) \\ \qquad\qquad\quad ((m^o\ (\mathsf{add1}\ k))\ (\mathsf{vec{::}}\ e\ es)))))) \rightsquigarrow s^o\end{array}}{\Gamma \vdash (\mathsf{ind\text{-}Vec}\ \ell\ t\ m\ b\ s)\ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ ((m^o\ \ell^o)\ t^o)\ (\mathsf{ind\text{-}Vec}\ \ell^o\ t^o\ m^o\ b^o\ s^o))}\ [\textsc{VecE-3}]$$

$$\frac{\begin{array}{l}\Gamma \vdash \ell_1 \equiv \ell_2 : \mathsf{Nat} \\ \Gamma \vdash t_1 \equiv t_2 : (\mathsf{Vec}\ E\ \ell_1) \\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((k\ \mathsf{Nat}))\ (\Pi\ ((x\ (\mathsf{Vec}\ E\ k)))\ \mathcal{U})) \\ \Gamma \vdash b_1 \equiv b_2 : ((m_1\ \mathsf{zero})\ \mathsf{vecnil}) \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((k\ \mathsf{Nat})) \\ \qquad (\Pi\ ((e\ E)) \\ \qquad\quad (\Pi\ ((es\ (\mathsf{Vec}\ E\ k))) \\ \qquad\qquad (\Pi\ ((almost\ ((m_1\ k)\ es))) \\ \qquad\qquad\quad ((m_1\ (\mathsf{add1}\ k))\ (\mathsf{vec{::}}\ e\ es))))))\end{array}}{\Gamma \vdash \begin{array}{c}(\mathsf{ind\text{-}Vec}\ \ell_1\ t_1\ m_1\ b_1\ s_1) \\ \equiv \\ (\mathsf{ind\text{-}Vec}\ \ell_2\ t_2\ m_2\ b_2\ s_2)\end{array} : ((m_1\ \ell_1)\ t_1)}\ [\textsc{VecSame-ind-Vec}]$$

$$\frac{\begin{array}{l}\Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((k\ \mathsf{Nat}))\ (\Pi\ ((x\ (\mathsf{Vec}\ E\ k)))\ \mathcal{U})) \\ \Gamma \vdash b_1 \equiv b_2 : ((m_1\ \mathsf{zero})\ \mathsf{vecnil}) \\ \Gamma \vdash s \equiv s : (\Pi\ ((k\ \mathsf{Nat})) \\ \qquad (\Pi\ ((e\ E)) \\ \qquad\quad (\Pi\ ((es\ (\mathsf{Vec}\ E\ k))) \\ \qquad\qquad (\Pi\ ((almost\ ((m_1\ k)\ es))) \\ \qquad\qquad\quad ((m_1\ (\mathsf{add1}\ k))\ (\mathsf{vec{::}}\ e\ es))))))\end{array}}{\Gamma \vdash (\mathsf{ind\text{-}Vec}\ \mathsf{zero}\ \mathsf{vecnil}\ m_1\ b_1\ s) \equiv b_2 : ((m_2\ \mathsf{zero})\ \mathsf{vecnil})}\ [\textsc{VecSame-i-Vι1}]$$

$$\frac{\begin{array}{l}\Gamma \vdash \ell_1 \equiv \ell_2 : \mathsf{Nat} \\ \Gamma \vdash e_1 \equiv e_2 : E \\ \Gamma \vdash es_1 \equiv es_2 : (\mathsf{Vec}\ E\ \ell_1) \\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((k\ \mathsf{Nat}))\ (\Pi\ ((x\ (\mathsf{Vec}\ E\ k)))\ \mathcal{U})) \\ \Gamma \vdash b_1 \equiv b_2 : ((m_1\ \mathsf{zero})\ \mathsf{vecnil}) \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((k\ \mathsf{Nat})) \\ \qquad\qquad (\Pi\ ((e\ E)) \\ \qquad\qquad\quad (\Pi\ ((es\ (\mathsf{Vec}\ E\ k))) \\ \qquad\qquad\qquad (\Pi\ ((\mathit{almost}\ ((m_1\ k)\ es))) \\ \qquad\qquad\qquad\quad ((m_1\ (\mathsf{add1}\ k))\ (\mathsf{vec{::}}\ e\ es))))))\end{array}}{\Gamma \vdash \begin{array}{c}(\mathbf{ind\text{-}Vec}\ (\mathsf{add1}\ \ell_1)\ (\mathsf{vec{::}}\ e_1\ es_1)\ m_1\ b_1\ s_1) \\ \equiv \\ ((((s_2\ \ell_2)\ e_2)\ es_2) \\ (\mathbf{ind\text{-}Vec}\ \ell_2\ es_2\ m_2\ b_2\ s_2))\end{array} : \begin{array}{c}((m_2\ (\mathsf{add1}\ \ell_1)) \\ (\mathsf{vec{::}}\ e_1\ es_1))\end{array}}\ [\textsc{VecSame-i-Vι2}]$$

