The Little Typer

Daniel P. Friedman and David Thrane Christiansen

Foreword by Robert Harper   Afterword by Conor McBride

Drawings by Duane Bibby

The Little Typer

The Little Typer

Daniel P. Friedman
David Thrane Christiansen

Drawings by Duane Bibby

Foreword by Robert Harper

Afterword by Conor McBride

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Computer Modern Unicode by the authors using LaTeX. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Names: Friedman, Daniel P., author. | Christiansen, David Thrane, author.
Title: The little typer / Daniel P. Friedman and David Thrane Christiansen ; drawings by Duane Bibby ; foreword by Robert Harper ; afterword by Conor McBride.
Description: Cambridge, MA : The MIT Press, [2018] | Includes index.
Identifiers: LCCN 2018017792 | ISBN 9780262536431 (pbk. : alk. paper)
Subjects: LCSH: Functional programming (Computer science) | Logic programming. | Computer logic. | Type theory.
Classification: LCC QA76.63 .F75 2018 | DDC 005.101/5113–dc23 LC record available at `https://lccn.loc.gov/2018017792`

10 9 8 7 6 5 4 3 2 1

To Mary, with all my love.

Til Lisbet, min elskede.

(**Contents**

Foreword

Dependent type theory, the subject of this book, is a wonderfully beguiling, and astonishingly effective, unification of mathematics and programming. In type theory when you prove a theorem you are writing a program to meet a specification—and you can even run it when you are done! A proof of the fundamental theorem of arithmetic amounts to a program for factoring numbers. And it works the other way as well: every program is a proof that its specification is sensible enough to be implementable. Type theory is a hacker's paradise.

And yet, for many, type theory remains an esoteric world of sacred texts, revered figures, and arcane terminology—a hermetic realm out of the novels of Umberto Eco. Be mystified no longer! My colleagues Dan Friedman and David Christiansen reveal the secrets of type theory in an engaging, organic style that is both delightful and enlightening, particularly for those for whom running code is the touchstone of rigor. You will learn about normal forms, about canonization, about families of types, about dependent elimination, and even learn the ulterior motives for induction.

When you are done, you will have reached a new level of understanding of both mathematics and programming, gaining entrance to what is surely the future of both. Enjoy the journey, the destination is magnificent!

Robert Harper
Pittsburgh
February, 2018

Preface

A program's type describes its behavior. *Dependent types* are a first-class part of a language, which makes them vastly more powerful than other kinds of types. Using just one language for types and programs allows program descriptions to be just as powerful as the programs that they describe.

If you can write programs, then you can write proofs. This may come as a surprise—for most of us, the two activities seem as different as sleeping and bicycling. It turns out, however, that tools we know from programming, such as pairs, lists, functions, and recursion, can also capture patterns of reasoning. An understanding of recursive functions over non-nested lists and non-negative numbers is all you need to understand this book. In particular, the first four chapters of *The Little Schemer* are all that's needed for learning to write programs and proofs that work together.

While mathematics is traditionally carried out in the human mind, the marriage of math and programming allows us to run our math just as we run our programs. Similarly, combining programming with math allows our programs to directly express *why* they work.

Our goal is to build an understanding of the important philosophical and mathematical ideas behind dependent types. The first five chapters provide the needed tools to understand dependent types. The remaining chapters use these tools to build a bridge between math and programming. The turning point is chapter 8, where types become statements and programs become proofs.

Our little language Pie makes it possible to experiment with these ideas, while still being small enough to be understood completely. The implementation of Pie is designed to take the mystery out of implementing dependent types. We encourage you to modify, extend, and hack on it—you can even bake your own Pie in the language of your choice. The first appendix, *The Way Forward*, explains how Pie relates to fully-featured dependently typed languages, and the second appendix, *Rules Are Made to Be Spoken*, gives a complete description of how the Pie implementation works. Pie is available from `http://thelittletyper.com`.

Acknowledgments

We thank Bob and Conor for their lyrical and inspiring foreword and afterword. They are renowned for their creative work in type theory and type practice, and for their exceptional writing. They have made major contributions to the intellectual framework behind *The Little Typer*, and their influence can be found throughout.

Suzanne Menzel, Mitch Wand, Gershom Bazerman, and Michael Vanier read multiple drafts of the book, providing detailed feedback on both the content and the exposition. Their willingness to read and re-read the text has been invaluable. Ben Boskin implemented the specification of Pie in miniKanren, alerting us to several errors and omissions in the process.

We would additionally like to thank Edwin Brady, James Chapman, Carl Factora, Jason Hemann, Andrew Kent, Weixi Ma, Wouter Swierstra, and the students in Indiana University's special topics courses on dependent types in the Spring semesters of 2017 and 2018 for their careful, considered feedback and penetrating questions. Both the clarity and the correctness of the contents were considerably improved as a result of their help.

Sam Tobin-Hochstadt, NSF grant 1540276, and the School of Informatics, Computing, and Engineering generously supported the second author during his postdoctoral fellowship at Indiana University, during which most of the writing occurred. The administrative staff, especially Lynne Mikolon and Laura Reed, as well as the chair of Computer Science Amr Sabry, continue to make Indiana University such a supportive and exciting environment.

Marie Lee and Stephanie Cohen at the MIT Press shepherded us through the process of making this book real. Similarly, a "little" book wouldn't be a "little" book without Duane Bibby's wonderful artwork.

The technical contributions of the Scheme, Racket, and LaTeX communities were tremendously valuable. In particular, we heavily used both Sam Tobin-Hochstadt's Typed Racket and Robby Findler and Matthias Felleisen's contract system while implementing Pie. Dorai Sitaram's SLaTeX system was once again invaluable in typesetting our examples, and Carl Eastlund's TeX macros and extensions to SLaTeX saved us many hours of work.

The Sweetpea Baking Company in Portland, Oregon provided a good working environment and a much-needed napkin.

Adam Foltzer introduced the authors to one another following David's internship at Galois, Inc. in 2014. We are very grateful that he brought us together.

Finally, we would like to thank Mary Friedman for her support, patience, delicious lunches, and occasional suppers during long hours of writing at the Friedman home, and Lisbet Thrane Christiansen for her support, patience, jokesmithing, help with French, and occasional consultation on graphic design.

Guidelines for the Reader

Do not rush through this book. Read carefully, including the frame notes; valuable hints are scattered throughout the text. Read every chapter. Remember to take breaks so each chapter can sink in. Read systematically. If you do not *fully* understand one chapter, you will understand the next one even less. The questions are ordered by increasing difficulty; later questions rely on comfort gained earlier in the book.

Guess! This book is based on intuition, and yours is as good as anyone's. Also, if you can, experiment with the examples while you read. The Recess that starts on page 62 contains instructions for using Pie.

From time to time, we show computation steps in a chart. Stop and work through each chart, even the long ones, and convince yourself *that* each step makes sense by understanding *why* it makes sense.

The **Laws** and **Commandments** summarize the meanings of expressions in Pie. Laws describe which expressions are meaningful, and Commandments describe which expressions are the same as others. For a Commandment to apply, it is assumed that the corresponding Laws are satisfied.

Food appears in some examples for two reasons. First, food is easier to visualize than abstract symbols. We hope the food imagery helps you to better understand the examples and concepts. Second, we want to provide a little distraction. Expanding your mind can be tiring; these snacks should help you get through the afternoon. As such, we hope that thinking about food will lead you to take some breaks and relax.

You are now ready to start. Good luck! We hope you enjoy the book.

Bon appétit!

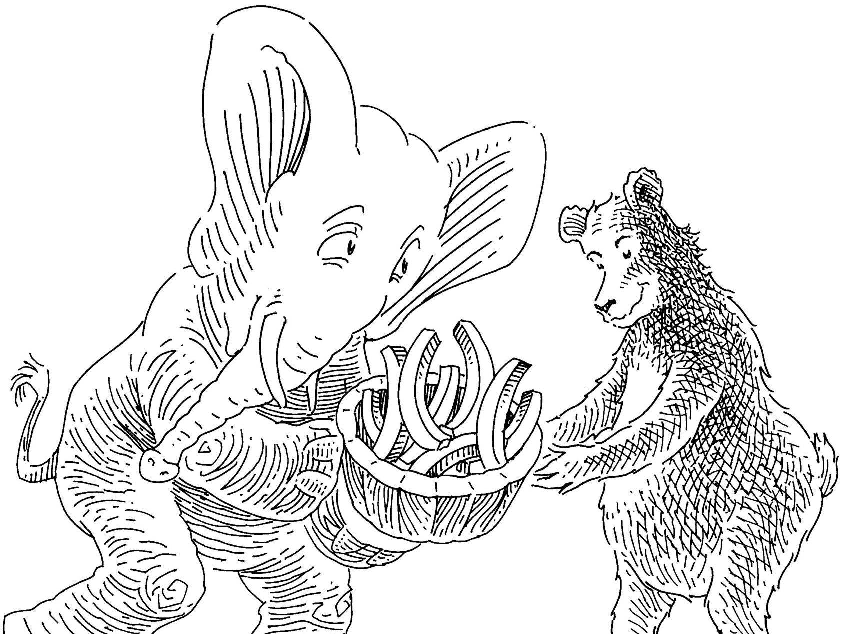
Daniel P. Friedman
Bloomington, Indiana

David Thrane Christiansen
Portland, Oregon

The Little Typer

1

The More Things Change, the More They Stay the Same



Welcome back! | 1 | It's good to be here!

Let's dust off and update some of our old toys for a new language called Pie.

Is it obvious that this is an Atom?

```
'atom
```

2 Not at all. What does Atom mean?

To be an Atom is to be an *atom*.†

†In Lisp, atoms are symbols, numbers, and many other things. Here, atoms are only symbols.

3 Then 'atom is an Atom because 'atom is an atom.

Is it obvious that this is an Atom?

```
'ratatouille
```

4 Yes, because 'ratatouille is also an atom.

But what does it precisely mean to be an atom?

Atoms are built from a tick mark directly followed by one or more letters and hyphens.†

†In Pie, only atoms use the tick mark.

5 So, is it obvious that

```
'is-it-obvious-that-this-is-an-atom
```

is an Atom?

Certainly, because atoms can contain hyphens.

What about

```
'---
```

and

```
---
```

and

```
'
```

Are they atoms?

6

```
'---
```

is an atom because hyphens can appear anywhere in an atom;

```
---
```

is not an atom because it's missing the tick mark; and

```
'
```

is not an atom because it is neither followed by a letter nor by a hyphen.

Is 'Atom an Atom?

7 Yes, even 'Atom is an Atom, because it is a tick mark followed by one or more letters or hyphens.

Is 'at0m an Atom?

8 No, because atoms can contain only letters and hyphens, as mentioned in frame 5, and the character 0 is not a letter. It is the digit zero.

Is 'cœurs-d-artichauts an Atom?

9 Yes, because œ is a letter.

Is 'ἄτομον an Atom?

10 That's Greek to me!

But Greek letters are letters, so it must also be an Atom.

The Law of Tick Marks

A tick mark directly followed by one or more letters and hyphens is an Atom.

Sentences such as

'ratatouille is an Atom

and

'cœurs-d-artichauts is an Atom

are called *judgments*.†

†Thanks, Per Martin-Löf (1942–).

11 What is the point of a judgment?

A judgment is an attitude that a person takes towards expressions. When we come to know something, we are making a judgment.

What can be judged about Atom and 'courgette?

12 'courgette is an Atom.

A *form of judgment* is an observation with blank spaces in it, such as

____ is a ____.

13 Are there other forms of judgment?

Another form of "judgment" is "judgement."

14 Very funny.

Is

'ratatouille

the same

Atom

as

'ratatouille?

15 Yes.

They are the same Atom because both have the same letters after the tick mark.

Is

'ratatouille

the same

Atom

as

'baguette?

16 No.

They have different letters after the tick mark.

The Commandment of Tick Marks

Two expressions are the same Atom if their values are tick marks followed by identical letters and hyphens.

The second form of judgment is that

____ is the same ____ as ____.

17 So

'citron is the same Atom as 'citron

is a judgment.

It is a judgment, and we have reason to believe it.

Is

'pomme is the same Atom as 'orange

a judgment?

18 It is a judgment, but we have no reason to believe it. After all, one should not compare apples and oranges.

Is it obvious that

(cons 'ratatouille 'baguette)†

is a

(Pair Atom Atom)?

†When ready, see page 62 for "typing" instructions.

19 No, it isn't.

What does it mean to be a

(Pair Atom Atom)?

To be a

(Pair Atom Atom)

is to be a pair whose **car** is an Atom, like 'ratatouille, and whose **cdr** is also an Atom, like 'baguette.

20 The names cons, **car**, and **cdr** seem familiar. What do they mean again? And what do they have to do with pairs?

A pair begins with cons[†] and ends with two more parts, called its **car** and its **cdr**.

[†]In Lisp, `cons` is used to make lists longer. Here, cons only constructs pairs.

21 Okay. That means that

(cons 'ratatouille 'baguette)

is a

(Pair Atom Atom)

because (cons 'ratatouille 'baguette) is a pair whose **car** is an Atom, and whose **cdr** is also an Atom.

Is cons a Pair, then?

Neither cons nor Pair alone is even an expression. Both require two arguments.[†]

Is

(cons 'ratatouille 'baguette)

the same

(Pair Atom Atom)

as

(cons 'ratatouille 'baguette)?

[†]In Lisp, `cons` is a procedure and has meaning on its own, but forms such as **cond** or **lambda** are meaningless if they appear alone.

22 What does it mean for two expressions to be the same

(Pair Atom Atom)?

It means that both **car**s are the same Atom and that both **cdr**s are the same Atom.

23 Then

(cons 'ratatouille 'baguette)

is indeed the same

(Pair Atom Atom)

as

(cons 'ratatouille 'baguette).

24 Is

(cons 'ratatouille 'baguette)

the same

(Pair Atom Atom)

as

(cons 'baguette 'baguette)?

The **car** of

(cons 'ratatouille 'baguette)

is 'ratatouille, while the **car** of

(cons 'baguette 'baguette)

is 'baguette.

So we have no reason to believe that they are the same (Pair Atom Atom).

---

25 How can

(**cdr**
  (cons 'ratatouille 'baguette))

be described?

It is an

Atom.

---

26 Expressions that describe other expressions, such as Atom, are called *types*.

Is (Pair Atom Atom)† a type?

†When a name, such as Pair or Atom, refers to a type, it starts with an upper-case letter.

Yes, because it describes pairs where the **car** and **cdr** are both Atoms.

---

27 The third form of judgment is

____ is a type.

This means that both

Atom is a type

and

(Pair Atom Atom) is a type

are judgments.

The Law of Atom

Atom **is a type.**

Is

'courgette is a type

a judgment?

28 It is a judgment, but we have no reason to believe it because 'courgette doesn't describe other expressions.

Are Atom and Atom the same type?

29 Presumably. They certainly look like they should be.

The fourth and final form of judgment is

____ and ____ are the same type.

30 Ah, so

Atom and Atom are the same type

is a judgment, and we have reason to believe it.

The Four Forms of Judgment

1. _____ is a _____. **2. _____ is the same _____ as _____.**

3. _____ is a type. **4. _____ and _____ are the same type.**

Is this a judgment?

Atom and (Pair Atom Atom) are the same type.

31 Yes, it is a judgment, but there is no reason to believe it.

Are

(Pair Atom Atom)

and

(Pair Atom Atom)

the same type?

32 That certainly seems believable.

33 Judgments are acts of knowing, and believing is part of knowing.

Aren't judgments sentences?

34 Sentences get their meaning from those who understand them. The sentences capture thoughts that we have, and thoughts are more important than the words we use to express them.

Ah, so coming to know that

    (Pair Atom Atom)

and

    (Pair Atom Atom)

are the same type was a judgment?

35 It was.

Is 'pêche the same 'fruit as 'pêche?

Good question.

Is 'pêche a 'fruit?

36 No, it is not, because

    'fruit is a type

is not believable.

Some forms of judgment only make sense after an earlier judgment.†

†This earlier judgment is sometimes called a *presupposition*.

Which are these?

37 To ask whether an expression is described by a type, one must have already judged that the supposed type is a type. To ask whether two expressions are the same according to a type, one must first judge that both expressions are described by the type.†

What judgment is necessary before asking whether two expressions are the same type?

†To describe the expressions, the supposed type must also be a type.

To ask whether two expressions are the same type, one must first judge that each expression is, in fact, a type.

Is

```
(car
  (cons 'ratatouille 'baguette))
```

the same

```
Atom
```

as

```
'ratatouille?
```

38 This looks familiar. Presumably, **car** finds the **car** of a pair, so they *are* the same.

Is

```
(cdr
  (cons 'ratatouille 'baguette))
```

the same

```
Atom
```

as

```
'baguette?
```

39 It must certainly be, because the **cdr** of the pair is 'baguette.

So

```
(car
  (cons
    (cons 'aubergine 'courgette)
    'tomato))
```

is a . . .

40 . . . (Pair Atom Atom),

because

```
(cons 'aubergine 'courgette)
```

is a pair whose **car** is the Atom 'aubergine and whose **cdr** is the Atom 'courgette.

Is

```
(car
  (cdr
    (cons 'ratatouille
      (cons 'baguette 'olive-oil))))
```

the same

```
Atom
```

as

```
'baguette?
```

41 Yes, it is.

42 Expressions that are written differently may nevertheless be the same, as seen in frames 39–41. One way of writing these expressions is more direct than the others.

'baguette certainly seems more direct than

```
(car
  (cdr
    (cons 'ratatouille
      (cons 'baguette 'olive-oil)))).
```

---

43 The *normal form* of an expression is the most direct way of writing it. Any two expressions that are the same have identical normal forms, and any two expressions with identical normal forms are the same.

Is 'olive-oil the normal form of

```
(cdr
  (cdr
    (cons 'ratatouille
      (cons 'baguette 'olive-oil))))?
```

---

44 That question is incomplete.

Sameness is always according to a type, so normal forms are also determined by a type.