Equality

$$\frac{\Gamma \vdash X\ \ \mathbf{type} \rightsquigarrow X^o \qquad \Gamma \vdash \mathit{from} \in X^o \rightsquigarrow \mathit{from}^o \qquad \Gamma \vdash \mathit{to} \in X^o \rightsquigarrow \mathit{to}^o}{\Gamma \vdash (\mathbf{=}\ X\ \mathit{from}\ \mathit{to})\ \ \mathbf{type} \rightsquigarrow (\mathbf{=}\ X^o\ \mathit{from}^o\ \mathit{to}^o)}\ [\textsc{EqF}]$$

$$\frac{\Gamma \vdash X_1 \equiv X_2\ \ \mathbf{type} \qquad \Gamma \vdash \mathit{from}_1 \equiv \mathit{from}_2 : X_1 \qquad \Gamma \vdash \mathit{to}_1 \equiv \mathit{to}_2 : X_1}{\Gamma \vdash (\mathbf{=}\ X_1\ \mathit{from}_1\ \mathit{to}_1) \equiv (\mathbf{=}\ X_2\ \mathit{from}_2\ \mathit{to}_2)\ \ \mathbf{type}}\ [\textsc{EqSame-=}]$$

$$\frac{\Gamma \vdash \mathit{mid} \in X \rightsquigarrow \mathit{mid}^o \qquad \Gamma \vdash \mathit{from} \equiv \mathit{mid}^o : X \qquad \Gamma \vdash \mathit{mid}^o \equiv \mathit{to} : X}{\Gamma \vdash (\mathsf{same}\ \mathit{mid}) \in (\mathbf{=}\ X\ \mathit{from}\ \mathit{to}) \rightsquigarrow (\mathsf{same}\ \mathit{mid}^o)}\ [\textsc{EqI}]$$

$$\frac{\Gamma \vdash \mathit{from} \equiv \mathit{to} : X}{\Gamma \vdash (\mathsf{same}\ \mathit{from}) \equiv (\mathsf{same}\ \mathit{to}) : (\mathbf{=}\ X\ \mathit{from}\ \mathit{from})}\ [\textsc{EqSame-same}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (\mathbf{=}\ X\ \mathit{from}\ \mathit{to})\ t^o) \\ \Gamma \vdash m \in (\Pi\ ((x\ X))\ \mathcal{U}) \rightsquigarrow m^o \\ \Gamma \vdash b \in (m^o\ \mathit{from}) \rightsquigarrow b^o\end{array}}{\Gamma \vdash (\mathbf{replace}\ t\ m\ b)\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (m^o\ \mathit{to})\ (\mathbf{replace}\ t^o\ m^o\ b^o))}\ [\textsc{EqE-1}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : (\mathbf{=}\ X\ \mathit{from}\ \mathit{to}) \\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((x\ X))\ \mathcal{U}) \\ \Gamma \vdash b_1 \equiv b_2 : (m_1\ \mathit{from})\end{array}}{\Gamma \vdash (\mathsf{replace}\ t_1\ m_1\ b_1) \equiv (\mathsf{replace}\ t_2\ m_2\ b_2) : (m_1\ \mathit{to})}\ [\textsc{EqSame-replace}]$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{expr} \equiv \mathit{expr} : X \\ \Gamma \vdash m \equiv m : (\Pi\ ((x\ X))\ \mathcal{U}) \\ \Gamma \vdash b_1 \equiv b_2 : (m\ \mathit{expr})\end{array}}{\Gamma \vdash (\mathsf{replace}\ (\mathsf{same}\ \mathit{expr})\ m\ b_1) \equiv b_2 : (m\ \mathit{expr})}\ [\textsc{EqSame-rl}]$$

The Core Pie version of **cong** takes three arguments, rather than two, as can be seen in the grammar on page 393. The first argument in the Core Pie version is the type of the expressions being equated, and it is needed in order for a sameness checking algorithm to take types into account.

$$\frac{\begin{array}{l}\Gamma \vdash t\ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (\mathbf{=}\ X_1\ \mathit{from}\ \mathit{to})\ t^o) \\ \Gamma \vdash f\ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (\Pi\ ((x\ X_2))\ Y)\ f^o) \\ \Gamma \vdash X_1 \equiv X_2\ \mathbf{type}\end{array}}{\Gamma \vdash (\mathsf{cong}\ t\ f)\ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (\mathbf{=}\ Y\ (f^o\ \mathit{from})\ (f^o\ \mathit{to}))\ (\mathsf{cong}\ X_1\ t^o\ f^o))}\ [\textsc{EqE-2}]$$

$$\frac{\begin{array}{l}\Gamma \vdash X_1 \equiv X_2\ \mathbf{type} \\ \Gamma \vdash f_1 \equiv f_2 : (\Pi\ ((x\ X_1))\ Y) \\ \Gamma \vdash t_1 \equiv t_2 : (\mathbf{=}\ X_1\ \mathit{from}\ \mathit{to})\end{array}}{\Gamma \vdash (\mathsf{cong}\ X_1\ t_1\ f_1) \equiv (\mathsf{cong}\ X_2\ t_2\ f_2) : (\mathbf{=}\ Y\ (f_1\ \mathit{from})\ (f_1\ \mathit{to}))}\ [\textsc{EqSame-cong}]$$

$$\frac{\Gamma \vdash \mathit{expr}_1 \equiv \mathit{expr}_2 : X \qquad \Gamma \vdash f_1 \equiv f_2 : (\Pi\ ((x\ X))\ Y)}{\Gamma \vdash (\mathsf{cong}\ X\ (\mathsf{same}\ \mathit{expr}_1)\ f_1) \equiv (\mathsf{same}\ (f_2\ \mathit{expr}_2)) : (\mathbf{=}\ X\ (f_1\ \mathit{expr}_1)\ (f_1\ \mathit{expr}_1))}\ [\textsc{EqSame-cl}]$$

$$\frac{\Gamma \vdash t\ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (\mathbf{=}\ X\ \mathit{from}\ \mathit{to})\ t^o)}{\Gamma \vdash (\mathsf{symm}\ t)\ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (\mathbf{=}\ X\ \mathit{to}\ \mathit{from})\ (\mathsf{symm}\ t^o))}\ [\textsc{EqE-3}]$$

$$\frac{\Gamma \vdash t_1 \equiv t_2 : (\mathbf{=}\ X\ \mathit{from}\ \mathit{to})}{\Gamma \vdash (\mathsf{symm}\ t_1) \equiv (\mathsf{symm}\ t_2) : (\mathbf{=}\ X\ \mathit{to}\ \mathit{from})}\ [\textsc{EqSame-symm}]$$

$$\frac{\Gamma \vdash expr_1 \equiv expr_2 : X}{\Gamma \vdash (\textsf{symm}\ (\textsf{same}\ expr_1)) \equiv (\textsf{same}\ expr_2) : (\textsf{=}\ X\ expr_1\ expr_1)}\ [\textsc{EqSame-sι}]$$

Pie contains two eliminators for equality that are not discussed in the preceding chapters: **trans** and **ind-=**. **trans** allows evidence of equality to be "glued together:" if the TO of one equality is the same as the FROM of another, **trans** allows the construction of an equality connecting the FROM of the first equality to the TO of the second.

$$\frac{\begin{array}{l}\Gamma \vdash t_1\ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textsf{=}\ X\ from\ mid_1)\ t_1^o) \\ \Gamma \vdash t_2\ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textsf{=}\ Y\ mid_2\ to)\ t_2^o) \\ \Gamma \vdash X \equiv Y\ \textbf{type} \\ \Gamma \vdash mid_1 \equiv mid_2 : X\end{array}}{\Gamma \vdash (\textsf{trans}\ t_1\ t_2)\ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textsf{=}\ X\ from\ to)\ (\textsf{trans}\ t_1^o\ t_2^o))}\ [\textsc{EqE-4}]$$

$$\frac{\Gamma \vdash t_1 \equiv t_2 : (\textsf{=}\ X\ from\ mid) \qquad \Gamma \vdash t_3 \equiv t_4 : (\textsf{=}\ X\ mid\ to)}{\Gamma \vdash (\textsf{trans}\ t_1\ t_3) \equiv (\textsf{trans}\ t_2\ t_4) : (\textsf{=}\ X\ from\ to)}\ [\textsc{EqSame-trans}]$$

$$\frac{\Gamma \vdash expr_1 \equiv expr_2 : X \qquad \Gamma \vdash expr_2 \equiv expr_3 : X}{\Gamma \vdash (\textsf{trans}\ (\textsf{same}\ expr_1)\ (\textsf{same}\ expr_2)) \equiv (\textsf{same}\ expr_3) : (\textsf{=}\ X\ expr_3\ expr_3)}\ [\textsc{EqSame-tι}]$$

The most powerful eliminator for equality is called **ind-=**: it expresses *induction on evidence of equality*. **ind-=** is sometimes called J[5] or *path induction*. Pie's **ind-=** treats the FROM as a parameter, rather than an index;[6] this version of induction on evidence of equality is sometimes called *based path induction*.

$$\frac{\begin{array}{l}\Gamma \vdash t\ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textsf{=}\ X\ from\ to)\ t^o) \\ \Gamma \vdash m \in (\Pi\ ((x\ X))\ (\Pi\ ((\imath\ (\textsf{=}\ X\ from\ x)))\ \mathcal{U})) \rightsquigarrow m^o \\ \Gamma \vdash b \in ((m^o\ from)\ (\textsf{same}\ from)) \rightsquigarrow b^o\end{array}}{\Gamma \vdash (\textsf{ind-=}\ t\ m\ b)\ \textbf{synth} \rightsquigarrow (\texttt{the}\ ((m^o\ to)\ t^o)\ (\textsf{ind-=}\ t^o\ m^o\ b^o))}\ [\textsc{EqE-5}]$$

[5]Thanks again, Per Martin-Löf.