Is 'olive-oil the normal form of the Atom

```
(cdr
  (cdr
    (cons 'ratatouille
      (cons 'baguette 'olive-oil))))?
```

---

45 Yes, it is.

Is

```
(cons 'ratatouille 'baguette)
```

a normal

```
(Pair Atom Atom)?†
```

† *Normal* is short for *in normal form.*

Yes, (cons 'ratatouille 'baguette) is normal.

Does every expression have a normal form?

46
It does not make sense to ask whether an expression has a normal form without specifying its type.

Given a type, however, every expression described by that type does indeed have a normal form determined by that type.

If two expressions are the same according to their type, then they have identical normal forms. So this must mean that we can check whether two expressions are the same by comparing their normal forms!

Normal Forms

Given a type, every expression described by that type has a *normal form*, which is the most direct way of writing it. If two expressions are the same, then they have identical normal forms, and if they have identical normal forms, then they are the same.

47
What is the normal form of

```
(car
  (cons
    (cons 'aubergine 'courgette)
    'tomato))?
```

What about the type?

If the type is

```
(Pair Atom Atom),
```

then the normal form is

```
(cons 'aubergine 'courgette).
```

48
Nice catch!

The previous description of what it means to be a

```
(Pair Atom Atom)
```

is incomplete. It must mean . . .

. . . to be a pair whose **car** is an Atom, and whose **cdr** is also an Atom, *or an expression that is the same as such a pair.*

Normal Forms and Types

Sameness is always according to a type, so normal forms are also determined by a type.

Is

```
(car
  (cons
    (cons 'aubergine 'courgette)
    'tomato))
```

the same

```
(Pair Atom Atom)
```

as

```
(cons 'aubergine 'courgette)?
```

49 Yes, the two expressions are the same (Pair Atom Atom) because the normal form of

```
(car
  (cons
    (cons 'aubergine 'courgette)
    'tomato))
```

is

```
(cons 'aubergine 'courgette).
```

Why is

```
(cons 'aubergine 'courgette)
```

the same (Pair Atom Atom) as

```
(cons 'aubergine 'courgette)?
```

50 That seems pretty obvious.

Yes, but not everything that *seems* obvious is *actually* obvious.†

Frame 23 describes what it means for one expression to be the same

```
(Pair Atom Atom)
```

as another.

†In Lisp, two uses of `cons` with the same atoms yield pairs that are not `eq`. Here, however, they cannot be distinguished in any way.

51 Both

```
(cons 'aubergine 'courgette)
```

and

```
(cons 'aubergine 'courgette)
```

have cons at the top. 'aubergine is the same Atom as 'aubergine, and 'courgette is the same Atom as 'courgette.

Both expressions have the same **car** and have the same **cdr**. Thus, they are the same

```
(Pair Atom Atom).
```

The First Commandment of cons

Two cons-expressions are the same (Pair A D) if their cars are the same A and their cdrs are the same D. Here, A and D stand for any type.

Perfect.

What is the normal form of

```
(Pair
  (cdr
    (cons Atom 'olive))
  (car
    (cons 'oil Atom)))?
```

52 It is (Pair 'olive 'oil), right?

Actually, the expression

```
(Pair
  (cdr
    (cons Atom 'olive))
  (car
    (cons 'oil Atom)))
```

is neither described by a type, nor is it a type, so asking for its normal form is meaningless.†

†Expressions that cannot be described by a type and that are not themselves types are also called *ill-typed*.

53 Why not?

Because Pair is not a type when its arguments are actual atoms.

It is only an expression when its arguments are types such as Atom.

54 Does that mean that Pair can't be used together with **car** and **cdr**?

No, not at all. What is the normal form of

```
(Pair
  (car
    (cons Atom 'olive))
  (cdr
    (cons 'oil Atom)))?
```

55 What is its type? Normal forms are according to a type.

Types themselves also have normal forms. If two types have identical normal forms, then they are the same type, and if two types are the same type, then they have identical normal forms.

56 The normal form of the type

```
(Pair
  (car
    (cons Atom 'olive))
  (cdr
    (cons 'oil Atom)))?
```

must be (Pair Atom Atom) because the normal form of

```
(car
  (cons Atom 'olive))
```

is Atom and the normal form of

```
(cdr
  (cons 'oil Atom))
```

is Atom.

Normal Forms of Types

Every expression that is a type has a normal form, which is the most direct way of writing that type. If two expressions are the same type, then they have identical normal forms, and if two types have identical normal forms, then they are the same type.

That's it. Now we know that

```
(cons 'ratatouille 'baguette)
```

is also a

```
(Pair
  (car
    (cons Atom 'olive))
  (cdr
    (cons 'oil Atom)))
```

because . . .

57 . . . the normal form of

```
(Pair
  (car
    (cons Atom 'olive))
  (cdr
    (cons 'oil Atom)))
```

is

```
(Pair Atom Atom),
```

and

```
(cons 'ratatouille 'baguette)
```

is a

```
(Pair Atom Atom).
```

Another way to say this is that

```
(Pair
  (car
    (cons Atom 'olive))
  (cdr
    (cons 'oil Atom)))
```

and

```
(Pair Atom Atom)
```

are the same type.

58 If an expression is a

```
(Pair
  (car
    (cons Atom 'olive))
  (cdr
    (cons 'oil Atom)))
```

then it is also a

```
(Pair Atom Atom)
```

because those two types are the same type.

Similarly, if an expression is a

```
(Pair Atom Atom)
```

then it is also a

```
(Pair
  (car
    (cons Atom 'olive))
  (cdr
    (cons 'oil Atom)))
```

because those two types are the same type.

59 And likewise for

```
(Pair
  Atom
  (cdr
    (cons 'oil Atom))),
```

which is also the same type.

Is '6 an Atom?

60 No. We have no reason to believe that

'6 is an Atom,

because the digit 6 is neither a letter nor a hyphen, right?

Right. Is

(cons '17 'pepper)

a

(Pair Atom Atom)?

61 No, because the **car** of (cons '17 'pepper) is '17, which is *not* an Atom.

It sure would be natural to have numbers, though.

Numbers are certainly convenient. Besides Atom and Pair, we can check whether something is a Nat.

62 Let's give it a try.

Is 1 a Nat?†

† Nat is a short way of writing *natural number*.

63 Yes, 1 is a Nat.

Is 1729 a Nat?

64 Yes, 1729 is a Nat. Not only is it a Nat, it's also famous!†

† Thank you, Srinivasa Ramanujan (1887–1920) and Godfrey Harold Hardy (1877–1947).

Is −1 a Nat?

65 Hmm. Sure?

No, it isn't. What about −23?

66 It's not very clear.

Positive numbers are Nats.

67 Ah, then −23 is not a Nat?

We prefer a positive point of view.

What is the smallest Nat?

68 Isn't 0 a natural number?

Oh yeah, one can't always be positive.†
How can one get the rest of the Nats?

†The number 1, however, *is* always positive.

69 One can use our old friend add1. If n is a Nat, then (add1 n) is also a Nat, and it is always a positive Nat even if n is 0.

How many Nats are there?

Lots!

70 Is there a largest Nat?

No, because one can always …

71 … add one with add1?

That's right!†

Is 0 the same Nat as 26?

†Thank you, Giuseppe Peano (1838–1932).

72 Clearly not.

Is (+ 0 26)† the same as 26?

†Even though we have not explained + yet, use your knowledge of addition for now.

73 That question has no meaning. But can we ask if they are the same Nat?

Of course.

Is (+ 0 26) the same Nat as 26?

74 Yes, because the normal form of (+ 0 26) is 26, and 26 is certainly the same as 26.

What does zero mean?

75 Does zero mean the same as 0?

76 In Pie, zero and 0 are two ways to write the same Nat.

Is one the same Nat as 1?

Well, if zero is the same Nat as 0, that would make sense.

---

77 Actually, one has no meaning. But (add1 zero) is another way to write the number 1.

It is possible to make ***one*** be (add1 zero) by *defining* it.

```
(define one
  (add1 zero))
```

Why is the box around the definition dashed?

---

78 A dashed box means that there is something the matter with the definition, so the definition in the dashed box is not available for use later.

What is the matter with that definition?

It looks okay.

---

79 When defining a name, it is necessary to first **claim** the name with a type, and ***one*** is a Nat.

```
(claim one
  Nat)
(define one
  (add1 zero))
```

So ***two*** can be defined as

```
(claim two
  Nat)
(define two
  (add1 one))
```

Claims before Definitions

Using define to associate a name with an expression requires that the expression's type has previously been associated with the name using claim.

If 1 is another way of writing (add1 zero), what is another way of writing 4?

80 Shouldn't it be

```
(add1
  (add1
    (add1
      (add1 zero))))?
```

Can't we define ***four*** to mean that?

Of course.

```
(claim four
  Nat)
(define four
  (add1
    (add1
      (add1
        (add1 zero)))))
```

Is there another way of writing 8 as well?

81 It must be

```
(add1
  (add1
    (add1
      (add1
        (add1
          (add1
            (add1
              (add1 zero)))))))).
```

Is 8 normal?

82 It seems that way. But why is 8 normal?

---

8 is *normal* because its top,[†] add1 is a *constructor*, and because the argument tucked under the top add1, namely 7, is normal.

[†]The top add1 in frame 81 is underlined this one time for emphasis.

83 Why is 7, also written

```
(add1
  (add1
    (add1
      (add1
        (add1
          (add1
            (add1 zero))))))),
```

normal?

---

7 is normal for the very same reason.

84 This must mean that zero is normal, or else (add1 zero) would not be normal.

What is at the top of zero?

85 It must be zero.

zero is normal because the top zero is a constructor, and it has no arguments.

Is

```
(add1
  (+ (add1 zero)
     (add1
       (add1 zero))))
```

normal?

86 No, because + is not a constructor.

An expression with a constructor at the top is called a *value*.†

Even though

```
(add1
  (+ (add1 zero)
     (add1
       (add1 zero))))
```

is not normal, it is a value.

†Values are also called *canonical* expressions.

87 It is not normal because

```
(+ (add1 zero)
   (add1
     (add1 zero)))
```

is not the most direct way of writing 3.

Values

An expression with a constructor at the top is called a *value*.

88 Here's another expression that is not normal.

```
(+ (add1
     (add1 zero))
   (add1 zero))
```

Is this the most direct way of writing 3?

No.

What exactly is a constructor?

---

89 Some expressions, such as Nat or (Pair Nat Atom), are types.

Part of explaining a new type is to say what its constructors are. The constructor expressions are the direct ways of building expressions with the new type.

What are some examples of constructors?

---

90 The constructors of Nat are zero and add1, while the constructor of Pair is cons.

What is the relationship between values and normal forms?

---

91 In a value, the top constructor's arguments need not be normal, but if they are, then the entire constructor expression is in normal form.

Are all values normal?

No.

```
(add1
  (+ (add1 zero)
     (add1
       (add1 zero))))
```

and

```
(add1
  (+ (add1 zero) (add1 one)))
```

are values, but they are not normal.

Values and Normal Forms

Not every value is in normal form. This is because the arguments to a constructor need not be normal. Each expression has only one normal form, but it is sometimes possible to write it as a value in more than one way.

What expressions can be placed in the empty box to make this expression *not* a Nat value?

(add1
  $\boxed{\qquad\qquad\qquad}$)

92 How about 'aubergine?

Indeed,

```
(add1
  'aubergine)
```

is not a Nat value because 'aubergine is an Atom, not a Nat.

When filling in boxes, the expectation is that the resulting expression is described by a type.

93 If any Nat expression is placed in the box, however, the whole expression *is* a value. The whole expression has add1 at the top, and add1 is a Nat constructor.

Finding a value that is the same as some starting expression is called *evaluation.*

94 What about the type? Sameness, after all, requires types.

From time to time, when talking about sameness, we do not explicitly mention a type. Nevertheless, a type is always intended, and can be discovered by reading carefully.

95 Doesn't evaluation refer to finding the *meaning* of an expression, not just some simpler expression?

Not here. Expressions do not refer to some external notion of meaning—in Pie, there is nothing but expressions and what we judge about them.[†]

[†]In Lisp, values are distinct from expressions, and the result of evaluation is a value.

96 That is a new way of seeing evaluation.

Why is there a difference between normal forms and values?

Everything Is an Expression

In Pie, values are also expressions. Evaluation in Pie finds an expression, not some other kind of thing.

A normal expression has no remaining opportunities for evaluation. Usually, expressions that are normal are easier to understand. Finding a value is often enough, however, because the top constructor can be used to determine what must happen next.

97 If finding a value is often enough, does that mean we are free to find the value and stop whenever we want?

Yes, assuming that specific information about the constructor's arguments is never needed.

Is

```
(add1
  (+ (add1 zero)
     (add1
       (add1 zero))))
```

the same Nat as ***four***?

98 Here is a possible answer.

They are not the same Nat because

```
(add1
  (+ (add1 zero)
     (add1
       (add1 zero))))
```

is a value, and it certainly does not look like the variable ***four***. Finding the value of ***four*** does not help, because ***four***'s value looks very different.

Good try.

But they are actually the same Nat.

99 How can that be?

Two Nat expressions, that aren't values, are the same if their values are the same. There are exactly two ways in which two Nat values can be the same: one for each constructor.

If both are zero, then they are the same Nat.

100 What about when both values have add1 at the top?

The Commandment of zero

zero is the same Nat as zero.

If the arguments to each add1 are the same Nat, then both add1-expressions are the same Nat value.

Why is
  (add1 zero)
the same
  Nat
as
  (add1 zero)?

101 Both expressions are values. Both values have add1 at the top, so their arguments should be the same Nat.

The arguments are both zero, which is a value, and zero is the same Nat value as zero.

The Commandment of add1

If n is the same Nat as k, then (add1 n) is the same Nat as (add1 k).

Why is

(add1 (**+** 0 1))

the same

Nat

as

(add1 (**+** 1 0))?

102 Both of these Nats have add1 at the top, so they are values.

They are the same because

(**+** 0 1)

is the same

Nat

as

(**+** 1 0).

Why is (**+** 0 1) the same Nat as (**+** 1 0)?

103 These Nats are not values, so to determine whether they are the same, the first step is to find their values.

Both expressions have (add1 zero) as a value, and frame 101 explains why

(add1 zero)

is the same

Nat

as

(add1 zero).

That's right.

104 Does this mean that ***four*** could have been defined like this?

```
(define four
  (add1
    (+ (add1 zero)
       (add1
         (add1 zero)))))
```

Why is that box dashed?

105 ***four*** is already defined, and can't be defined again.

Definitions Are Forever

Once a name has been claimed, it cannot be reclaimed, and once a name has been defined, it cannot be redefined.

And yes, ***four*** could have been defined like that initially.

In fact, no other expression could tell the difference between the two definitions of ***four*** because both define ***four*** to be the same Nat.

106 Is cons a constructor?

Yes, cons constructs Pairs.

107 Is it necessary to evaluate **car**'s argument in order to evaluate a **car**-expression?

Yes. To find the value of a **car**-expression, start by finding the value of its argument.

What can be said about the argument's value?

108 The argument's value has cons at the top.

After finding the argument's value, what comes next?

109 The value is the first argument to cons.

What is the value of

```
(car
  (cons (+ 3 5) 'baguette))?
```

110 The first argument to cons is

```
(+ 3 5),
```

which is not a value.

To find the value of a car-expression, first find the value of the argument, which is (cons a d).[†] The value of

```
(car
  (cons a d))
```

is then the *value* of a.

How can the value of a cdr-expression be found?

[†] Here, a is for car and d is for cdr.

111 Like car, start by evaluating cdr's argument until it becomes (cons a d). Then, the value of

```
(cdr
  (cons a d))
```

is the value of d.

Do all constructors have arguments?

No. Recall from frame 86 that zero is a constructor.

What does it mean for two expressions to be the same (Pair Atom Nat)?

112 It must mean that the value of each expression has cons at the top. And, their cars are the same Atom and their cdrs are the same Nat.

Very good.

113 Are atoms constructors?

The atom 'bay is a constructor, and so is the atom 'leaf.

114 Are *all* atoms constructors?

Yes. Each atom constructs itself.

Does this mean that atoms are values?

115 Yes, it does, because the explanation of why

  Atom is a type

says that atoms are Atom values.

Right.

In the expression zero, what is the top constructor?

116 It must be zero, because zero is a constructor of no arguments.

117 In the expression 'garlic, what is the top constructor?

The atom 'garlic is the only constructor, so it must be the top constructor.

Is Nat a constructor, then?

---

118 No, Nat is not a constructor. zero and add1 are constructors that create *data*, while Nat *describes* data that is just zero, or data that has add1 at its top and another Nat as its argument.

Is Pair a constructor?

No, because Pair-expressions *describe* expressions with cons at the top. Constructors create *data*, not types.

What is Pair called, then?

---

119 Pair is a *type constructor* because it constructs a type. Likewise, Nat and Atom are type constructors.

Is

```
(cons zero 'onion)
```

a

```
(Pair Atom Atom)?
```

No.

Isn't it a

```
(Pair Nat Atom)?
```

---

120 Indeed it is! But

```
(cons 'zero 'onion)
```

is a

```
(Pair Atom Atom).
```

What is the type of

```
(cons 'basil
  (cons 'thyme 'oregano))?†
```

† Thank you, Julia Child (1912–2004).

Based on what we've seen, it must be a

```
(Pair Atom
  (Pair Atom Atom)).
```

---

121 Indeed it is.

All right, that's enough for now. My head is going to explode!

It might be a good idea to read this chapter one more time. Judgments, expressions, and types are the most important ideas in this book.

[122] Some fresh vegetables would be nice after all this reading.

Now go enjoy some delicious homemade ratatouille!

2

Doin' What Comes Naturally

How was the ratatouille?

1 *Très bien*, thanks for asking.

In chapter 1, there are constructors, which build values, and type constructors, which build types.

car, however, is neither a constructor nor a type constructor.