[6]Thanks, Christine Paulin-Mohring (1962–)

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : (\mathbf{=}\ X\ \mathit{from}\ \mathit{to}) \\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((x\ X))\ (\Pi\ ((t\ (\mathbf{=}\ X\ \mathit{from}\ x)))\ \mathcal{U})) \\ \Gamma \vdash b_1 \equiv b_2 : ((m_1\ \mathit{from})\ (\mathsf{same}\ \mathit{from}))\end{array}}{\Gamma \vdash (\mathsf{ind\text{-}=}\ t_1\ m_1\ b_1) \equiv (\mathsf{ind\text{-}=}\ t_2\ m_2\ b_2) : ((m_1\ \mathit{to})\ t_1)}\ [\textsc{EqSame-ind-=}]$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{expr} \equiv \mathit{expr} : X \\ \Gamma \vdash m \equiv m : (\Pi\ ((x\ X))\ (\Pi\ ((t\ (\mathbf{=}\ X\ \mathit{expr}\ x)))\ \mathcal{U})) \\ \Gamma \vdash b_1 \equiv b_2 : ((m\ \mathit{expr})\ (\mathsf{same}\ \mathit{expr}))\end{array}}{\Gamma \vdash (\mathsf{ind\text{-}=}\ (\mathsf{same}\ \mathit{expr})\ m\ b_1) \equiv b_2 : ((m\ \mathit{expr})\ (\mathsf{same}\ \mathit{expr}))}\ [\textsc{EqSame-i-=ι}]$$

Either

$$\frac{\Gamma \vdash P\ \ \mathbf{type} \rightsquigarrow P^o \qquad \Gamma \vdash S\ \ \mathbf{type} \rightsquigarrow S^o}{\Gamma \vdash (\mathsf{Either}\ P\ S)\ \ \mathbf{type} \rightsquigarrow (\mathsf{Either}\ P^o\ S^o)}\ [\textsc{EitherF}]$$

$$\frac{\Gamma \vdash P_1 \equiv P_2\ \ \mathbf{type} \qquad \Gamma \vdash S_1 \equiv S_2\ \ \mathbf{type}}{\Gamma \vdash (\mathsf{Either}\ P_1\ S_1) \equiv (\mathsf{Either}\ P_2\ S_2)\ \ \mathbf{type}}\ [\textsc{EitherSame-Either}]$$

$$\frac{\Gamma \vdash \mathit{lt} \in P \rightsquigarrow \mathit{lt}^o}{\Gamma \vdash (\mathsf{left}\ \mathit{lt}) \in (\mathsf{Either}\ P\ S) \rightsquigarrow (\mathsf{left}\ \mathit{lt}^o)}\ [\textsc{EitherI-1}]$$

$$\frac{\Gamma \vdash \mathit{lt}_1 \equiv \mathit{lt}_2 : P}{\Gamma \vdash (\mathsf{left}\ \mathit{lt}_1) \equiv (\mathsf{left}\ \mathit{lt}_2) : (\mathsf{Either}\ P\ S)}\ [\textsc{EitherSame-left}]$$

$$\frac{\Gamma \vdash \mathit{rt} \in S \rightsquigarrow \mathit{rt}^o}{\Gamma \vdash (\mathsf{right}\ \mathit{rt}) \in (\mathsf{Either}\ P\ S) \rightsquigarrow (\mathsf{right}\ \mathit{rt}^o)}\ [\textsc{EitherI-2}]$$

$$\frac{\Gamma \vdash \mathit{rt}_1 \equiv \mathit{rt}_2 : S}{\Gamma \vdash (\mathsf{right}\ \mathit{rt}_1) \equiv (\mathsf{right}\ \mathit{rt}_2) : (\mathsf{Either}\ P\ S)}\ [\textsc{EitherSame-right}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (\mathsf{Either}\ P\ S)\ t^o) \\ \Gamma \vdash m \in (\Pi\ ((x\ (\mathsf{Either}\ P\ S)))\ \mathcal{U}) \rightsquigarrow m^o \\ \Gamma \vdash b_l \in (\Pi\ ((x\ P))\ (m^o\ (\mathsf{left}\ x))) \rightsquigarrow b_l^o \\ \Gamma \vdash b_r \in (\Pi\ ((x\ S))\ (m^o\ (\mathsf{right}\ x))) \rightsquigarrow b_r^o\end{array}}{\Gamma \vdash (\mathsf{ind\text{-}Either}\ t\ m\ b_l\ b_r)\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ (m^o\ t^o)\ (\mathsf{ind\text{-}Either}\ t^o\ m^o\ b_l^o\ b_r^o))}\ [\textsc{EitherE}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : (\mathsf{Either}\ P\ S) \\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((x\ (\mathsf{Either}\ P\ S)))\ \mathcal{U}) \\ \Gamma \vdash b_{l1} \equiv b_{l2} : (\Pi\ ((x\ P))\ (m_1\ (\mathsf{left}\ x))) \\ \Gamma \vdash b_{r1} \equiv b_{r2} : (\Pi\ ((x\ S))\ (m_1\ (\mathsf{right}\ x)))\end{array}}{\Gamma \vdash \begin{array}{c}(\textsf{ind-Either}\ t_1\ m_1\ b_{l1}\ b_{r1}) \\ \equiv \\ (\textsf{ind-Either}\ t_2\ m_2\ b_{l2}\ b_{r2})\end{array} : (m_1\ t_1)}\ [\textsc{EitherSame-ind-Either}]$$

$$\frac{\begin{array}{l}\Gamma \vdash lt_1 \equiv lt_2 : P \\ \Gamma \vdash m \equiv m : (\Pi\ ((x\ (\mathsf{Either}\ P\ S)))\ \mathcal{U}) \\ \Gamma \vdash b_{l1} \equiv b_{l2} : (\Pi\ ((x\ P))\ (m\ (\mathsf{left}\ x))) \\ \Gamma \vdash b_r \equiv b_r : (\Pi\ ((x\ S))\ (m\ (\mathsf{right}\ x)))\end{array}}{\Gamma \vdash \begin{array}{c}(\textsf{ind-Either}\ (\mathsf{left}\ lt_1)\ m\ b_{l1}\ b_r) \\ \equiv \\ (b_{l2}\ lt_2)\end{array} : (m\ (\mathsf{left}\ lt_1))}\ [\textsc{EitherSame-i-E}\iota 1]$$

$$\frac{\begin{array}{l}\Gamma \vdash rt_1 \equiv rt_2 : S \\ \Gamma \vdash m \equiv m : (\Pi\ ((x\ (\mathsf{Either}\ P\ S)))\ \mathcal{U}) \\ \Gamma \vdash b_l \equiv b_l : (\Pi\ ((x\ P))\ (m\ (\mathsf{left}\ x))) \\ \Gamma \vdash b_{r1} \equiv b_{r2} : (\Pi\ ((x\ S))\ (m\ (\mathsf{right}\ x)))\end{array}}{\Gamma \vdash \begin{array}{c}(\textsf{ind-Either}\ (\mathsf{right}\ rt_1)\ m\ b_l\ b_{r1}) \\ \equiv \\ (b_{r2}\ rt_2)\end{array} : (m\ (\mathsf{right}\ rt_1))}\ [\textsc{EitherSame-i-E}\iota 2]$$

Unit

$$\frac{}{\Gamma \vdash \mathsf{Trivial}\ \ \mathbf{type} \rightsquigarrow \mathsf{Trivial}}\ [\textsc{TrivF}]$$

$$\frac{}{\Gamma \vdash \mathsf{Trivial} \equiv \mathsf{Trivial}\ \ \mathbf{type}}\ [\textsc{TrivSame-Trivial}]$$

$$\frac{}{\Gamma \vdash \mathsf{sole}\ \ \mathbf{synth} \rightsquigarrow (\texttt{the}\ \mathsf{Trivial}\ \mathsf{sole})}\ [\textsc{TrivI}]$$

It is not necessary to have a rule stating that sole is the same Trivial as sole because every Trivial is the same as every other by the η-rule.

$$\frac{\Gamma \vdash c \equiv c : \mathsf{Trivial}}{\Gamma \vdash c \equiv \mathsf{sole} : \mathsf{Trivial}}\ [\textsc{TrivSame-}\eta]$$

Absurdities

$$\frac{}{\Gamma \vdash \mathsf{Absurd}\ \ \mathbf{type} \rightsquigarrow \mathsf{Absurd}}\ [\textsc{AbsF}]$$

$$\frac{}{\Gamma \vdash \mathsf{Absurd} \equiv \mathsf{Absurd}\ \ \mathbf{type}}\ [\textsc{AbsSame-Absurd}]$$

$$\frac{\Gamma \vdash t \in \mathsf{Absurd} \rightsquigarrow t^o \qquad \Gamma \vdash m\ \ \mathbf{type} \rightsquigarrow m^o}{\Gamma \vdash (\mathsf{ind\text{-}Absurd}\ t\ m)\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ m^o\ (\mathsf{ind\text{-}Absurd}\ t^o\ m^o))}\ [\textsc{AbsE}]$$

$$\frac{\Gamma \vdash t_1 \equiv t_2 : \mathsf{Absurd} \qquad \Gamma \vdash m_1 \equiv m_2 : \mathcal{U}}{\Gamma \vdash (\mathsf{ind\text{-}Absurd}\ t_1\ m_1) \equiv (\mathsf{ind\text{-}Absurd}\ t_2\ m_2) : m_1}\ [\textsc{AbsSame-}\mathsf{ind\text{-}Absurd}]$$

$$\frac{\Gamma \vdash c_1 \equiv c_1 : \mathsf{Absurd} \qquad \Gamma \vdash c_2 \equiv c_2 : \mathsf{Absurd}}{\Gamma \vdash c_1 \equiv c_2 : \mathsf{Absurd}}\ [\textsc{AbsSame-}\eta]$$

Universe

The rules for $\mathcal{U}$ work differently from other types. There is a formation rule and a number of introduction rules, but there is not an elimination rule that expresses induction the way that there is for types such as Nat and families such as Vec.