2 What is car, then?

car is an *eliminator*. Eliminators take apart the values built by constructors.

What is another eliminator?

3 If car is an eliminator, then surely cdr is also an eliminator.

Constructors and Eliminators

Constructors build values, and eliminators take apart values built by constructors.

Another way to see the difference is that values contain information, and eliminators allow that information to be used.

4 Is there anything that is both a constructor and an eliminator?

No, there is not.

It is possible to define a *function* that is as expressive as both car and cdr combined.

5 How?

It requires our old friend λ.

6 What is that? It doesn't look familiar.

7 Oops! It is also known as **lambda**.[†]

[†]λ can be optionally written `lambda`.

Oh, right, λ builds functions.

Does this mean that λ is a constructor?

---

8 Yes, it does, because every expression that looks like (λ (x_0 x …[†]) *body*) is a value.

What is the eliminator for such values?

[†]The notation x … means zero or more xs, so x_0 x … means one or more xs.

The only thing that can be done to a function is to apply it to arguments.

How can functions have an eliminator?

---

9 Applying a function to arguments *is* the function's eliminator.

Okay.

Eliminating Functions

Applying a function to arguments *is* the eliminator for functions.

---

10 What is the value of

```
(λ (flavor)
  (cons flavor 'lentils))?
```

It starts with a λ, so it is already a value.

---

11 Right.

What is the value of

```
((λ (flavor)
   (cons flavor 'lentils))
 'garlic)?
```

It must be (cons 'garlic 'lentils), if λ works the same way as **lambda** and cons is a constructor.

But doesn't this mean that cons's first argument is being evaluated, even though the cons-expression is already a value?

No, it does not, but that's a very good question. Replacing the λ-expression's *flavor* happens because the λ-expression is applied to an argument, not because of the cons.†

Every *flavor* in the body of the λ-expression is replaced with 'garlic, no matter what expression surrounds the *flavor*.

†Consistently replacing a variable with an expression is sometimes called *substitution*.

12 So this means that the value of

```
((λ (root)
   (cons root
     (cons (+ 1 2) root)))
 'potato)
```

is therefore

```
(cons 'potato
  (cons (+ 1 2) 'potato)),
```

right?

Why is there is no need to evaluate

```
(+ 1 2)
```

in the preceding frame?

13 The entire expression has cons at the top, so it is a value.

Frame 12 contains a small exaggeration. If the *root* (underlined here) in the body of the λ-expression occurs under another λ with the same name, then it is not replaced.

What is the value of

```
((λ (root)
   (cons root
     (λ (root)
       root)))
 'carrot)?
```

14 It must be

```
(cons 'carrot
  (λ (root)
    root))
```

because the inner *root* is under a λ-expression with the same name.

λ does work the same way as **lambda**, and that is indeed the right answer.

15 What about expressions that have these λ-expressions as their values?

To be an

(→ Atom
  (Pair Atom Atom))†

is to be a λ-expression that, applied to an Atom as its argument, evaluates to a

(Pair Atom Atom).

†This is pronounced "Arrow atom *pause* pair atom atom." And → can be written with two characters: ->.

---

16 Are they also

(→ (**car** (cons Atom 'pepper))
  (Pair (**cdr** (cons 'salt Atom)) Atom))?

Yes, these are also

(→ Atom
  (Pair Atom Atom))

because they too become a

(Pair Atom Atom)

when given an Atom as an argument.

---

17 It makes sense to ask what it means for two expressions to be the same Nat, the same Atom, or the same (Pair Nat Atom).

Does it also make sense to ask what it means for two expressions to be the same

(→ Nat
  Atom),

or the same

(→ (Pair Atom Nat)
  Nat)?

Yes, because

(**car**
  (cons Atom 'pepper))

is Atom and

(**cdr**
  (cons 'salt Atom))

is also Atom.

Yes, it does. Two expressions are the same

(→ Nat
  Atom)

when their values are the same

(→ Nat
  Atom).

18 Their values are λ-expressions. What does it mean for two λ-expressions to be the same

(→ Nat
  Atom)?

Two λ-expressions that expect the same number of arguments are the same if their bodies are the same. For example, two λ-expressions are the same

(→ Nat
  (Pair Nat Nat))

if their bodies are the same

(Pair Nat Nat).

19 Does this mean that

(λ (*x*)
  (cons *x* *x*))

is not the same

(→ Nat
  (Pair Nat Nat))

as

(λ (*y*)
  (cons *y* *y*))?

What is not the same about those expressions?

20 The names of the arguments are different. This usually doesn't matter, though. Does it matter here?

Two λ-expressions are also the same if there is a way to consistently rename the arguments to be the same that makes their bodies the same.†

Consistently renaming variables can't change the meaning of anything.

†Renaming variables in a consistent way is often called *alpha-conversion.* Thank you, Alonzo Church (1903–1995).

21 Is

(λ (*a* *d*)
  (cons *a* *d*))

the same

(→ Atom Atom
  (Pair Atom Atom))

as

(λ (*d* *a*)
  (cons *a* *d*))?

The Initial Law of Application

If f **is an**

$(\rightarrow Y$
$\quad X)$

and arg **is a** Y**, then**

$(f\ arg)$

is an X**.**

The Initial First Commandment of λ

Two λ-expressions that expect the same number of arguments are the same if their bodies are the same after consistently renaming their variables.

The Initial Second Commandment of λ

If f **is an**

$(\rightarrow Y$
$\quad X)$,

then f **is the same**

$(\rightarrow Y$
$\quad X)$

as

(**λ** (y)
  (f y)),

as long as y **does not occur in** f**.**

No, it is not, because consistently renaming the variables in the second λ-expression to match the arguments in the first λ-expression yields

```
(λ (a d)
  (cons d a)),
```

and (cons *d* *a*) is not the same (Pair Atom Atom) as (cons *a* *d*).

22 What about

```
(λ (y)
  (car
    (cons y y)))?
```

Is it the same

```
(→ Nat
  Nat)
```

as

```
(λ (x)
  x)?
```

The Law of Renaming Variables

Consistently renaming variables can't change the meaning of anything.

First, consistently rename *y* to *x*. Now, the question is whether

```
(car
  (cons x x))
```

is the same Nat as *x*.

23 There are precisely two ways that two expressions can be the same Nat. One way is for both their values to be zero. The other is for both their values to have add1 at the top and for the arguments to both add1s to be the same Nat.

These expressions are not Nat values because they do not have add1 at the top and they are not zero.

The value of *x* is not yet known, because the λ-expression has not been applied to an argument. But when the λ-expression has been applied to an argument, the value of *x* *is still* a Nat value because . . .

24 . . . because the λ-expression is an

```
(→ Nat
  Nat),
```

so the argument *x* can't be anything else.

Expressions that are not values and cannot *yet* be evaluated due to a variable are called *neutral*.

25 Does this mean that

    (cons y 'rutabaga)

is neutral?

No, it is not neutral, because

    (cons y 'rutabaga)

is a value.

If *x* is a (Pair Nat Atom), is

    (cdr x)

a value?

26 No, because **cdr** is an eliminator, and eliminators take apart values.

Without knowing the value of *x*, there is no way to find the value of (**cdr** *x*), so (**cdr** *x*) is neutral.

Neutral expressions make it necessary to expand our view on what it means to be the same. Each variable is the same as itself, no matter what type it has. This is because variables are only replaced *consistently*, so two occurrences of a variable cannot be replaced by values that are not the same.

27 So if we assume that *y* is a Nat, then

    (car
      (cons y 'rutabaga))

is the same Nat as *y* because the **car**-expression's normal form is *y*, and *y* is the same Nat as *y*.

Yes. And, likewise,

    (λ (x)
      (car
        (cons x x)))

is the same

    (→ Nat
      Nat)

as

    (λ (x)
      x).

28 Right, because the neutral expression *x* is the same Nat as *x*.

Is

```
(λ (x)
  (car x))
```

the same

```
(→ (Pair Nat Nat)
  Nat)
```

as

```
(λ (y)
  (car y))?
```

29 One would think so. But why?

The first step is to consistently rename *y* to *x*.

Is

```
(λ (x)
  (car x))
```

the same

```
(→ (Pair Nat Nat)
  Nat)
```

as

```
(λ (x)
  (car x))?
```

30 Yes, assuming that

```
(car x)
```

is the same Nat as

```
(car x).
```

But (**car** *x*) is not a variable, and it is not possible to find its value until *x*'s value is known.

If two expressions have identical eliminators at the top and all arguments to the eliminators are the same, then the expressions are the same. Neutral expressions that are written identically are the same, *no matter their type.*

31 So

```
(car x)
```

is indeed the same Nat as

```
(car x),
```

assuming that *x* is a (Pair Nat Nat).

The Commandment of Neutral Expressions

Neutral expressions that are written identically are the same, *no matter their type.*

Is

```
(λ (a d)
  (cons a d))
```

an

```
(→ Atom Atom
  (Pair Atom Atom))?
```

32 What does having more expressions after the → mean?

The expressions after an →, except the last[†] one, are the types of the arguments. The last one is the value's type.

[†]The last one is preceded by a *pause* when pronounced.

33 Okay, then,

```
(λ (a d)
  (cons a d))
```

is an

```
(→ Atom Atom
  (Pair Atom Atom)).
```

These expressions are certainly getting long.

One way to shorten them is the careful use of **define**, as in frame 1:77, which allows short names for long expressions.

34 Good idea.

Suppose that the constructor cons is applied to 'celery and 'carrot. We can refer to that value as ***vegetables***.

```
(claim vegetables
  (Pair Atom Atom))
(define vegetables
  (cons 'celery 'carrot))
```

From now on, whenever the name ***vegetables*** is used, it is the same

(Pair Atom Atom)

as

(cons 'celery 'carrot),

because that is how ***vegetables*** is **define**d.

35 Why does it say

(Pair Atom Atom)

after **claim**?

The Law and Commandment of define

Following

(claim *name X*) and (define *name expr*),

if

***expr* is an *X*,**

then

name* is an *X

and

***name* is the same *X* as *expr*.**

(Pair Atom Atom) describes how we can use ***vegetables***—we know that (**car** ***vegetables***) is an Atom, and also that (cons 'onion ***vegetables***) is a

(Pair Atom
  (Pair Atom Atom)).[†]

[†] They are a good start for lentil soup, too.

36 Ah, that makes sense.

Is

vegetables

the same

(Pair Atom Atom)

as

(cons (**car** ***vegetables***)
      (**cdr** ***vegetables***))?

37 Yes, because the value of each expression is a pair whose **car** is 'celery and whose **cdr** is 'carrot.

In fact, whenever

p is a (Pair Atom Atom),

then

p is the same

(Pair Atom Atom)

as

(cons (**car** p) (**cdr** p)).

Finding the values of (**car** p) and (**cdr** p) is not necessary.

38 That seems reasonable.

The Second Commandment of cons

If p is a (Pair A D), then it is the same (Pair A D) as (cons (car p) (cdr p)).

Is this definition allowed?

```
(claim five
  Nat)
(define five
  (+ 7 2))
```

39 What?

It is allowed, even though it is probably a foolish idea.

What would be the normal form of

(**+** *five* 5)?

40 It must be 10 because five plus 5 is ten.

Try again. Remember the strange definition of ***five*** . . .

41 . . . Oh, right, it would be 14 if ***five*** were defined to be 9.

That's right

42 Is *this* definition allowed? It doesn't seem particularly foolish.

```
(claim zero
  Nat)
(define zero
  0)
```

It is not as foolish as defining ***five*** to mean 9, but it is also not allowed.

Names that are already used, whether for constructors, eliminators, or previous definitions, are not suitable for use with **claim** or **define**.

43 Okay.

Names in Definitions

In Pie, only names that are not already used, whether for constructors, eliminators, or previous definitions, can be used with claim or define.

There is an eliminator for Nat that can distinguish between Nats whose values are zero and Nats whose values have add1 at the top. This eliminator is called **which-Nat**.

44 How does **which-Nat** tell which of the two kinds of Nats it has?

A **which-Nat**-expression has three arguments: *target*, *base*, and *step*:

(**which-Nat** *target*
  base
  step).

which-Nat checks whether

target is zero.

If so,

the value of the **which-Nat**-expression is

the value of *base*.

Otherwise, if

target is (add1 *n*),

then

the value of the **which-Nat**-expression is

the value of (*step* *n*).

45 So **which-Nat** both checks whether a number is zero and removes the add1 from the top when the number is not zero.

Indeed.

What is the normal form of

```
(which-Nat zero
  'naught
  (λ (n)
    'more))?
```

46 It must be 'naught because the target, zero, is zero, so the value of the which-Nat-expression is *base*, which is 'naught.

Why is *n* written dimly?

The dimness indicates that *n* is not used in the body of the λ-expression. Unused names are written dimly.

47 Why isn't it used?

---

which-Nat offers the possibility of using the smaller Nat, but it does not demand that it be used. But to offer this possibility, **which-Nat**'s last argument must accept a Nat.

48 Okay.

Dim Names

Unused names are written dimly, but they do need to be there.

What is the value of

```
(which-Nat 4
  'naught
  (λ (n)
    'more))?
```

49 It must be 'more because 4 is another way of writing (add1 3), which has add1 at the top. The normal form of

```
((λ (n)
   'more)
 3)
```

is

```
'more.
```

The Law of which-Nat

If *target* **is a** Nat, *base* **is an** X, **and** *step* **is an**

```
(→ Nat
  X),
```

then

```
(which-Nat target
  base
  step)
```

is an X.

The First Commandment of which-Nat

If (which-Nat zero
    base
    step)
is an X, **then it is the same** X **as** *base*.

The Second Commandment of which-Nat

If (which-Nat (add1 n)
    base
    step)
is an X, **then it is the same** X **as** (*step* n).

50 What is the normal form of

```
(which-Nat 5
  0
  (λ (n)
    (+ 6 n)))?
```

Is it 11 because

```
((λ (n)
   (+ 6 n))
 5)
```

is 11?

51 The normal form is 10 because the value of a **which-Nat** expression is determined by the Nat tucked under the target as an argument to the step.

Ah, so the normal form is 10 because

```
((λ (n)
   (+ 6 n))
 4)
```

is 10.

52 Define a function called ***gauss***[†] such that (***gauss*** *n*) is the sum of the Nats from zero to *n*.

What is the type of ***gauss***?

[†] Carl Friedrich Gauss (1777–1855), according to folklore, figured out that $0+\cdots+n = \frac{n(n+1)}{2}$ when he was in primary school and was asked to sum a long series.

The sum of Nats is a Nat.

```
(claim gauss
  (→ Nat
    Nat))
```

53 Right.

Now define it.

How?

54 The first step is to choose an example argument. Good choices are somewhere between 5 and 10—they're big enough to be interesting, but small enough to be manageable.

How about 5, then?

Sounds good.

What should the normal form of

(***gauss*** 5)

be?

55 It should be $0+1+2+3+4+5$, which is 15.

The next step is to shrink the argument.

(***gauss*** 4), which is 10, is almost (***gauss*** 5), which is 15.

A white box around a gray box contains unknown code that wraps a known expression. What should be in this white box to get

(***gauss*** 5)

from

(***gauss*** 4)?

$\boxed{\qquad\boxed{(\textit{gauss}\ 4)}\qquad}$

56 5 must be added to (***gauss*** 4), and our sum is 15.

$\boxed{(\mathbf{+}\ 5\ \boxed{(\textit{gauss}\ 4)}\ )}$

Next, make it work for any Nat that has add1 at the top.

If *n* is a Nat, then what should be in the box to get

(***gauss*** (add1 *n*))

from

(***gauss*** *n*)?

$\boxed{\qquad\boxed{(\textit{gauss}\ n)}\qquad}$

Remember that 5 is another way of writing (add1 4).

57 The way to find (***gauss*** (add1 *n*)) is to replace 4 with *n* in the preceding frame's answer.

$\boxed{(\mathbf{+}\ (\text{add1}\ n)\ \boxed{(\textit{gauss}\ n)}\ )}$

What about zero?

What is (***gauss*** zero)?

58 Clearly it is 0.

Now define ***gauss***.

Remember the white and gray boxes.

59 Piece of cake! The name, *n-1*, suggests that it represents a Nat that is tucked under (or one less than) *n*.

```
(define gauss
  (λ (n)
    (which-Nat n
      0
      (λ (n-1)
        (+ (add1 n-1) (gauss n-1))))))
```

Nice try, and it would deserve a solid box if recursion were an option, but recursion is *not* an option.

60 Why not?

Because recursion is not an option.

61 Why not?

Because recursion is not an option.

62 Okay. Please explain why recursion is not an option.

Recursion is not an option because every expression must have a value. Some recursive definitions make it possible to write expressions that do not have values.

63 What is an example of a recursive definition and an expression without a value?

forever is such a definition.

```
(claim forever
  (→ Nat
    Atom))
(define forever
  (λ (and-ever)
    (forever and-ever)))
```

What is the value of (*forever* 71)?

64 Good question.

Why does it have a dashed box?

Recursion is not an option, so recursive definitions (like *forever*) stay dashed forever.

65 But what about definitions like ***gauss*** that need recursion?

There is a safe alternative to recursive definitions. This alternative allows ***gauss***, along with many similar definitions, to be written without including the name ***gauss***.

66 Here is the start of a safe alternative definition of ***gauss***.

```
(define gauss
  (λ (n)
    [gauss is not an option here!]))
```

As far as it goes, it is correct. The point is that ***gauss*** cannot occur in its own definition.

67 Now it is clear what is meant by "Recursion is not an option."

Does this mean that it is impossible to write ***gauss*** in Pie?

It *is* possible to write ***gauss*** in Pie, but **which-Nat** and **define** are not up to the task. A different eliminator is needed, but the time is not yet ripe.

68 Patience is a virtue.

It is also possible to define shorter names for expressions such as (Pair Nat Nat).

69 What is the **claim** in this case?

70 Are types values?

Another good question!