Instead of an elimination rule, a $\mathcal{U}$ is used by placing it in a context where a type is expected, because one way to check that an expression is a type is by checking that it is a $\mathcal{U}$. Similarly, to check that two expressions are the same type, one can check that they are the same $\mathcal{U}$.

$$\frac{\Gamma \vdash e \in \mathcal{U} \rightsquigarrow c_t}{\Gamma \vdash e \ \textbf{type} \rightsquigarrow c_t}\ [\textsc{El}] \qquad \frac{\Gamma \vdash X \equiv Y : \mathcal{U}}{\Gamma \vdash X \equiv Y \ \textbf{type}}\ [\textsc{El-Same}]$$

The formation rule for $\mathcal{U}$ is akin to types that take no arguments: Atom, Nat, Trivial, and Absurd.

$$\frac{}{\Gamma \vdash \mathcal{U} \ \textbf{type} \rightsquigarrow \mathcal{U}}\ [\text{UF}] \qquad \frac{}{\Gamma \vdash \mathcal{U} \equiv \mathcal{U} \ \textbf{type}}\ [\textsc{Usame-}\mathcal{U}]$$

$$\frac{}{\Gamma \vdash \mathsf{Atom} \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ \mathsf{Atom})}\ [\text{UI-1}] \qquad \frac{}{\Gamma \vdash \mathsf{Atom} \equiv \mathsf{Atom} : \mathcal{U}}\ [\textsc{Usame-Atom}]$$

$$\frac{\Gamma \vdash A \in \mathcal{U} \rightsquigarrow A^o \qquad \Gamma, x : A^o \vdash D \in \mathcal{U} \rightsquigarrow D^o}{\Gamma \vdash (\Sigma\ ((x\ A))\ D) \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ (\Sigma\ ((x\ A^o))\ D^o))}\ [\text{UI-2}]$$

$$\frac{\Gamma \vdash A \in \mathcal{U} \rightsquigarrow A^o \qquad \Gamma, x : A^o \vdash (\Sigma\ ((x_1\ A_1)\ \ldots\ (x_n\ A_n))\ D) \in \mathcal{U} \rightsquigarrow Z}{\Gamma \vdash (\Sigma\ ((x\ A)\ (x_1\ A_1)\ \ldots\ (x_n\ A_n))\ D) \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ (\Sigma\ ((x\ A^o))\ Z))}\ [\text{UI-3}]$$

$$\frac{\Gamma \vdash A \in \mathcal{U} \rightsquigarrow A^o \qquad \Gamma \vdash \textbf{fresh} \rightsquigarrow x \qquad \Gamma, x : A^o \vdash D \in \mathcal{U} \rightsquigarrow D^o}{\Gamma \vdash (\mathsf{Pair}\ A\ D) \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ (\Sigma\ ((x\ A^o))\ D^o))}\ [\text{UI-4}]$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 : \mathcal{U} \qquad \Gamma, x : A_1 \vdash D_1 \equiv D_2 : \mathcal{U}}{\Gamma \vdash (\Sigma\ ((x\ A_1))\ D_1) \equiv (\Sigma\ ((x\ A_2))\ D_2) : \mathcal{U}}\ [\textsc{Usame-}\Sigma]$$

$$\frac{\Gamma \vdash X \in \mathcal{U} \rightsquigarrow X^o \qquad \Gamma, x : X^o \vdash R \in \mathcal{U} \rightsquigarrow R^o}{\Gamma \vdash (\Pi\ ((x\ X))\ R) \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ (\Pi\ ((x\ X^o))\ R^o))}\ [\text{UI-5}]$$

$$\frac{\Gamma \vdash X \in \mathcal{U} \rightsquigarrow X^o \qquad \Gamma, x : X^o \vdash (\Pi\ ((x_1\ X_1)\ \ldots\ (x_n\ X_n))\ R) \in \mathcal{U} \rightsquigarrow R^o}{\Gamma \vdash (\Pi\ ((x\ X)\ (x_1\ X_1)\ \ldots\ (x_n\ X_n))\ R)\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ \mathcal{U}\ (\Pi\ ((x\ X^o))\ R^o))}\ [\text{UI-6}]$$

$$\frac{\Gamma \vdash X \in \mathcal{U} \rightsquigarrow X^o \qquad \Gamma \vdash \mathbf{fresh} \rightsquigarrow x \qquad \Gamma, x : X^o \vdash R \in \mathcal{U} \rightsquigarrow R^o}{\Gamma \vdash (\rightarrow\ X\ R)\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ \mathcal{U}\ (\Pi\ ((x\ X^o))\ R^o))}\ [\text{UI-7}]$$

$$\frac{\begin{array}{l}\Gamma \vdash X \in \mathcal{U} \rightsquigarrow X^o \\ \Gamma \vdash \mathbf{fresh} \rightsquigarrow x \\ \Gamma, x : X^o \vdash (\rightarrow\ X_1\ \ldots\ X_n\ R) \in \mathcal{U} \rightsquigarrow R^o\end{array}}{\Gamma \vdash (\rightarrow\ X\ X_1\ \ldots\ X_n\ R)\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ \mathcal{U}\ (\Pi\ ((x\ X^o))\ R^o))}\ [\text{UI-8}]$$

$$\frac{\Gamma \vdash X_1 \equiv X_2 : \mathcal{U} \qquad \Gamma, x : X_1 \vdash Y_1 \equiv Y_2 : \mathcal{U}}{\Gamma \vdash (\Pi\ ((x\ X_1))\ Y_1) \equiv (\Pi\ ((x\ X_2))\ Y_2) : \mathcal{U}}\ [\text{USAME-}\Pi]$$

$$\frac{}{\Gamma \vdash \mathsf{Nat}\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ \mathcal{U}\ \mathsf{Nat})}\ [\text{UI-9}] \qquad \frac{}{\Gamma \vdash \mathsf{Nat} \equiv \mathsf{Nat} : \mathcal{U}}\ [\text{USAME-Nat}]$$