Expressions such as Atom, Nat, and (Pair Atom Nat), are types, and each of these types is a $\mathcal{U}$.[†]

[†] $\mathcal{U}$, pronounced "you," is short for *universe*, because it describes *all* the types (except for itself).

---

71 Are all types values?

Some types are values.

An expression that is a type is a value when it has a type constructor at its top. So far, we have seen the type constructors Nat, Atom, Pair, $\to$, and $\mathcal{U}$.

Type Values

An expression that is described by a type is a value when it has a constructor at its top. Similarly, an expression that is a type is a value when it has a type constructor at its top.

---

72 Which expressions are described by

```
(car
  (cons Atom 'prune))?
```

No.

```
(car
  (cons Atom 'prune))
```

is a type, but not a value, because **car** is neither a constructor nor a type constructor.

Because

  (**car**
    (cons Atom 'prune))

and

  Atom

are the same type,

  (**car**
    (cons Atom 'prune))

describes the same expressions as Atom.

73 What is the difference between type constructors and constructors?

Type constructors construct types, and constructors (or *data* constructors) construct values that are described by those types.

Judging that an expression is a type requires knowing its constructors. But the meaning of $\mathcal{U}$ is not given by knowing all the type constructors, because new types can be introduced.

74 Is (cons Atom Atom) a $\mathcal{U}$?

No, but

  (cons Atom Atom)

is a

  (Pair $\mathcal{U}$ $\mathcal{U}$).

An atom, like 'plum, is an Atom. On the other hand, Atom is not an Atom, it's a type described by $\mathcal{U}$.

75 Let's think about (Pair Atom Atom).

Is

  (cons Atom Atom)

a

  (Pair Atom Atom)?

No, it is not, because Atom is a type, not an Atom.

76 Is $\mathcal{U}$ a $\mathcal{U}$?

No, but $\mathcal{U}$ is a type. No expression can be its own type.†

†It would be possible for $\mathcal{U}$ to be a $\mathcal{U}_1$, and $\mathcal{U}_1$ to be a $\mathcal{U}_2$, and so forth. Thank you, Bertrand Russell (1872–1970), and thanks, Jean-Yves Girard (1947–). Here, a single $\mathcal{U}$ is enough because $\mathcal{U}$ is not described by a type.

77 Is every expression that is a $\mathcal{U}$ also a type?

Yes, if X is a $\mathcal{U}$, then X is a type.

78 Is every type described by $\mathcal{U}$?

Every $\mathcal{U}$ Is a Type

Every expression described by $\mathcal{U}$ is a type, but not every type is described by $\mathcal{U}$.

Every expression described by $\mathcal{U}$ is a type, but not every expression that is a type is described by $\mathcal{U}$.

79 Is

(cons Atom Nat)

a

(Pair $\mathcal{U}$ $\mathcal{U}$)?

Yes, it is.

Define ***Pear*** to mean the type of pairs of Nats.

80 That must be

```
(claim Pear
  𝒰)
(define Pear
  (Pair Nat Nat))
```

From now on, the meaning of ***Pear*** is

(Pair Nat Nat).

The name has only four characters, but the type has fourteen.

Is ***Pear*** the same type as (Pair Nat Nat), everywhere that it occurs?

81 Yes, by the Commandment of **define**.

Is (cons 3 5) a ***Pear***?

82 Yes, because

(cons 3 5)

is a

(Pair Nat Nat),

and

Pear

is *defined* to be precisely that type.

That's a good point.

83 Is ***Pear*** a value?

No. Names defined with **define** are neither type constructors nor constructors. Thus, they are not values.

Is there an eliminator for ***Pear***?

84 Does that mean an eliminator that takes apart values of type ***Pear***?

Yes.

An eliminator for ***Pear*** must allow the information in values with type ***Pear*** to be used.

85 What does it mean to allow the information to be used?

An eliminator for ***Pear*** that allows the information in any ***Pear*** to be used is one that applies a function to the two Nat arguments in the ***Pear***.

86 Okay.

Which functions can be applied to two Nats as arguments?

87 Here's one: **+**.

88 What about an expression that exchanges the Nats?

How about

```
(λ (a d)
  (cons d a))?
```

89 Very good. What about an expression that extracts the first Nat from a ***Pear***?

That must be

```
(λ (a d)
  a).
```

90 Very close. Actually, it would be

```
(λ (a d)
  a).
```

Okay. But the expression is correct except for dimness, right?

91 Indeed. To get a value of type X[†] from a ***Pear***, one must have an expression of type

```
(→ Nat Nat
  X).
```

What type does **+** have?

[†] X can be any type at all.

It takes two Nats and produces a Nat, so it must be

```
(→ Nat Nat
  Nat).
```

92 That's right.

What would be the type of

```
(λ (a d)
  (cons d a)),
```

when both *a* and *d* are Nats?

Clearly it must be

```
(→ Nat Nat
  Pear),
```

which is the same as

```
(→ Nat Nat
  (Pair Nat Nat)).
```

How can a

λ-expression

be used with a

Pear?

Definitions Are Unnecessary

Everything can be done without definitions, but they do improve understanding.

Try this:

```
(claim Pear-maker
  U)
(define Pear-maker
  (→ Nat Nat
    Pear))

(claim elim-Pear
  (→ Pear Pear-maker
    Pear))
(define elim-Pear
  (λ (pear maker)
    (maker (car pear) (cdr pear))))
```

Is there a way to write the **claim** of ***elim-Pear*** without using ***Pear*** or ***Pear-maker***?

93 Yes, by replacing ***Pear-maker*** and both ***Pear***s with their respective definitions.

```
(claim elim-Pear
  (→ (Pair Nat Nat)
      (→ Nat Nat
        (Pair Nat Nat))
    (Pair Nat Nat)))
```

The names ***Pear*** and ***Pear-maker*** were never necessary. Is the name ***elim-Pear*** necessary?

When are definitions necessary?

94 Never!

That's right. ***elim-Pear*** is the same as the λ-expression that is its definition.

What is the value of

```
(elim-Pear
  (cons 3 17)
  (λ (a d)
    (cons d a)))?
```

95 How about

```
((λ (pear maker)
    (maker (car pear) (cdr pear)))
 (cons 3 17)
 (λ (a d)
    (cons d a)))?
```

That's a good start. But it is not yet a value.

96 The value is (cons 17 3).

Because ***elim-Pear*** means the same thing as the λ-expression in its definition,

```
(car
  (cons 3 17))
```

is the same Nat as 3,

```
(cdr
  (cons 3 17))
```

is the same Nat as 17, and

```
((λ (a d)
   (cons d a))
 3 17)
```

is the same ***Pear*** as

```
(cons 17 3).
```

What does it mean to add two pears?

97 Is it just adding the first and second Nats of each pear?

Good guess.

What type does this *pearwise* addition have?

98 The type is

```
(→ Pear Pear
  Pear),
```

right?

How can pearwise addition be defined using ***elim-Pear***?

99 That's pretty hard.

Won't it be necessary to eliminate both pears because both of their Nats are part of the result?

Indeed.

Define ***pearwise+***, so that

```
(pearwise+
  (cons 3 8)
  (cons 7 6))
```

is the same ***Pear*** as

```
(cons 10 14).
```

100 First, split *anjou* and *bosc* into their respective parts, then add their first parts and their second parts.

```
(claim pearwise+
  (→ Pear Pear
    Pear))
(define pearwise+
  (λ (anjou bosc)
    (elim-Pear anjou
      (λ (a1 d1)
        (elim-Pear bosc
          (λ (a2 d2)
            (cons
              (+ a1 a2)
              (+ d1 d2))))))))
```

It might be a good idea to take a break, then come back and re-read this chapter.

101 Yes, that does seem like a good idea.

But how can we ever get to chapter 3?

By getting to chapter 3.

102 It's a good thing recursion is not an option.

Go eat two tacos de nopales
but look out for the spines.

This page is intentionally left blank.[†]

[†]Recursion can be subtle. Apologies to Guy L. Steele Jr., whose thesis inspired this joke.

A Forkful of Pie

It's time to play with Pie.	1	Isn't it impolite to play with your food?
While pie is indeed a delicious food, Pie is a language, and a little playing around with it won't hurt.	2	Let's get started.
Using Pie is very much like a conversation: it accepts claims, definitions, and expressions and it replies with feedback.	3	What sort of feedback?
For claims and definitions, the feedback is whether they are meaningful. For expressions, the feedback is also the expression's type and normal form.	4	What if they are not meaningful?
Pie explains what is wrong with them, and sometimes adds a helpful hint.	5	What might be wrong with an expression?
Eat your vegetables before the Pie. Try typing `'spinach` and see what happens.	6	Pie responds with `(the Atom 'spinach)`. What does `the` mean here?
It means that 'spinach is an Atom. In Pie, an expression must either be a type or be described by a type. Pie can find the types of many expressions on its own, including atoms.	7	What about (car 'spinach)?

That expression is not described by a type because 'spinach is not a pair.

8 Can Pie always determine the type that describes an expression?

No, sometimes Pie needs help.

In that case, use a `the`-expression[†] to tell Pie which type is intended.

[†] `the`-expressions are also referred to as *type annotations*.

9 For example?

Pie cannot determine the type of a cons-expression that stands alone.

10 Why not? Isn't it obvious that

(cons 'spinach 'cauliflower)

is a

(Pair Atom Atom)?

It is obvious to us, but later, cons becomes more magnificent, and that increased power means that the type cannot be determined automatically.

11 How, then, can Pie determine that

(cons 'spinach 'cauliflower)

is a pair?

Try this:

```
(the (Pair Atom Atom)
  (cons 'spinach 'cauliflower)).
```

12 So a `the`-expression associates an expression with its type, both in Pie's feedback and in the expressions we write.

The Law of `the`

If X is a type and e is an X, then

(`the` X e)

is an X.

There are two kinds of expressions in Pie: those for which Pie can determine a type on its own, and those for which Pie needs our help.

13 Are there other ways to help Pie with types?

Yes. In chapter 1, **claim** is required before its associated **define**, which tells Pie what type to use for the definition's meaning.

14 Why not just use **claim** and **define** every time Pie can't determine the type of an expression?

That would work, but keeping all the names straight might be exhausting.

15 Are there any other ways to help Pie find a type?

There is one more way. If an expression is used somewhere where only one type makes sense, then that type is used.

16 What is an example of this?

While checking that

```
(the (Pair Atom
       (Pair Atom Atom))
  (cons 'spinach
    (cons 'kale 'cauliflower)))
```

is described by a type, Pie uses

```
(Pair Atom Atom)
```

as a type for

```
(cons 'kale 'cauliflower).
```

17 Here, the inner `cons` doesn't need a `the` because its type is coming from the outer `cons`'s type.

Are expressions with `the` at the top values?

No.

The value of

(`the` *X* *e*)

is the value of *e*.

18 So what is the value of

```
(car
  (the (Pair Atom Nat)
    (cons 'brussels-sprout 4)))?
```

The Commandment of `the`

If X is a type and e is an X, then

(`the` X e)

is the same X as e.

The value is one little round 'brussels-sprout.

Now try this:

```
U
```

19 Pie said:

```
U
```

Why wasn't it

```
(the U U)?
```

$\mathcal{U}$ *is* a type, but it does not *have* a type. This is because no expression can be its own type, as seen in the note in frame 2:77.

When an expression is a type, but does not have a type, Pie replies with just its normal form.

20 Are there any other types that don't have the type $\mathcal{U}$?

Yes. (Pair $\mathcal{U}$ $\mathcal{U}$), (Pair Atom $\mathcal{U}$), and

($\to$ $\mathcal{U}$
  $\mathcal{U}$)

are all types that do not have $\mathcal{U}$ as their type.

21 Are there any other aspects of Pie that would be good to know?

This is enough for now. There's time for more Pie later.

22 What's the next step?

Have fun playing.

23 Sounds like a plan!

Eat your vegetables
and enjoy your Pie.

3

Eliminate All Natural Numbers!

Here is the dashed definition of ***gauss*** from frame 2:59.

```
(define gauss
  (λ (n)
    (which-Nat n
      0
      (λ (n-1)
        (+ (add1 n-1) (gauss n-1))))))
```

Now, it is time to define ***gauss*** properly, without explicit recursion.

[1] Does that mean that we are about to define ***gauss*** like this?

```
(define gauss
  (λ (n)
    ...without gauss here? ))
```

Why are recursive definitions not an option?

[2] Because **they are not an option**.

Exactly.

But some recursive definitions always yield a value.

[3] Like ***gauss***, right?

That's right.

What is the normal form of (***gauss*** 0)?

[4] It is zero.

What is the value of (***gauss*** 1)?

[5] It is 1 because

1. (***gauss*** (add1 zero)) is the same as
2. (**+** 1 (***gauss*** zero)) is the same as[†]
3. (add1 (***gauss*** zero))

[†]When expressions are vertically aligned with a bar to their left, assume that "is the same as" follows all but the last one. This kind of chart is called a "same as" chart.

Is that the value?

6 Is there more to do?

3. (add1 (***gauss*** zero))
4. (add1 zero)

Sameness

If a "same as" chart could show that two expressions are the same, then this fact can be used anywhere without further justification. "Same As" charts are only to help build understanding.

Actually,

(add1 (***gauss*** zero))

is *already* a value. Why?

7 Oh, because it has the constructor add1 at the top.

Exactly.

What is the normal form of (***gauss*** 1)?

8 It is (add1 zero).

Why does (***gauss*** 2) have a normal form?

9 Because (***gauss*** 2)'s normal form relies only on the normal form of (***gauss*** 1), which *has* a normal form, and the normal form of **+**.

Does **+** have a normal form?

+ does, once it's defined. Assume that **+** does, for now.

10 All right.

Why does (***gauss*** 3) have a normal form?

11 Because (***gauss*** 3)'s normal form relies only on the normal form of (***gauss*** 2), which has a normal form, and the normal form of **+**. For now, we're assuming **+** has a normal form.

Why does (***gauss*** (add1 *k*)) have a normal form for any Nat *k*?

12 Because

(***gauss*** (add1 *k*))'s normal form

relies only on

(***gauss*** *k*)'s normal form, *k*'s value, and the normal form of **+**.

k's value must either be zero or have add1 at the top. We already know that

(***gauss*** 0) has a normal form,

and we just checked that

(***gauss*** (add1 *k*)) has a normal form for any Nat *k*.

A function that assigns a value to *every* possible argument is called a *total function*.

Both **+** and ***gauss*** are total.

13 Are there any functions that aren't total?

Total Function

A function that always assigns a value to *every* possible argument is called a *total function.*

Not here. In Pie, *all functions are total.*[†]

What is an eliminator?

[†]Because all functions are total, the order in which subexpressions are evaluated does not matter. If some functions were not total, then the order of evaluation would matter because it would determine whether or not functions were applied to the arguments for which they did not have values.

14 An eliminator takes apart values built by constructors.

What does it mean to take apart a Nat?

15 Doesn't **which-Nat** take apart a Nat?

This means that **which-Nat** is an eliminator for Nat. But Nats that have add1 at the top have a smaller Nat tucked under, and **which-Nat** does not eliminate the smaller Nat.

16 Is there a way to eliminate the smaller Nat?

One way to eliminate the smaller Nat is with **iter-Nat**.

17 What is **iter-Nat**?

An **iter-Nat**-expression looks like this:

```
(iter-Nat target
  base
  step).
```

Like **which-Nat**, when *target* is zero, the value of the **iter-Nat**-expression is the value of *base*.

18 How is **iter-Nat** unlike **which-Nat**?

Unlike **which-Nat**, when *target* is (add1 *n*), the value of the **iter-Nat**-expression is the value of

```
(step
  (iter-Nat n
    base
    step)).
```

19 So each add1 in the value of *target* is replaced by a *step*, and the zero is replaced by *base*.

The Law of iter-Nat

If *target* **is a** Nat**,** *base* **is an** X**, and** *step* **is an**

(→ X
  X),

then

(**iter-Nat** *target*
  base
  step)

is an X**.**

The First Commandment of iter-Nat

If (**iter-Nat** zero
  base
  step)
is an X**, then it is the same** X **as** *base***.**

The Second Commandment of iter-Nat

If (**iter-Nat** (add1 n)
  base
  step)
is an X**, then it is the same** X **as**

(*step*
  (**iter-Nat** n
    base
    step)).

That's right.

What is the normal form of

```
(iter-Nat 5
  3
  (λ (smaller)
    (add1 smaller)))?
```

20 It is 8 because add1 applied five times successively to 3 is 8:

```
(add1
  (add1
    (add1
      (add1
        (add1 3))))).
```

Is the **iter-Nat**-expression's type the same as *base*'s type?

21 It must be, because the value of the **iter-Nat**-expression is the value of *base* when *target* is zero.

Let's use X as a name for *base*'s type.

What is *step*'s type?

22 *step* is applied to *base*, and it is also applied to an almost-answer built by *step*. So *step* must be an

$(\rightarrow X$
  $X)$.

Just as with **which-Nat** in frame 2:45, the names *target*, *base*, and *step* are convenient ways to refer to **iter-Nat**'s arguments.

What are the target, base, and step in this **iter-Nat**-expression?

```
(iter-Nat 5
  3
  (λ (k)
    (add1 k)))
```

23 The target is

5,

The base is

3,

and the step is

```
(λ (k)
  (add1 k)).
```

Thus far, we have referred to **+** as if it were completely understood, and assumed that it has a normal form, but there is no definition for **+**.

What should **+**'s type be?

24 **+** takes two Nats and returns a Nat.

```
(claim +
  (→ Nat Nat
    Nat))
```

That's right.

If recursion were an option, then this would be a proper definition.

```
(define +
  (λ (n j)
    (which-Nat n
      j
      (λ (n-1)
        (add1 (+ n-1 j))))))
```

How can **+** be defined with **iter-Nat**?

25 Defining **+** using **iter-Nat** requires a base and a step. The base is *j* because of this "same as" chart:

1. (**+** zero *j*)
2. *j*

Is there a good way to find the step?

The step is based on the wrapper box in the recursive version of **+**. It describes how to change an almost-answer, $+_{n\text{-}1}$, into an answer.

Replace the gray box (which contains the recursion) with the argument to the step as the almost-answer. Remember the white box.

26 Here goes.

```
(claim step-+
  (→ Nat
    Nat))
(define step-+
  (λ (+n-1)
    (add1 +n-1)))
```

We can't define a new name unless all the names in both the type and the definition are already defined.†

†If definitions could refer to each other, then we could not guarantee that every defined function would be a total function.

27 And **+** refers to ***step-+***, which is now defined. This definition deserves a solid box!

```
(define +
  (λ (n j)
    (iter-Nat n
      j
      step-+)))
```

Yes, **+** is now defined.

What is (**+** (add1 zero) 7)?

28 It is 8 because

```
1. (+ (add1 zero) 7)
2. (iter-Nat (add1 zero)
     7
     step-+)
3. (step-+
     (iter-Nat zero
       7
       step-+))
4. (add1
     (iter-Nat zero
       7
       step-+))
5. (add1 7),
```

which is 8.

Can **iter-Nat** be used to define ***gauss***?

29 **iter-Nat** shows a way to repeatedly eliminate the smaller Nat tucked under an add1.

Eliminating the smaller Nat . . . this sounds like the approach that ***gauss*** follows.

Close, but the step doesn't have enough information. ***gauss*** needs an eliminator that combines the expressiveness of both **which-Nat** and **iter-Nat**. This eliminator is called **rec-Nat**.

30 What is **rec-Nat**?

The step for **rec-Nat** is applied to two arguments: the smaller Nat tucked under the add1, and the recursive answer on the smaller Nat. This is the approach used in the definition of ***gauss*** in frame 2:59.

This is the **rec-Nat** pattern.†

† The **rec-Nat** pattern is also referred to as *primitive recursion*. Thank you, Rózsa Péter (1905–1977), Wilhelm Ackermann (1896–1962), Gabriel Sudan (1899–1977), and David Hilbert (1862–1943).

31 How can ***gauss*** be defined using **rec-Nat**?

In this frame, there are two definitions of ***gauss***: the dashed box from frame 2:59 and a version using **rec-Nat**.

What are the differences?

```
(define gauss
  (λ (n)
    (which-Nat n
      0
      (λ (n-1)
        (+ (add1 n-1) (gauss n-1))))))
```

```
(define gauss
  (λ (n)
    (rec-Nat n
      0
      (λ (n-1 gauss_{n-1})
        (+ (add1 n-1) gauss_{n-1})))))
```

32 There are three differences:

1. **which-Nat** is replaced by **rec-Nat**,
2. the inner λ-expression has one more variable, $gauss_{n\text{-}1}$, and
3. the recursion (***gauss*** *n-1*) is replaced by the almost-answer $gauss_{n\text{-}1}$.

The names *n-1* and $gauss_{n\text{-}1}$ are chosen to be suggestive of what they mean, but they are just variable names.

The arguments to **rec-Nat** have the same special names as **iter-Nat**: they are always called *target*, *base*, and *step*.

33 How can we determine the values of **rec-Nat**-expressions?

As with **iter-Nat**, if the target is zero, then the value of the **rec-Nat**-expression is the value of the base.

34 What about when the target has add1 at the top?

---

which-Nat applies its step to the smaller Nat tucked under the add1.

iter-Nat applies its step to an **iter-Nat**-expression with the same base and step, but with the smaller Nat tucked under add1 as the new target.

How could these be combined?

35 Here is a guess.

The step is applied to the smaller Nat. The step is, however, also applied to a **rec-Nat**-expression with the same base and step, but with that very same smaller Nat as the target.

Good guess. When **rec-Nat** is used with a non-zero Nat as the target, the target shrinks by removing an add1 each time. Once again, the base and step do not change.

What is the value of

```
(rec-Nat (add1 zero)
  0
  (λ (n-1 almost)
    (add1
      (add1 almost))))?
```

36 It is the step applied to zero and the new **rec-Nat** expression. That is,

```
((λ (n-1 almost)
   (add1
     (add1 almost)))
 zero
 (rec-Nat zero
   0
   (λ (n-1 almost)
     (add1
       (add1 almost))))).
```

37 The resulting expression in the preceding frame is not a value, but it is the same as the original one.

What is the value?

It is

```
(add1
  (add1
    (rec-Nat zero
      0
      (λ (n-1 almost)
        (add1
          (add1 almost)))))),
```

which is a value because it has add1 at the top.

---

38 What is its normal form?

It is

```
(add1
  (add1 0)).
```

The target is zero and the base is 0.

---

39 A **rec-Nat**-expression is an expression only if the target is a Nat.

What type should the base and step have?

---

40 The base must have some type. Let's call it X, again. X can be any type, but the **rec-Nat**-expression has the same type as the base—namely X.

Is that all?

---

41 No.

If the base is an X, then the step must be an

```
(→ Nat X
  X).
```

Why is this the right type for the step?

The step is applied to two arguments: the first is a Nat because it is tucked under an add1 in a target. The second argument is *almost*. *almost* is an X because *almost* is also built by **rec-Nat**.

How does this relate to the step's type in **which-Nat** and **iter-Nat**?

42 Like **which-Nat**, **rec-Nat**'s step accepts the smaller Nat tucked under the target's add1. Like **iter-Nat**, it also accepts the recursive almost-answer.

Here is a function that checks whether a Nat is zero.

```
(claim step-zerop
  (→ Nat Atom
    Atom))
(define step-zerop
  (λ (n-1 zerop_{n-1})
    'nil))

(claim zerop
  (→ Nat
    Atom))
(define zerop
  (λ (n)
    (rec-Nat n
      't
      step-zerop)))†
```

† We use 't and 'nil as two arbitrary values. This may be familiar to Lispers (Thank you, John McCarthy (1927–2011)), but *zerop* is called *zero?* in Scheme (Thanks, Gerald J. Sussman (1947–) and Guy L Steele (1954–)).

43 Why use **rec-Nat**, which is recursive, to define something that only needs to determine whether the top constructor is zero or add1? After all, **which-Nat** would have been good enough.

---

which-Nat is easy to explain, but **rec-Nat** can do anything that **which-Nat** (and **iter-Nat**) can do.

Why are the λ-variables in ***step-zerop*** called *n-1* and $zerop_{n\text{-}1}$?

44 The name *n-1* is once again chosen to suggest one less than *n* because it is one less than the target Nat, that is, the Nat expression being eliminated. The name $zerop_{n\text{-}1}$ suggests (***zerop*** *n-1*).

The step is merely a λ-expression, so any other unused variable names would work, but this style of naming variables in steps is used frequently.

Both arguments to ***step-zerop*** are unused, which is why they are dim. Thus, the definition only seems to be recursive; in fact, it is not.

45 What is the point of a λ-expression that does not use its arguments?

The step used with **rec-Nat** always takes two arguments, though it need not always use them.

What is the value of (***zerop*** 37)?

46 Let's see.

1. (***zerop*** (add1 36))
2. (**rec-Nat** (add1 36)
     't
     step-zerop)
3. (***step-zerop*** 36
     (**rec-Nat** 36
       't
       step-zerop))
4. **'**nil

The value is determined immediately. The value for 36, which is (add1 35), is not necessary, so there's no reason to find it.

We need not evaluate expressions until their values actually become necessary. Otherwise, it would take a lot of work to evaluate the argument to ***step-zerop***

```
(rec-Nat 36
  't
  step-zerop),
```

so the "same as" chart would have at least 105 more lines.

47 Sometimes laziness is a virtue.

Is (***zerop*** 37) the same as (***zerop*** 23)?

48 Yes indeed.

```
1. 'nil
2. (step-zerop 22
     (rec-Nat 22
       't
       step-zerop))
3. (rec-Nat (add1 22)
     't
     step-zerop)
4. (zerop (add1 22))
```

Here is the step for ***gauss***.

```
(claim step-gauss
  (→ Nat Nat
    Nat))
(define step-gauss
  (λ (n-1 gauss_n-1)
    (+ (add1 n-1) gauss_n-1)))
```

49 This definition uses the naming convention from frame 44.

Yes, it does.

Another advantage of defining a step is that its type is written explicitly, rather than implied by its use in **rec-Nat**.

50 The explicit type does make it easier to read and understand the definition.

λ-variables in a step like $\mathit{zerop}_{n\text{-}1}$ and $\mathit{gauss}_{n\text{-}1}$ are *almost* the answer, in the sense of frame 2:56.

51 Okay.

What is the solid-box definition of ***gauss***?

Here it is.

```
(define gauss
  (λ (n)
    (rec-Nat n
      0
      step-gauss)))
```

What is the base?

52 The base is the second argument to **rec-Nat**. In this case, it is 0, which is a Nat.

What is the step?

53 It is ***step-gauss***.

Indeed it is.

54 What is (***gauss*** zero) using this definition?

It is 0 because

```
(rec-Nat zero
  0
  step-gauss)
```

is the same as the second argument to **rec-Nat**, which is 0.

Here is a start for finding the value of (***gauss*** (add1 zero)).

```
1. (gauss (add1 zero))
2. (step-gauss zero
     (rec-Nat zero
       0
       step-gauss))
3. (+ (add1 zero)
     (rec-Nat zero
       0
       step-gauss))
```

Now finish finding the value.

55 Here we go.

```
4. (iter-Nat (add1 zero)                    ,
     (rec-Nat zero
       0
       step-gauss)
     step-+)
5. (step-+
     (iter-Nat zero
       (rec-Nat zero
         0
         step-gauss)
       step-+))
6. (add1
     (iter-Nat zero
       (rec-Nat zero
         0
         step-gauss)
       step-+)),
```

which is a value because it has add1 at the top.

Is that value normal?

56 No, but this chart finds its normal form.

```
7. | (add1
   |   (rec-Nat zero
   |     0
   |     step-gauss))
8. | (add1 0),
```

which is indeed normal.

Why is **rec-Nat** always safe to use?

57 That's a good question.

When the target has add1 at its top, then **rec-Nat** is recursive. If recursion is not an option, why is this acceptable?

If the step does not rely on the almost-answer, as in frame 43, then a value has already been reached. If the step does rely on the almost-answer, then the recursion is guaranteed to reach the base, which is always a value or an expression that becomes a value.

58 How do we know that?

Because every target Nat is the same as either zero or (add1 n), where n is a smaller Nat.

59 How do we know that n is smaller?

The only way that it could be the same or larger is if the target Nat were built from infinitely many add1s. But because every function is total, there is no way to do this. Likewise, no step can fail to be total, because here *all* functions are total, and each step applies a function.

60 So why can't we use this style of reasoning for any recursive definition?

---

61 This style of reasoning cannot be expressed with our tools. But once we are convinced that **rec-Nat** with a total step is a way to eliminate any target Nat, we no longer need to reason carefully that each new definition is total.†

†Loosely speaking: we can't, but even if we were able to, it would be exhausting.

Are there more interesting examples of definitions using **rec-Nat**?

---

62 It can be used to define $*$† to mean multiplication.

In other words, if *n* and *j* are Nats, then

($*$ *n* *j*)

should be the product of *n* and *j*.

†$*$ is pronounced "times."

$*$ takes two Nats and their product is a Nat. So here is $*$'s type.

```
(claim *
  (→ Nat Nat
    Nat))
```

---

63 At each step, **+** adds one to the answer so far. What does $*$ do at each step?

$*$ adds *j*, its second argument, to the almost-answer.

---

64 Here is *make-step-*$*$, which yields a step function for any *j*.

```
(claim make-step-*
  (→ Nat
    (→ Nat Nat
      Nat)))
(define make-step-*
  (λ (j)
    (λ (n-1 *n-1)
      (+ j *n-1))))
```

That doesn't look like the preceding steps.

No matter what *j* is, ***make-step-∗*** constructs an appropriate *step*. This step takes two arguments because steps used with **rec-Nat** take two arguments, as in ***step-zerop*** from frame 46.

Now define ∗.

65 Okay.

The argument to ***make-step-∗*** is *j*, which is added to the product at each step. The base is 0 because multiplying by zero is 0.

```
(define ∗
  (λ (n j)
    (rec-Nat n
      0
      (make-step-∗ j))))
```

It may look as though ***make-step-∗*** is doing something new. It is a λ-expression that produces a new λ-expression. Instead of this two-step process, it is possible to collapse the nested λs into a single λ.

```
(claim step-∗
  (→ Nat Nat Nat
    Nat))
(define step-∗
  (λ (j n-1 ∗ₙ₋₁)
    (+ j ∗ₙ₋₁)))
```

make-step-∗ produces a step for any given *j*. And, despite their seeming difference, ***make-step-∗*** and ***step-∗*** have the *same definition*.

66 That can't be the same definition. It has a three-argument λ-expression.

67 Does that mean that

```
(→ Nat Nat Nat
  Nat)
```

is also a shorter way of writing something?

In fact, all λ-expressions expect exactly one argument.

```
(λ (x y z)
  (+ x (+ y z)))
```

is merely a shorter way of writing

```
(λ (x)
  (λ (y)
    (λ (z)
      (+ x (+ y z))))).
```

---

68 If a function takes three arguments, it is possible to apply the function to just one of them.

Is it also possible to apply the function to just two arguments?

It is a shorter way of writing

```
(→ Nat
  (→ Nat
    (→ Nat
      Nat))).
```

---

69 Does this mean that every function takes exactly one argument?

If f is an

```
(→ Nat Nat Nat
  Nat)
```

then

$(f\ x\ y\ z)$

is merely a shorter way of writing

$((f\ x\ y)\ z)$,

which is a shorter way of writing

$(((f\ x)\ y)\ z)$.

Indeed. Every function takes exactly one argument.

Defining functions that take multiple arguments as nested one-argument functions is called *Currying.*†

†Thank you, Haskell B. Curry (1900–1982) and Moses Ilyich Schönfinkel (1889–1942).

70 Now the definition of ∗ deserves a box.

```
(define ∗
  (λ (n j)
    (rec-Nat n
      0
      (step-∗ j))))
```

Even though ***step-∗*** looks like a three-argument λ-expression, it can be given just one argument. **rec-Nat** expects that its *step* is a function that would get exactly two arguments.

Here are the first five lines in the chart for the normal form of (∗ 2 29).

```
1. (∗ 2 29)
2. ((λ (n j)
      (rec-Nat n
        0
        (step-∗ j)))
    2 29)
3. (rec-Nat (add1
              (add1 zero))
     0
     (step-∗ 29))
4. ((step-∗ 29)
    (add1 zero)
    (rec-Nat (add1 zero)
      0
      (step-∗ 29)))
5. ((λ (n-1 ∗n-1)
      (+ 29 ∗n-1))
    (add1 zero)
    (rec-Nat (add1 zero)
      0
      (step-∗ 29)))
```

Now, find its normal form.

71 Ah, Currying is involved.

```
6.  (+ 29
       (rec-Nat (add1 zero)
         0
         (step-∗ 29)))
7.  (+ 29
       ((step-∗ 29)
        zero
        (rec-Nat zero
          0
          (step-∗ 29))))
8.  (+ 29
       (+ 29
          (rec-Nat zero
            0
            (step-∗ 29))))
9.  (+ 29
       (+ 29 0))
10. 58
```

Are any steps left out of this chart?

The Law of rec-Nat

If *target* **is a** Nat**,** *base* **is an** X**, and** *step* **is an**

```
(→ Nat X
  X)
```

then

```
(rec-Nat target
  base
  step)
```

is an X**.**

The First Commandment of rec-Nat

If (**rec-Nat** zero
base
step)
is an X**, then it is the same** X **as** *base***.**

The Second Commandment of rec-Nat

If (**rec-Nat** (add1 *n*)
base
step)
is an X**, then it is the same** X **as**

```
(step n
  (rec-Nat n
    base
    step)).
```

Yes, making (**+** 29 0) and the resultant (**+** 29 29) normal.[†]

[†]This chart saves paper, energy, and time.

72 Thanks.

At first, this chart seemed like it would be tedious.

That's just right.

What is a good name for this definition?

```
(claim step-☐
  (→ Nat Nat
    Nat))
(define step-☐
  (λ (n-1 almost)
    (∗ (add1 n-1) almost)))

(claim ☐
  (→ Nat
    Nat))
(define ☐
  (λ (n)
    (rec-Nat n
      0
      step-☐)))
```

73 This function always returns 0.

Very observant.

A shortcoming of types like Nat is that they don't say anything about *which* Nat was intended. Later, we encounter more powerful types that allow us to talk about *particular* Nats.[†]

[†]Actually, the definition in frame 73 was supposed to be *factorial*. The oversight, however, survived unnoticed in more drafts than the authors would like to admit. We leave the task of correcting it to the reader.

74 So these powerful types prevent defining ***five*** to be 9 as in frame 2:36?

Absolutely not.

Types do not prevent foolishness like defining ***five*** to be 9. We can, however, write *some* of our thoughts as types.

75 Interesting.

Go eat (+ 2 2) bananas, and rest up.

4

Easy as Pie

In frame 2:70, we defined ***Pear*** as

```
(claim Pear
  𝒰)
(define Pear
  (Pair Nat Nat))
```

Pear's eliminator was defined using **car** and **cdr**.

1 And . . .

What must an eliminator for ***Pear*** do?

2 An eliminator must expose (or unpack) information in a ***Pear***.

What about Pair's eliminator? What must it do?

3 An eliminator for Pair must expose information in a Pair.

That's close.

As seen in frame 1:22, Pair alone is not an expression, however

(Pair Nat Nat)

is an expression and it has an eliminator.

(Pair Nat Atom)

also has an eliminator.

4 Here's another try: an eliminator for

(Pair Nat Nat)

must expose information in a particular

(Pair Nat Nat),

and an eliminator for

(Pair Nat Atom)

must expose information in a particular

(Pair Nat Atom).