$$\frac{\Gamma \vdash E \in \mathcal{U} \rightsquigarrow E^o}{\Gamma \vdash (\mathsf{List}\ E)\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ \mathcal{U}\ (\mathsf{List}\ E^o))}\ [\text{UI-10}]$$

$$\frac{\Gamma \vdash E_1 \equiv E_2 : \mathcal{U}}{\Gamma \vdash (\mathsf{List}\ E_1) \equiv (\mathsf{List}\ E_2) : \mathcal{U}}\ [\text{USAME-List}]$$

$$\frac{\Gamma \vdash E \in \mathcal{U} \rightsquigarrow E^o \qquad \Gamma \vdash \ell \in \mathsf{Nat} \rightsquigarrow \ell^o}{\Gamma \vdash (\mathsf{Vec}\ E\ \ell)\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ \mathcal{U}\ (\mathsf{Vec}\ E^o\ \ell^o))}\ [\text{UI-11}]$$

$$\frac{\Gamma \vdash E_1 \equiv E_2 : \mathcal{U} \qquad \Gamma \vdash \ell_1 \equiv \ell_2 : \mathsf{Nat}}{\Gamma \vdash (\mathsf{Vec}\ E_1\ \ell_1) \equiv (\mathsf{Vec}\ E_2\ \ell_2) : \mathcal{U}}\ [\text{USAME-Vec}]$$

$$\frac{\Gamma \vdash X \in \mathcal{U} \rightsquigarrow X^o \qquad \Gamma \vdash \mathit{from} \in X^o \rightsquigarrow \mathit{from}^o \qquad \Gamma \vdash \mathit{to} \in X^o \rightsquigarrow \mathit{to}^o}{\Gamma \vdash (=\ X\ \mathit{from}\ \mathit{to})\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ \mathcal{U}\ (=\ X^o\ \mathit{from}^o\ \mathit{to}^o))}\ [\text{UI-12}]$$

$$\frac{\Gamma \vdash X_1 \equiv X_2 : \mathcal{U} \qquad \Gamma \vdash \mathit{from}_1 \equiv \mathit{from}_2 : X_1 \qquad \Gamma \vdash \mathit{to}_1 \equiv \mathit{to}_2 : X_1}{\Gamma \vdash (= X_1\ \mathit{from}_1\ \mathit{to}_1) \equiv (= X_2\ \mathit{from}_2\ \mathit{to}_2) : \mathcal{U}}\ [\textsc{Same}\text{-}=]$$

$$\frac{\Gamma \vdash P_1 \equiv P_2 : \mathcal{U} \qquad \Gamma \vdash S_1 \equiv S_2 : \mathcal{U}}{\Gamma \vdash (\mathsf{Either}\ P_1\ S_1) \equiv (\mathsf{Either}\ P_2\ S_2) : \mathcal{U}}\ [\textsc{Same}\text{-}\mathsf{Either}]$$

$$\frac{\Gamma \vdash P \in \mathcal{U} \rightsquigarrow P^o \qquad \Gamma \vdash S \in \mathcal{U} \rightsquigarrow S^o}{\Gamma \vdash (\mathsf{Either}\ P\ S)\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ \mathcal{U}\ (\mathsf{Either}\ P^o\ S^o))}\ [\text{UI-13}]$$

$$\frac{}{\Gamma \vdash \mathsf{Trivial}\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ \mathcal{U}\ \mathsf{Trivial})}\ [\text{UI-14}]$$

$$\frac{}{\Gamma \vdash \mathsf{Trivial} \equiv \mathsf{Trivial} : \mathcal{U}}\ [\textsc{Same}\text{-}\mathsf{Trivial}]$$

$$\frac{}{\Gamma \vdash \mathsf{Absurd}\ \ \mathbf{synth} \rightsquigarrow (\mathtt{the}\ \mathcal{U}\ \mathsf{Absurd})}\ [\text{UI-15}]$$

$$\frac{}{\Gamma \vdash \mathsf{Absurd} \equiv \mathsf{Absurd} : \mathcal{U}}\ [\textsc{Same}\text{-}\mathsf{Absurd}]$$