But this would imply that there are lots of eliminators for Pair, because it is always possible to nest them more deeply, as in frame 2:36.

5 That sounds like lots of names to remember.

It would be!

As it turns out, there is a better way. It is possible to provide an eliminator for (Pair A D), *no matter what A and D are.*

6 No matter what? Even if A were 'apple-pie?

Okay, not absolutely anything.

Based on frame 1:54, (Pair A D) is not a type unless A and D are *types.* That is, A must be a type and D must be a type.

7 Whew! What does that eliminator look like?

Here's an example.

```
(claim kar
  (→ (Pair Nat Nat)
    Nat))
```

```
(define kar
  (λ (p)
    (elim-Pair
      Nat Nat
      Nat
      p
      (λ (a d)
        a))))
```

Because ***elim-Pair*** has not yet been defined, the definition of ***kar*** is in a dashed box, however, nothing else is the matter with it.

8 Why does ***elim-Pair*** have so many arguments?

In this definition, ***elim-Pair*** has the type Nat as its first three arguments. The first two specify the types of the **car** and the **cdr** of the Pair to be eliminated.[†] The third Nat specifies that the inner λ-expression results in a Nat.

[†] Thus, the types of the arguments *a* and *d* in the inner λ-expression are also Nat.

9 What does the inner λ-expression mean?

The inner λ-expression describes how to use the information in *p*'s value. That information is the **car** and the **cdr** of *p*.

10 Why is *d* dim?

The argument name *d* is dim because it is declared in the inner λ-expression, but it is not used, just as in frame 2:47.

Now define a similar function ***kdr*** that finds the **cdr** of a pair of Nats.

11 It's nearly the same as ***kar***.

```
(claim kdr
  (→ (Pair Nat Nat)
    Nat))
```

```
(define kdr
  (λ (p)
    (elim-Pair
      Nat Nat
      Nat
      p
      (λ (a d)
        d))))
```

This time, *a* is dim because it is not used in the inner λ-expression, while *d* is dark because it is used. Because ***elim-Pair*** is not yet defined, ***kdr*** is in a dashed box, just like ***kar***.

That's right.

Write a definition called ***swap*** that swaps the **car** and **cdr** of a (Pair Nat Atom).

12 Here is ***swap***'s type.

```
(claim swap
  (→ (Pair Nat Atom)
    (Pair Atom Nat)))
```

Now define ***swap***.

13 And here is ***swap***'s definition. Once again, it is in a dashed box, like ***kar*** and ***kdr***.

```
(define swap
  (λ (p)
    (elim-Pair
      Nat Atom
      (Pair Atom Nat)
      p
      (λ (a d)
        (cons d a)))))
```

In general, ***elim-Pair*** is used like this:

```
(elim-Pair
  A D
  X
  p
  f),
```

where p is a (Pair A D) and f determines the value of the expression from the **car** and the **cdr** of p. This value must have type X.

What is ***elim-Pair***'s type?

14 Here is a guess. It could be

```
(→ A D
    X
    (Pair A D)
    (→ A D
      X)
  X)
```

because *A*, *D*, and *X* are the first three arguments, the fourth argument is a (Pair *A* *D*), and the fifth argument is a maker for *X* based on an *A* and a *D*.

But what are *A*, *D*, and *X* in that expression?

15 Are *A*, *D*, and *X* the first three arguments to ***elim-Pair***?

Do they refer to types that are already defined?

16 No. They refer to *whatever the arguments are.*

Names that occur in an expression must refer to either a definition or to an argument named by a λ. There is clearly no λ in that expression, and neither A nor D nor X are defined.

17 This must mean that the expression in frame 14 is not, in fact, a type.

Indeed.

The thought process makes sense, however. Recall what it means to be an

```
(→ Y
  X).
```

18 An

```
(→ Y
  X)
```

is a λ-expression that, when given a Y, results in an X. It can also be an expression whose value is such a λ-expression, right?

Are Y and X types?

19 They must be. Otherwise,

```
(→ Y
  X)
```

would not be a type.

In the proposed type for ***elim-Pair***, are A, D, and X type constructors?

20 No, they are not the same kind of expression as Nat and Atom, because they can be different each time ***elim-Pair*** is applied, but Nat is always Nat.

In the proposed type for ***elim-Pair***, are A, D, and X names that are defined to mean types?

21 No, because again, they can be different each time ***elim-Pair*** is applied, but once a name is **define**d, it always means the same thing.

The eliminator must be able to talk about *any* types A, D, and X.

22 It sounds like $\rightarrow$ can't do the job.

It can't, but Π[†] can.

† Π is pronounced "pie," and it can optionally be written `Pi`.

23 What does Π mean?

Here's an example.

```
(claim flip
  (Π ((A 𝒰)
      (D 𝒰))
    (→ (Pair A D)
      (Pair D A))))
(define flip
  (λ (A D)
    (λ (p)
      (cons (cdr p) (car p)))))
```

24 Does that mean that a λ-expression's type can be a Π-expression?

Good question.

It can.

25 If both Π and $\rightarrow$ can describe λ-expressions, how do they differ?

What is the value of (***flip*** Nat Atom)?

26 It must be the λ-expression

```
(λ (p)
  (cons (cdr p) (car p)))
```

because ***flip*** is defined to be a λ-expression and it is applied to two arguments, Nat and Atom.

What is the value of

((***flip*** Nat Atom) (cons 17 'apple))?

27 It is

(cons 'apple 17),

which is a

(Pair Atom Nat).

The difference between Π and → is in the type of an expression in which a function is applied to arguments.

(***flip*** Nat Atom)'s type is

```
(→ (Pair Nat Atom)
  (Pair Atom Nat)).
```

This is because when an expression described by a Π-expression is applied, the argument expressions replace the *argument names* in the *body* of the Π-expression.

28 How does the body of a Π-expression relate to the body of a λ-expression?

Both Π-expressions and λ-expressions introduce argument names, and the body is where those names can be used.

29 What are *argument names*?

In this Π-expression,

```
(Π ((A 𝓤)
    (D 𝓤))
  (→ (Pair A D)
    (Pair D A))),
```

the argument names are *A* and *D*. Π-expressions can have one or more argument names, and these argument names can occur in the body of the Π-expressions.

30 What is the *body* of a Π-expression?

In this Π-expression,

```
(Π ((A 𝒰)
    (D 𝒰))
  (→ (Pair A D)
     (Pair D A))),
```

the body is

```
(→ (Pair A D)
   (Pair D A)).
```

It is the type of the body of the λ-expression that is described by the body of the Π-expression.

[31] What do the A and the D refer to in the Π-expression's body?

The Intermediate Law of Application

If f is a

```
(Π ((Y 𝒰))
   X)
```

and Z is a $\mathcal{U}$, then

```
(f Z)
```

is an X

where every Y has been consistently replaced by Z.

The A and the D in the body refer to specific types that are not yet known. No matter which two types A and D are arguments to the λ-expression that is described by the Π-expression, the result of applying that λ-expression is always an

```
(→ (Pair A D)
   (Pair D A)).
```

[32] Does that mean that the type of

```
(flip Atom (Pair Nat Nat))
```

is

```
(→ (Pair Atom
         (Pair Nat Nat))
   (Pair (Pair Nat Nat)
         Atom))?
```

That's right.

Why is that the case?

33 The variables A and D are replaced with their respective arguments: Atom and (Pair Nat Nat).

Are

```
(Π ((A 𝒰)
    (D 𝒰))
  (→ (Pair A D)
    (Pair D A)))
```

and

```
(Π ((Lemon 𝒰)
    (Meringue 𝒰))
  (→ (Pair Lemon Meringue)
    (Pair Meringue Lemon)))
```

the same type?

34 Yes, because consistently renaming variables as in frame 2:21 does not change the meaning of anything.

Are

```
(Π ((A 𝒰)
    (D 𝒰))
  (→ (Pair A D)
    (Pair D A)))
```

and

```
(Π ((A 𝒰)
    (D 𝒰))
  (→ (Pair
        (car
          (cons A D))
        (cdr
          (cons A D)))
    (Pair D A)))
```

the same type?

35 Yes, because

```
(car
  (cons A D))
```

and A are the same type, and

```
(cdr
  (cons A D))
```

and D are the same type.

Could we have defined ***flip*** this way?

```
(claim flip
  (Π ((A U)
      (D U))
    (→ (Pair A D)
      (Pair D A))))
(define flip
  (λ (C A)
    (λ (p)
      (cons (cdr p) (car p)))))
```

36 Here's a guess.

In this definition, the names in the outer λ-expression are different from the names in the Π-expression. That seems like it should not work. *A* is in the wrong place, and *C* is neither *A* nor *D*.

The proposed definition of ***flip*** in frame 36 is allowed. Like defining ***five*** to mean 9, however, it is foolish.

37 Why is it allowed?

The names in the outer λ need not match the names in the Π-expression. The *C* in the outer λ-expression matches the *A* in the Π-expression because they are both the first names. The *A* in the outer λ-expression matches the *D* in the Π-expression because they are both the second names. What matters is the *order* in which the arguments are named.†

What does *p* in the inner λ-expression match?

†Even though it is not wrong to use names that do not match, it is confusing. We always use matching names.

38 The *p* matches the (Pair *A* *D*) after the →, which gives the inner λ-expression's argument type.

How can the *C* and the *A* in the definition in frame 36 be consistently renamed to improve the definition?

39 First, the *A* should be renamed to *D*. Then, the *C* can be renamed to *A*.

Isn't this the definition in frame 24?

Is it now possible to define a single eliminator for Pair?

40 Yes. Shouldn't the type be

```
(Π ((A 𝒰)
    (D 𝒰)
    (X 𝒰))
  (→ (Pair A D)
      (→ A D
        X)
    X))?
```

It looks a lot like the type in frame 14.

That's right.

What is the definition of ***elim-Pair***?

41 How about this?

```
(claim elim-Pair
  (Π ((A 𝒰)
      (D 𝒰)
      (X 𝒰))
    (→ (Pair A D)
        (→ A D
          X)
      X)))
(define elim-Pair
  (λ (A D X)
    (λ (p f)
      (f (car p) (cdr p)))))
```

Now ***kar*** deserves a solid box.

```
(define kar
  (λ (p)
    (elim-Pair
      Nat Nat
      Nat
      p
      (λ (a d)
        a))))
```

42 And so does ***kdr***.

```
(define kdr
  (λ (p)
    (elim-Pair
      Nat Nat
      Nat
      p
      (λ (a d)
        d))))
```

So does ***swap***.

43 Right.

```
(define swap
  (λ (p)
    (elim-Pair
      Nat Atom
      (Pair Atom Nat)
      p
      (λ (a d)
        (cons d a)))))
```

Even though a Π-expression can have any number of argument names, it is simplest to first describe when a one-argument Π-expression is a type.

To be a

```
(Π ((Y 𝒰))
  X)
```

is to be a λ-expression that, when applied to a type T, results in an expression with the type that is the result of consistently replacing every Y in X with T.

44 Forgetting something?

It can also be an expression *whose value is* such a λ-expression.

45 It is important not to forget evaluation.

Is this a complete description of Π-expressions?

No, not yet.

Based on one-argument Π-expressions, what does it mean to be a

```
(Π ((Y 𝒰) (Z 𝒰))
  X)?
```

46 It must mean to be a λ-expression or an expression that evaluates to a λ-expression that, when applied to two types T and S, results in an expression whose type is found by consistently replacing every Y in X with T and every Z in the new X with S.

Π-expressions can have any number of arguments, and they describe λ-expressions that have the same number of arguments.

What expressions have the type

```
(Π ((A 𝒰))
  (→ A
    (Pair A A)))?
```

47 How about this one?

```
(λ (A)
  (λ (a)
    (cons a a)))?
```

Here is a name for a familiar expression.[†]

```
(claim twin-Nat
  (→ Nat
    (Pair Nat Nat)))
(define twin-Nat
  (λ (x)
    (cons x x)))
```

What is the value of

```
(twin-Nat 5)?
```

[†]It is familiar from frame 2:19.

48 It is

```
(cons 5 5).
```

Here is a very similar definition.

```
(claim twin-Atom
  (→ Atom
    (Pair Atom Atom)))
(define twin-Atom
  (λ (x)
    (cons x x)))
```

What is the value of

```
(twin-Atom 'cherry-pie)?
```

49 It is

```
(cons 'cherry-pie 'cherry-pie).
```

What is the matter with these definitions? Why don't they deserve solid boxes?

There is nothing specific to Nat or Atom about

```
(λ (a)
  (cons a a)).
```

Instead of writing a new definition for each type, Π can be used to build a general-purpose ***twin*** that works for *any* type.

50 Here is the general-purpose ***twin***.

```
(claim twin
  (Π ((Y U))
    (→ Y
      (Pair Y Y))))
(define twin
  (λ (Y)
    (λ (x)
      (cons x x))))
```

What is the value of (***twin*** Atom)?

51 (***twin*** Atom) is

```
(λ (x)
  (cons x x)).
```

What is (***twin*** Atom)'s type?

52 Consistently replacing every *Y* in

```
(→ Y
  (Pair Y Y))
```

with Atom results in

```
(→ Atom
  (Pair Atom Atom)).
```

What is the relationship between ***twin-Atom***'s type and (***twin*** Atom)'s type?

53 ***twin-Atom***'s type and (***twin*** Atom)'s type are the same type.

Next, define ***twin-Atom*** using the general-purpose ***twin***.

```
(claim twin-Atom
  (→ Atom
    (Pair Atom Atom)))
```

54 It can be done using the technique from frame 27.

```
(define twin-Atom
  (twin Atom))
```

55

Is

(***twin-Atom*** 'cherry-pie)

the same

(Pair Atom Atom)

as

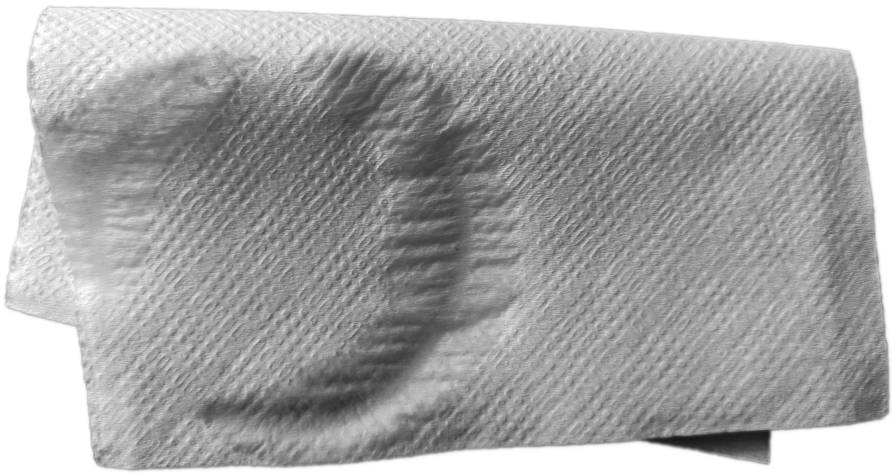
((***twin*** Atom) 'cherry-pie)?

Yes, and its value, but also its normal form is

(cons 'cherry-pie 'cherry-pie).

There's twice as much for dessert!

Now go to your favorite confectionary shop and share a delicious cherry Π.

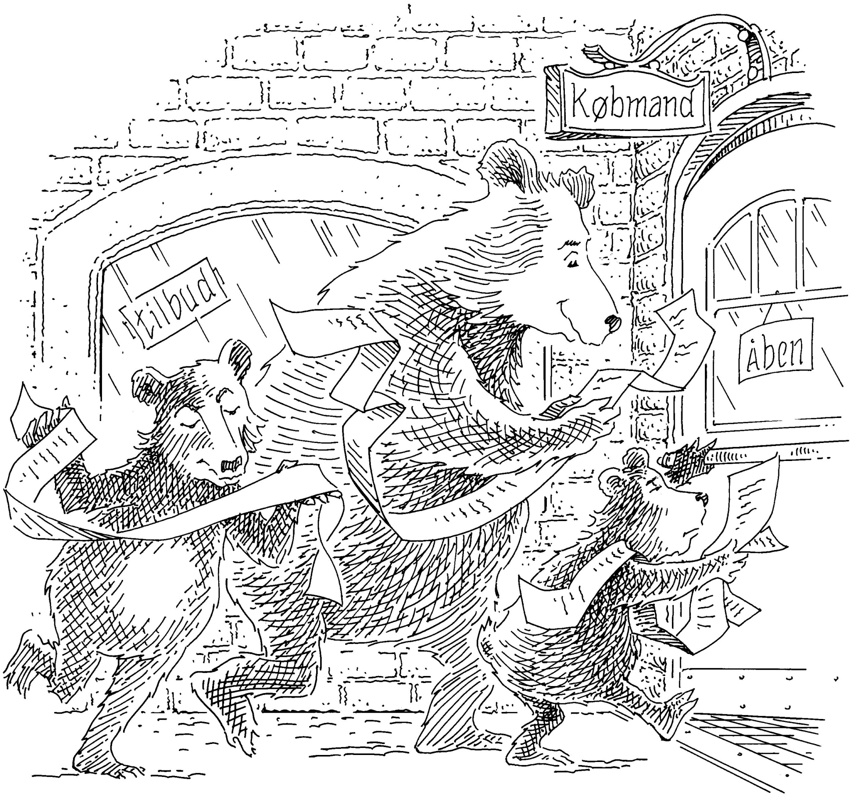


Ceci n'est pas une serviette! [†]

[†] Thank you, René François Ghislain Magritte (1898–1967).

5

Lists, Lists, and More Lists



How was that Π?

1 Delicious. A napkin would have made eating less messy, though.

Before we begin, have you

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

2 That's quite the list of expectations.

Yes, but they're great expectations.

```
(claim expectations
  (List Atom))
(define expectations
  (:: 'cooked
    (:: 'eaten
      (:: 'tried-cleaning
        (:: 'understood
          (:: 'slept nil))))))
```

3 This is confusing in these ways:

- :: has not yet been described,
- the type constructor List has not been described, and
- the atom 'nil has been used as part of ***step-zerop***.

Is 'nil the same as nil in frame 3?

4 No, it isn't, because the nil in frame 3 is not an Atom—it does not begin with a tick mark.

Is nil an expression?

List is a type constructor. If E is a type, then (List E)[†] is a type.

[†] Pronounced "list of entries of type E," or simply "list of E."

5 What does it mean to be a (List E), then?

The Law of List

If E is a type, then (List E) is a type.

Is nil a (List Atom)?

6 nil looks like it plays the role of the empty list in frame 3.

Yes, nil is a (List Atom).

Is nil a (List Nat)?

7 Not likely, because nil is a (List Atom).

Actually, nil is a (List Nat) as well.

Is nil a (List (List Atom))?

8 Yes, because (List Atom) is a type, so (List (List Atom)) is also a type. What about (List (Pair Nat Atom))?

Is nil one of those, too?

Yes, it is.

9 Does that mean that nil is a

(List 'potato)

as well?

No, it is not, because 'potato is not a type.

10 Is it for the same reason that

(Pair 'olive 'oil)

in frame 1:52 is not a type?

Yes.

(List 'potato) is not a type, because 'potato is an Atom, not a type.

11 Okay. This means that if E is a type, then (List E) is a type, right?

And if (List E) is a type, then nil is a (List E).

12 All right.

Is nil a constructor?

Yes, nil is a constructor.

Guess the other constructor of (List E).

13 Based on ***expectations***, :: is the other constructor.

How does ::† differ from cons?

The constructor :: builds a List ...

†For historical reasons, :: is also pronounced "cons" or "list-cons."

14 ... but the constructor cons builds a Pair.

It is possible to have a list of pairs, or a pair of lists.

When is (:: e es†) a (List E)?

†The plural of e is es and is pronounced *ease*. es is used because the rest of a list could have any number of entries.

15 Well, es must be a (List E). es could be nil, and nil is a (List E).

Can e be anything at all?

16 Of course!

Of course not! Try again.

17 Here's a guess: e must be an E because E has not yet been used for anything else.

Right answer; wrong reason.

e must be an E because in order to use an eliminator for (List E), we must know that everything in the list is an E.

Define ***rugbrød***[†] to be the ingredients of Danish rye bread.

[†]Pronounced [ˈʁuˌb̥ʁœðˀ]. If this is no help, ask a Dane.

18 What are the ingredients?

The ingredients in *rugbrød* are:

- whole-grain rye flour,
- rye kernels, soaked until soft,
- pure water,
- active sourdough, and
- salt.

19 What type should ***rugbrød*** have?

(List Atom), because each ingredient is an Atom.

20 Okay, here goes.

```
(claim rugbrød
  (List Atom))
(define rugbrød
  (:: 'rye-flour
    (:: 'rye-kernels
      (:: 'water
        (:: 'sourdough
          (:: 'salt nil))))))
```

Very good.

21 Yes, ***rugbrød*** is quite tasty! It does need something on top, though.

Let's get back to that.

How does ***rugbrød*** differ from 5?

22 They appear to have nothing in common. 5 is made up of add1 and zero. Also, 5 is not tasty.

How many ingredients does ***rugbrød*** contain?

23 Five.

In addition to only requiring five ingredients, ***rugbrød*** doesn't even need kneading.

24 Does :: have something to do with add1, then?

:: makes a list bigger, while add1 makes a Nat bigger.

Does nil have something to do with zero as well?

25 nil is the smallest list, while zero is the smallest Nat.

Does the eliminator for lists look like one for Nats?

The Law of nil

nil is a (List E), no matter what type E is.

The Law of ::

If e is an E and es is a (List E),
then (:: e es) is a (List E).

Yes, it does.

What type does

(**rec-Nat** *target*
  base
  step)

have?

[26] The **rec-Nat**-expression is an X when

- *target* is a Nat,
- *base* is an X, and
- *step* is an ($\rightarrow$ Nat X X).

The eliminator for (List E) is written

(**rec-List** *target*
  base
  step)

and it is an X when

- *target* is a (List E),
- *base* is an X, and
- *step* is an ($\rightarrow$ E (List E) X X).

How does this differ from **rec-Nat**?

[27] **rec-List**'s *step* takes one more argument than **rec-Nat**'s *step*—it takes e, an entry from the list.

Nicely done!

In both cases, the step accepts every argument from the corresponding constructor as well as the recursive elimination of the smaller value.

[28] Eliminators expose the information in values.

The base exposes a lot of information about the result of a **rec-List**. What are two uses of **rec-List** that have 0 as their base?

[29] One use is to find the length of a list. Another is to find the sum of all the Nats in a (List Nat).

Those are two good examples.

With this definition

```
(claim step-[    ]
  (→ Atom (List Atom) Nat
    Nat))
(define step-[    ]
  (λ (e es n)
    (add1 n)))
```

what is the value of

```
(rec-List nil
  0
  step-[    ])?
```

30 It must be 0, because 0 is the base and the value of the base must be the value for nil.

That's right.

A ***kartoffelmad*** is ***rugbrød*** with ***toppings*** and ***condiments***.

```
(claim toppings
  (List Atom))
(define toppings
  (:: 'potato
    (:: 'butter† nil)))

(claim condiments
  (List Atom))
(define condiments
  (:: 'chives
    (:: 'mayonnaise† nil)))
```

†Or your favorite non-dairy alternative.

31 That sounds *lækkert*!

The Law of rec-List

If *target* **is a** (List E), *base* **is an** X, **and** *step* **is an**

(→ E (List E) X
  X),

then

(rec-List *target*
  base
  step)

is an X**.**

The First Commandment of rec-List

If (rec-List nil
  base
  step)

is an X**, then it is the same** X **as** *base***.**

The Second Commandment of rec-List

If (rec-List (:: *e* *es*)
  base
  step)

is an X**, then it is the same** X **as**

(*step* *e* *es*
  (rec-List *es*
    base
    step)).

It is!

What is the value of

```
(rec-List condiments
  0
  step-□)?
```

32 Let's see.

```
1. (rec-List (:: 'chives
               (:: 'mayonnaise nil))
     0
     step-□)
2. (step-□
     'chives
     (:: 'mayonnaise nil)
     (rec-List (:: 'mayonnaise nil)
       0
       step-□))
3. (add1
     (rec-List (:: 'mayonnaise nil)
       0
       step-□))
```

What is the normal form? Feel free to leave out the intermediate expressions.

33 The normal form is

```
(add1
  (add1 zero)),
```

better known as 2.

The **rec-List** expression replaces each :: in ***condiments*** with an add1, and it replaces nil with 0.

What is a good name to fill in the box?

34 The name ***length*** seems about right.

```
(claim step-length
  (→ Atom (List Atom) Nat
    Nat))
(define step-length
  (λ (e es length_es)
    (add1 length_es)))
```

Then this must be ***length***.

```
(claim length
  (→ (List Atom)
    Nat))
(define length
  (λ (es)
    (rec-List es
      0
      step-length)))
```

35 But what about the length of

```
(:: 17
  (:: 24
    (:: 13 nil)))?
```

That's easy, just replace Atom with Nat.

```
(claim step-length
  (→ Nat (List Nat) Nat
    Nat))
(define step-length
  (λ (e es length_es)
    (add1 length_es)))
```

36 And here's ***length*** for a list of Nats.

```
(claim length
  (→ (List Nat)
    Nat))
(define length
  (λ (es)
    (rec-List es
      0
      step-length)))
```

Lists can contain entries of any type, not just Atom and Nat.

What can be used to make a version of ***step-length*** that works for all types?

37 It's as easy as Π.

```
(claim length
  (Π ((E 𝒰))
    (→ (List E)
      Nat)))
```

That **claim** requires a step.

```
(claim step-length
  (Π ((E 𝒰))
    (→ E (List E) Nat
      Nat)))
```

38 At each step, the length grows by add1.

```
(define step-length
  (λ (E)
    (λ (e es length_es)
      (add1 length_es))))
```

This uses the same technique as ***step-∗*** in frame 3:66 to apply ***step-length*** to *E*.

Now define ***length***.

39 Passing *E* to ***step-length*** causes it to take three arguments.

```
(define length
  (λ (E)
    (λ (es)
      (rec-List es
        0
        (step-length E)))))
```

Why is *e* in ***step-length*** dim?

40 Because the specific entries in a list aren't used when finding the length.

What is the value of (***length*** Atom)?

41 It is

```
(λ (es)
  (rec-List es
    0
    (step-length Atom))),
```

which is found by replacing each *E* with Atom in the inner λ-expression's body.

Define a specialized version of ***length*** that finds the number of entries in a (List Atom).

42 This uses the same technique as the definition of ***twin-Atom*** in frame 4:54.

```
(claim length-Atom
  (→ (List Atom)
    Nat))
(define length-Atom
  (length Atom))
```

That is a useful technique.

Now it is time to assemble a delicious *kartoffelmad* from a slice of bread, ***toppings***, and ***condiments***.

Define a function that appends two lists.

43 What should be the definition's type?

Is it possible to append a (List Nat) and a (List (Pair Nat Nat))?

44 No.

All the entries in a list must have the same type.

List Entry Types

All the entries in a list must have the same type.

As long as two lists contain the same entry type, they can be appended, no matter which entry type they contain.

What does this say about the type in ***append***'s definition?

45 The type must be a Π-expression.

```
(claim append
  (Π ((E 𝒰))
              ))
```

Exactly.

What are the rest of the arguments?

46 There must be two (List E) arguments. Also, the result is a (List E). From that, ***append*** must be a λ-expression.

Here is the claim. Now start the definition.

```
(claim append
  (Π ((E U))
    (→ (List E) (List E)
      (List E))))
```

47 It is a λ-expression, but the body remains a mystery.

```
(define append
  (λ (E)
    (λ (start end)
                    )))
```

What goes in the box?

48 Some kind of **rec-List**.

What is the value of

```
(append Atom
  nil
  (:: 'salt
    (:: 'pepper nil)))?
```

49 Clearly it must be

```
(:: 'salt
  (:: 'pepper nil)).
```

And what is the normal form of

```
(append Atom
  (:: 'cucumber
    (:: 'tomato nil))
  (:: 'rye-bread nil))?
```

50 It must be

```
(:: 'cucumber
  (:: 'tomato
    (:: 'rye-bread nil))).
```

The value of (***append*** *E* nil *end*) should be the value of *end*. Thus, ***append***'s last argument *end* is the base.

51 What about the step?

The step's type is determined by the Law of **rec-List**. It should work for any entry type.

52 How about this one?

```
(claim step-append
  (Π ((E U))
    (→ E (List E) (List E)
      (List E))))
```

Using the previous frame as an example, fill in the rest of ***step-append***.†

```
(define step-append
  (λ (E)
    (λ (e es append_es)
      [          ])))

(define append
  (λ (E)
    (λ (start end)
      (rec-List start
        end
        (step-append E)))))
```

†The expression (***step-append*** *E*) should be a step for ***append*** when the list contains entries of type *E*. Be mindful of the Currying.

53 If $append_{es}$ is

nil,

then the ***step-append*** should produce

(:: 'rye-bread nil).

If $append_{es}$ is

(:: 'rye-bread nil),

then the ***step-append*** should produce

```
(:: 'tomato
  (:: 'rye-bread nil)).
```

Finally, if $append_{es}$ is

```
(:: 'tomato
  (:: 'rye-bread nil)),
```

then the ***step-append*** should produce

```
(:: 'cucumber
  (:: 'tomato
    (:: 'rye-bread nil))).
```

That is good reasoning.

What is the proper definition?

54 Now ***append*** deserves a solid box.

```
(define step-append
  (λ (E)
    (λ (e es append_es)
      (:: e append_es))))

(define append
  (λ (E)
    (λ (start end)
      (rec-List start
        end
        (step-append E)))))
```

This definition of ***append*** is very much like **+**.

55 Is there an **iter-List**, like **iter-Nat**, and could it be used to define ***append***?

Nothing would stop us from defining **iter-List**, but there is no need, because **rec-List** can do everything that **iter-List** could do, just as **rec-Nat** can do everything that **iter-Nat** and **which-Nat** can do.

56 Okay, let's use the more expressive eliminators here.

It is also possible to define ***append*** in another way, replacing :: with something else.

57 Is that possible?

Yes, it is. Instead of using :: to "cons" entries from the first list to the front of the result, it is also possible to ***snoc***† entries from the second list to the back of the result.

For example, the value of

```
(snoc Atom toppings 'rye-bread)
```

is

```
(:: 'potato
  (:: 'butter
    (:: 'rye-bread nil))).
```

What is ***snoc***'s type?

† Thanks, David C. Dickson (1947–)

58 ***snoc***'s type is

```
(claim snoc
  (Π ((E 𝒰))
    (→ (List E) E
      (List E))))
```

What must the step do?

The step must "cons" the current entry of the list onto the result.

59 Oh, so it's just like ***step-append***.

Now define ***snoc***.

60 Here is ***snoc***.

```
(define snoc
  (λ (E)
    (λ (start e)
      (rec-List start
        (:: e nil)
        (step-append E)))))
```

Well done.

Now define ***concat***, which should behave like ***append*** but use ***snoc*** in its step.

```
(claim concat
  (Π ((E 𝒰))
    (→ (List E) (List E)
      (List E))))
```

concat's type is the same as ***append***'s type because they do the same thing.

61 In addition to using ***snoc*** instead of the List "cons" ::, ***concat*** must eliminate the second list.

```
(claim step-concat
  (Π ((E 𝒰))
    (→ E (List E) (List E)
      (List E))))
(define step-concat
  (λ (E)
    (λ (e es concat_es)
      (snoc E concat_es e))))

(define concat
  (λ (E)
    (λ (start end)
      (rec-List end
        start
        (step-concat E)))))
```

A list can be reversed using ***snoc*** as well.

What should the type of ***reverse*** be?

62 ***reverse*** accepts a single list as an argument.

```
(claim reverse
  (Π ((E 𝒰))
    (→ (List E)
      (List E))))
```

What should be done at each step?

63 At each step, *e* should be ***snoc***'d onto the back of the reversed *es*.

```
(claim step-reverse
  (Π ((E 𝒰))
    (→ E (List E) (List E)
      (List E))))
```

Now define ***step-reverse*** and ***reverse***.

64 Here they are.

```
(define step-reverse
  (λ (E)
    (λ (e es reverse_es)
      (snoc E reverse_es e))))

(define reverse
  (λ (E)
    (λ (es)
      (rec-List es
        nil†
        (step-reverse E)))))
```

†When using Pie, it is necessary to replace this nil with `(the (List E) nil)`.

Now it is time for something *lækkert*.

```
(claim kartoffelmad
  (List Atom))
(define kartoffelmad
  (append Atom
    (concat Atom
      toppings condiments)
    (reverse Atom
      (:: 'plate
        (:: 'rye-bread nil)))))
```

What is ***kartoffelmad***'s normal form?

65 It is

```
(:: 'chives
  (:: 'mayonnaise
    (:: 'potato
      (:: 'butter
        (:: 'rye-bread
          (:: 'plate nil)))))).
```

It's a good thing we asked for the normal form instead of the value. Otherwise, you'd have to assemble all but the `'chives` while eating it!

66 Reversing lists is hungry work.

Have yourself a nice *kartoffelmad*,

and get ready for more delicious Π.

RUGBRØD

Day 1

Mix about 150g sourdough, 400g dark whole rye flour, and 1L water in a bowl and mix until no flour clumps remain.

Add enough water to completely cover 500g whole or cracked rye kernels with water, and let them soak. Cover both bowls with a cloth and let them sit.

Day 2

Take some of the dough, and save it in the fridge for next time. Drain the kernels. Mix one tablespoon salt, 450g rye flour and the soaked kernels into the dough.

Pour the dough into a Pullman loaf pan (or a proper rugbrød pan if you have one) and cover with a cloth.

Day 3

Bake the bread at 180° C for 90 minutes, or for 80 minutes in a convection oven.

Wrap the baked bread in a towel and allow it to cool slowly before tasting it.

The Rest of Your Life

If not baking weekly, feed the saved sourdough every week by throwing away half and adding fresh rye flour and water.

Make your bread your own by adding sunflower seeds, flax seeds, dark malt, pumpkin seeds, or whatever else strikes your fancy.

KARTOFFELMAD

Take a thin slice of rugbrød, approximately 0.75cm. Spread it with butter. Artfully arrange slices of cooled boiled new potato on the buttered bread, and top with mayonnaise and chives.

LÆKKERT!

6
Precisely How Many?

. . .

1 After all that sandwich, some Π would go great.

We're glad you asked . . .

2 I'm pretty good at anticipating what you want me to ask.

Naturally. Let's get started.

Let's define a function ***first*** that finds the first entry in *any* List.

3 Wouldn't that be easy to do?

Actually, it would be impossible!

4 Why would it be impossible?

It is impossible because nil has no first entry . . .

5 . . . and therefore ***first*** would not be total.

What about a function, ***last*** that, instead of finding the first entry, finds the last entry in a List?

6 The function ***last*** would also not be total, because nil has no last entry.

To write a total function ***first***, we must use a more specific type constructor than List. This more specific type constructor is called Vec, which is short for "vector," but it is really just a list with a length.

An expression (Vec E k)† is a type when E is a type and k is a Nat. The Nat gives the length of the list.

Is (Vec Atom 3) a type?

†Pronounced "list of E with length k," or simply "list of E length k."

7 Can types contain expressions that aren't types?

Just as types can be the outcome of evaluating an expression (as in frame 1:55), some types contain other expressions that are not themselves types.

8 Then (Vec Atom 3) is a type because Atom is a type and 3 is clearly a Nat.

Is

```
(Vec
  (cdr
    (cons 'pie
      (List (cdr (cons Atom Nat)))))
  (+ 2 1))
```

a type?