The Grammar of Pie

e	::=	(the e e)	Type annotation
	|	x	Variable reference
	|	Atom	Atom type
	|	'$\lceil sym \rceil$	Atom literal
	|	(Pair e e)	Non-dependent pair type
	|	(Σ ((x e)$^+$) e)	Dependent pair type
	|	(cons e e)	Pair constructor
	|	(**car** e)	First projection
	|	(**cdr** e)	Second projection
	|	(→ e e^+)	Non-dependent function type
	|	(Π ((x e)$^+$) e)	Dependent function type
	|	(λ (x^+) e)	Functions
	|	(e e^+)	Application
	|	Nat	Natural number type
	|	zero	Zero
	|	(add1 e)	Successor
	|	$\lceil n \rceil$	Natural number literal
	|	(**which-Nat** e e e)	Case operator on natural numbers
	|	(**iter-Nat** e e e)	Simply-typed iteration on natural numbers
	|	(**rec-Nat** e e e)	Simply-typed recursion on natural numbers
	|	(**ind-Nat** e e e e)	Induction on natural numbers
	|	(List e)	List type
	|	nil	Empty list
	|	(:: e e)	List expansion
	|	(**rec-List** e e e)	Simply-typed list recursion
	|	(**ind-List** e e e e)	Induction on lists
	|	(Vec e e)	Length-indexed vector type
	|	vecnil	Empty vector
	|	(vec:: e e)	Vector extension
	|	(**head** e)	Head of a vector
	|	(**tail** e)	Tail of a vector
	|	(**ind-Vec** e e e e e)	Induction on vectors
	|	(**=** e e e)	Equality type
	|	(same e)	Reflexivity of equality
	|	(**symm** e)	Symmetry of equality
	|	(**cong** e e)	Equality is a congruence
	|	(**replace** e e e)	Transportation along equality
	|	(**trans** e e)	Transitivity of equality
	|	(**ind-=** e e e)	Induction on equality
	|	(Either e e)	Sum type
	|	(left e)	First injection
	|	(right e)	Second injection
	|	(**ind-Either** e e e e)	Eliminator for sums
	|	Trivial	Unit type
	|	sole	Unit constructor
	|	Absurd	Empty type
	|	(**ind-Absurd** e e)	Eliminator for empty type (a.k.a. *ex falso quodlibet*)
	|	$\mathcal{U}$	Universe

The Grammar of Core Pie

The main differences between Pie and Core Pie are that Core Pie does not have some of the features found in Pie: digits for natural numbers, the type constructors → and Pair, and functions that can be applied to more than one argument. Additionally, non-dependent eliminators require extra type information in Core Pie, because they do not have a motive. In this grammar, gray highlights indicate modifications from Pie.

c	::=	(the c c)	Type annotation
	|	x	Variable reference
	|	Atom	Atom type
	|	'⌈sym⌉	Atom literal
	|	(Σ ((x c)) c)	Dependent pair type
	|	(cons c c)	Pair constructor
	|	(**car** c)	First projection
	|	(**cdr** c)	Second projection
	|	(Π ((x c)) c)	Dependent function type
	|	(λ (x) c)	Functions
	|	(c c)	Application
	|	Nat	Natural number type
	|	zero	Zero
	|	(add1 c)	Successor
	|	(**which-Nat** c (the c c) c)	Case operator on natural numbers
	|	(**iter-Nat** c (the c c) c)	Simply-typed iteration on natural numbers
	|	(**rec-Nat** c (the c c) c)	Simply-typed recursion on natural numbers
	|	(**ind-Nat** c c c c)	Induction on natural numbers
	|	(List c)	List type
	|	nil	Empty list
	|	(:: c c)	List expansion
	|	(**rec-List** c (the c c) c)	Simply-typed list recursion
	|	(**ind-List** c c c c)	Induction on lists
	|	(Vec c c)	Length-indexed vector type
	|	vecnil	Empty vector
	|	(vec:: c c)	Vector extension
	|	(**head** c)	Head of a vector
	|	(**tail** c)	Tail of a vector
	|	(**ind-Vec** c c c c c)	Induction on vectors
	|	(= c c c)	Equality type
	|	(same c)	Reflexivity of equality
	|	(**symm** c)	Symmetry of equality
	|	(**cong** c c c)	Equality is a congruence
	|	(**replace** c c c)	Transportation along equality
	|	(**trans** c c)	Transitivity of equality
	|	(**ind-=** c c c)	Induction on equality
	|	(Either c c)	Sum type
	|	(left c)	First injection
	|	(right c)	Second injection
	|	(**ind-Either** c c c c)	Eliminator for sums
	|	Trivial	Unit type
	|	sole	Unit constructor
	|	Absurd	Empty type
	|	(**ind-Absurd** c c)	Eliminator for empty type (a.k.a. *ex falso quodlibet*)
	|	$\mathcal{U}$	Universe

Afterword

Well, that was fun, and now I'm full, and so are you. I was a Little Lisper once; now I'm a Typer, too. Types provide the means to put the meaning on machines, to program computation as an act of explanation. How is doing doing good? (How is lunch made out of food?) When are lurking loop instructions struck from structural inductions? A strong introduction, a sweet reduction, rich and warm: the chefs are on joyous normal form.

Pairs and atoms made my cradle. Pattern matching filled my youth. Now my kitchen's rich with Σ, poaching pairs of things with truth. Cookery: it's not just flattery. Who's the pudding kidding without the proof? It takes Π to make a promise and a promise to make trust, to make windows you can see through and build gates that do not rust. Here is Pie for Simple Simon: the faker at the fair went bust. I would serve Pie to my father, but he's dust.

Atoms offer difference in the act of giving name. $=$ transubstantiates two types which mean the same. Absurd is just another word for someone else to blame. Time flies like an $\rightarrow$. Pairs share out space. A $\mathcal{U}$niversal type of types unites the human race. But what on earth do we think we're doing in the first place? What's our game? We have the ways of making things, but things are evidence. Perhaps, one day, the thing we'll make is sense.

Conor McBride
Glasgow
February, 2018

Index