9 It must be, because

```
(cdr
  (cons 'pie
    (List (cdr (cons Atom Nat)))))
```

and

```
(List Nat)
```

are the same type, and because

```
(+ 2 1)
```

is the same Nat as

```
3.
```

That means that the expression is the same as

```
(Vec (List Nat) 3),
```

which is clearly a type.

The only constructor of (Vec *E* zero) is vecnil.

10 Is this because the length of vecnil is zero?

Precisely.

vec:: is the only constructor of

(Vec *E* (add1 *k*)).

11 What is *k* here?

Here, *k* can be any Nat.

(vec:: *e es*) is a (Vec *E* (add1 *k*)) when *e* is an *E* and *es* is a (Vec *E* *k*).

12 If an expression is a

(Vec *E* (add1 *k*)),

then its value has at least one entry, so it is possible to define ***first*** and ***last***, right?

Right. Is

(vec:: 'oyster vecnil)

a

(Vec Atom 1)?

13 Yes, because

'oyster

is an

Atom

and

vecnil

is a

(Vec Atom zero).

The Law of Vec

If E is a type and k is a Nat, then (Vec E k) is a type.

The Law of vecnil

vecnil **is a (Vec E zero).**

The Law of vec::

If e is an E and es is a (Vec E k), then (vec:: e es) is a (Vec E (add1 k)).

Is
```
(vec:: 'crimini
  (vec:: 'shiitake vecnil))
```
a
```
(Vec Atom 3)?
```

14 No, because it is not a list of precisely three atoms.

How does this relate to frame 11?

15 It is not a
```
(Vec Atom 3)
```
because
```
(vec:: 'shiitake vecnil)
```
is not a
```
(Vec Atom 2).
```

Why is
```
(vec:: 'shiitake vecnil)
```
not a
```
(Vec Atom 2)?
```

16 If it were, then
```
vecnil
```
would have to be a
```
(Vec Atom 1),
```
based on the description in frame 11.

Why can't that be the case?

17 Because
```
vecnil
```
is a
```
(Vec Atom zero),
```
and 1 is not the same Nat as zero.

Why is 1 not the same Nat as zero?

18 Frame 1:100 explains that two Nats are the same when their values are the same, and that their values are the same when either both are zero or both have add1 at the top.

It is now possible to define ***first-of-one***, which gets the first entry of a (Vec E 1).

19 But is it possible? So far, there are no eliminators for Vec.

Good point. Two of the eliminators for Vec are **head** and **tail**.

20 What do **head** and **tail** mean?

(**head** es) is an
  E
when
  es
is a
  (Vec E (add1 k)).

What form can the value of es take?

21 It cannot be vecnil because vecnil has zero entries. So es has vec:: at the top.

The expression
  (**head**
    (vec:: a d))
is the same E as a.

22 What about **tail**?

  (**tail** es)
is a
  (Vec E k)
when
  es
is a
  (Vec E (add1 k)).

23 es has vec:: at the top.

Is
  (**tail**
    (vec:: a d))
the same
  E
as
  d?

No, but

    (tail
      (vec:: a d))

is the same

    (Vec E k)

as

    d.

Now define ***first-of-one***.

24 ***first-of-one*** uses **head** to find the only entry.

```
(claim first-of-one
  (Π ((E U))
    (→ (Vec E 1)
      E)))
(define first-of-one
  (λ (E)
    (λ (es)
      (head es))))
```

What is the value of

    (first-of-one Atom
      (vec:: 'shiitake vecnil))?

25 It is 'shiitake.

What is the value of

    (first-of-one Atom vecnil)?

26 That question is meaningless because

    (first-of-one Atom vecnil)

is not described by a type, and this is because

    vecnil

is not a

    (Vec Atom 1).

That's right, the question is meaningless.

Now define ***first-of-two***.

27 It is very much like ***first-of-one***.

```
(claim first-of-two
  (Π ((E U))
    (→ (Vec E 2)
      E)))
(define first-of-two
  (λ (E)
    (λ (es)
      (head es))))
```

What is the value of

```
(first-of-two Atom
  (vec:: 'matsutake
    (vec:: 'morel
      (vec:: 'truffle vecnil))))?
```

28 That is quite a valuable list of mushrooms.

The question, however, doesn't make sense because that valuable list of mushrooms has three, instead of precisely two, mushrooms.

Good point.

It is now time for ***first-of-three***.

29 Is there a way to define a ***first*** that works for any length?

No, there is not, because there is no ***first*** entry when the length is zero. But it is possible to define a ***first*** that finds the first entry in any list that has *at least one* entry.

30 That sounds difficult.

Actually, it's not that difficult.

In fact, it's as easy as . . .

31 . . . as Π?

Π-expressions are more flexible than we have seen thus far.

32 What is a more flexible kind of Π?

A mushroom pot pie, for one.

33 What is a more flexible kind of Π-expression?

Here is ***first***'s claim.

```
(claim first
  (Π ((E 𝒰)
      (ℓ Nat))
    (→ (Vec E (add1 ℓ))
      E)))
```

What is new here?

34 After the argument name ℓ, it says Nat. Earlier, it always said 𝒰 after argument names in Π-expressions.

The *E* in

```
(→ (Vec E (add1 ℓ))
  E)
```

refers to whatever 𝒰 is the first argument to ***first***. Does this mean that the ℓ in (add1 ℓ) refers to whatever Nat is the second argument to ***first***?

The Law of Π

The expression

```
(Π ((y Y))
  X)
```

is a type when Y is a type, and X is a type if y is a Y.

Precisely. The (add1 ℓ) ensures that the list that is the third argument to ***first*** has at least one entry.

Now define ***first***.

35 Here it is, in a well-deserved solid box.

```
(define first
  (λ (E ℓ)
    (λ (es)
      (head es))))
```

What is the value of

```
(first Atom 3
  (vec:: 'chicken-of-the-woods
    (vec:: 'chantrelle
      (vec:: 'lions-mane
        (vec:: 'puffball vecnil)))))?
```

36 It is 'chicken-of-the-woods.

But why is the number of entries

(add1 ℓ)

instead of just

ℓ?

There is no first entry to be found in vecnil, which has zero entries.

No matter what ℓ is, (add1 ℓ) can never be the same Nat as zero, so vecnil is not a (Vec *E* (add1 ℓ)).

[37] We avoid attempting to define a non-total function by using a more specific type to rule out unwanted arguments.

Use a More Specific Type

Make a function total by using a more specific type to rule out unwanted arguments.

The same definition could have been written with two nested Π-expressions.

```
(claim first
  (Π ((E 𝒰))
    (Π ((ℓ Nat))
      (→ (Vec E (add1 ℓ))
        E))))
(define first
  (λ (E)
    (λ (ℓ)
      (λ (es)
        (head es)))))
```

Why would this be the same definition?

[38] This would have been the same definition because Π-expressions with many argument names are shorter ways of writing nested Π-expressions with one argument name each.

This definition could also have been written with three nested Π-expressions.

```
(claim first
  (Π ((E 𝒰))
    (Π ((ℓ Nat))
      (Π ((es (Vec E (add1 ℓ))))
        E))))
(define first
  (λ (E)
    (λ (ℓ)
      (λ (es)
        (head es)))))
```

Why would *this* have been the same definition?

39 Would it really have been the same definition?

The previous definition had an →, while this definition does not.

In fact, →-expressions are a shorter way of writing Π-expressions when the argument name is not used in the Π-expression's body.

40 Ah, okay.

→ and Π

The type

$(\rightarrow\ Y$
$\quad X)$

is a shorter way of writing

$(\Pi\ ((y\ \ Y))$
$\quad X)$

when y **is not used in** X**.**

The Final Law of λ

If x is an X when y is a Y, then

```
(λ (y)
  x)
```

is a

```
(Π ((y Y))
  X).
```

The Final Law of Application

If f is a

```
(Π ((y Y))
  X)
```

and z is a Y, then

```
(f z)
```

is an X

where every y has been consistently replaced by z.

The Final First Commandment of λ

If two λ-expressions can be made the same

```
(Π ((y Y))
  X),
```

by consistently renaming their variables, then they are the same.

The Final Second Commandment of λ

If f is a

```
(Π ((y Y))
  X),
```

and y does not occur in f, then f is the same as

```
(λ (y)
  (f y)).
```

The type

```
(Π ((es (Vec E (add1 ℓ))))
  E)
```

could have been written

```
(→ (Vec E (add1 ℓ))
  E)
```

because *es* is not used in *E*.

We could also have written ***first***'s claim with a single Π-expression, and no →.

41 This last version of ***first*** could have been written like this.

```
(claim first
  (Π ((E 𝒰)
       (ℓ Nat)
       (es (Vec E (add1 ℓ))))
     E))
(define first
  (λ (E ℓ es)
    (head es)))
```

This is because nested Π-expressions could have been written as a single Π-expression.

A more specific type made it possible to define ***first***, our own typed version of **head**.

Is a more specific type needed to define ***rest***, our own version of **tail**?

42 Yes, it is, because (**tail** vecnil) is as meaningless as (**head** vecnil).

What is that more specific type?

43 The argument must have vec:: at the top.

Because the **head** is not part of the tail, the resulting Vec is shorter.

```
(claim rest
  (Π ((E 𝒰)
      (ℓ Nat))
    (→ (Vec E (add1 ℓ))
      (Vec E ℓ))))
```

Both **head** and **tail** are functions, and all functions are total. This means that they cannot be used with List because List does not rule out nil.

Now define ***rest***.

44 Here it is.

```
(define rest
  (λ (E ℓ)
    (λ (es)
      (tail es))))
```

Save those mushrooms

the oven is hot, and it's almost time to bake the Π.

7

It All Depends on the Motive

1 Our mushroom pot pie requires quite a few peas. Now it is time to define ***peas***, which produces as many peas as required.

What type expresses this behavior?

The type is

```
(→ Nat
  (List Atom))
```

because ***peas*** should be able to produce any number of peas.

---

2 How many peas should ***peas*** produce?

It depends.

---

3 What does it depend on?

It depends on how many peas are required—that is, the argument.

---

4 The type in frame 1,

```
(→ Nat
  (List Atom)),
```

is not specific enough. It does not express that ***peas*** produces *precisely* as many peas as were asked for.

The number of peas is the Nat argument. Does this type do the trick?

```
(claim peas
  (Π ((how-many-peas Nat))
    (Vec Atom how-many-peas)))
```

---

5 Yes, and the type expresses that the number of peas as the argument to ***peas*** depends on the number asked for. Such types are called *dependent types*.

Can ***peas*** be written using **rec-Nat**?

Sure.

```
(define peas
  (λ (how-many-peas)
    (rec-Nat how-many-peas
      vecnil
      (λ (ℓ-1 peas_ℓ-1)
        (vec:: 'pea peas_ℓ-1)))))
```

Dependent Types

A type that is determined by something that is not a type is called a *dependent type.*

The definition of ***peas*** is not an expression. To use **rec-Nat**, the base must have the same type as the $peas_{\ell\text{-}1}$ argument to the step. Here, though, the $peas_{\ell\text{-}1}$ might be a (Vec Atom 29), but vecnil is a (Vec Atom 0).

In other words, **rec-Nat** cannot be used when the type depends on the target Nat.

6 What about **iter-Nat**?

---

rec-Nat can do anything that **iter-Nat** can.

7 Is there something more powerful to use?

It is called **ind-Nat**, short for "induction on Nat."

8 What is **ind-Nat**?

---

ind-Nat is like **rec-Nat**, except it allows the types of the base and the almost-answer in the step, here $peas_{\ell\text{-}1}$, to include the target Nat.

In other words, **ind-Nat** is used for dependent types.

9 There is a Nat called *how-many-peas* included in
(Vec Atom *how-many-peas*).

Is this a dependent type?

Yes, it depends on the Nat *how-many-peas*.

To work with dependent types, **ind-Nat** needs an extra argument: to use **ind-Nat**, it is necessary to state *how* the types of both the base and the step's almost-answer depend on the target Nat.

10 What does this extra argument look like?

11 This extra argument, called the *motive*,[†] can be any

(→ Nat
  $\mathcal{U}$).

An ind-Nat-expression's type is the motive applied to the target Nat.

[†]Thanks, Conor McBride (1973–).

So the motive is a function whose body is a $\mathcal{U}$.

12 It is. The motive explains *why* the target is to be eliminated.

What is the motive for ***peas***?

That's a good question.

At least its type is clear.

```
(claim mot-peas†
  (→ Nat
    𝒰))
```

[†]"mot" is pronounced "moat."

Use ind-Nat for Dependent Types

Use ind-Nat instead of rec-Nat when the rec-Nat- or ind-Nat-expression's type depends on the target Nat. The ind-Nat-expression's type is the motive applied to the target.

13 Here is ***mot-peas***.

```
(define mot-peas
  (λ (k)
    (Vec Atom k)))
```

What is the value of (***mot-peas*** zero)?

It is the $\mathcal{U}$, and thus also the type,

(Vec Atom zero).

What type must the base for ***peas*** have?

14 Surely it must have the type

(Vec Atom zero),

because the value of the base is the value when zero is the target.

What should the base for ***peas*** be?

15 It must be vecnil because vecnil is the only

(Vec Atom zero).

This is also (***mot-peas*** zero).

What is the purpose of a step in **rec-Nat**?

16 In **rec-Nat**, the step's arguments are *n-1* and the almost-answer, which is the value from eliminating *n-1*.

Given *n-1* and the almost-answer, the step determines the value for (add1 *n-1*).

The step's arguments in **ind-Nat** are also *n-1* and an almost-answer.

What is the almost-answer's type?

17 The almost-answer's type is the motive applied to *n-1* because the almost-answer is the value for the target *n-1*.

What is the type of the value for the target (add1 *n-1*)?

18 The type of an **ind-Nat**-expression is the motive applied to the target.

If the motive is called *mot*, then the step's type is

```
(Π ((n-1 Nat))
  (→ (mot n-1)
    (mot (add1 n-1)))).
```

19 What is an example of a step for **ind-Nat**?

Here is the step for ***peas***.

```
(claim step-peas
  (Π ((ℓ-1 Nat))
    (→ (mot-peas ℓ-1)
      (mot-peas (add1 ℓ-1)))))
(define step-peas
  (λ (ℓ-1)
    (λ (peasℓ-1)
      (vec:: 'pea peasℓ-1))))
```

[20] Why does ***mot-peas*** appear twice in ***step-peas***'s type?

Good question.

What is the value of (***mot-peas*** *ℓ-1*)?

[21] It is (Vec Atom *ℓ-1*).

The Law of ind-Nat

If *target* **is a** Nat**,** *mot* **is an**

```
(→ Nat
  𝒰),
```

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

```
(Π ((n-1 Nat))
  (→ (mot n-1)
    (mot (add1 n-1)))),
```

then

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

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

The First Commandment of ind-Nat

The ind-Nat-expression

```
(ind-Nat zero
  mot
  base
  step)
```

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

The Second Commandment of ind-Nat

The ind-Nat-expression

```
(ind-Nat (add1 n)
  mot
  base
  step)
```

and

```
(step n
  (ind-Nat n
    mot
    base
    step))
```

are the same (*mot* (add1 *n*)).

This is $peas_{\ell\text{-}1}$'s type, which describes a list containing ℓ-*1* peas.

What is the value of

(***mot-peas*** (add1 ℓ-*1*)),

and what does it mean?

22 It is

(Vec Atom (add1 ℓ-*1*)),

which describes a list containing

(add1 ℓ-*1*)

peas.

Induction on Natural Numbers

Building a value for any natural number by giving a value for zero and a way to transform a value for n into a value for $n+1$ is called *induction on natural numbers.*

The step must construct a value for (add1 *ℓ-1*) from a value for *ℓ-1*.

Look at ***step-peas***'s type again. What does it mean in prose?

23 No matter what Nat *ℓ-1* is, ***step-peas*** can take a

(Vec Atom *ℓ-1*)

and produce a

(Vec Atom (add1 *ℓ-1*)).

It does this by "consing" a 'pea to the front.

The base replaces

zero

with

vecnil

because

vecnil

is the only

(Vec Atom zero).

What does ***step-peas*** replace an add1 with?

24 ***step-peas*** replaces each add1 with a vec::, just as ***length*** in frame 5:34 replaces each :: in a list with add1.

Now it is possible to define ***peas***, using ***mot-peas*** and ***step-peas***.

25 Here is the definition.

```
(define peas
  (λ (how-many-peas)
    (ind-Nat how-many-peas
      mot-peas
      vecnil
      step-peas)))
```

What is the value of (***peas*** 2)?

Here are the first two steps.

```
1. (peas
     (add1
       (add1 zero)))
2. (ind-Nat (add1
              (add1 zero))
     mot-peas
     vecnil
     step-peas)
3. (step-peas (add1 zero)
     (ind-Nat (add1 zero)
       mot-peas
       vecnil
       step-peas))
```

Now, find its value. Remember that arguments need not be evaluated.

26 Here it is,

```
4. (vec:: 'pea
     (ind-Nat (add1 zero)
       mot-peas
       vecnil
       step-peas)).
```

And finally, we find its normal form,

```
5. (vec:: 'pea
     (step-peas zero
       (ind-Nat zero
         mot-peas
         vecnil
         step-peas)))
6. (vec:: 'pea
     (vec:: 'pea
       (ind-Nat zero
         mot-peas
         vecnil
         step-peas)))
7. (vec:: 'pea
     (vec:: 'pea vecnil)),
```

which is normal.

If the motive's argument is dim, then **ind-Nat** works just like **rec-Nat**. Define a function ***also-rec-Nat*** using **ind-Nat** that works just like **rec-Nat**.

```
(claim also-rec-Nat
  (Π ((X 𝒰))
    (→ Nat
        X
        (→ Nat X
          X)
      X)))
```

27 The type does not depend on the target, so *k* is dim.

```
(define also-rec-Nat
  (λ (X)
    (λ (target base step)
      (ind-Nat target
        (λ (k)
          X)
        base
        step))))
```

Just as ***first*** finds the first entry in a list, ***last*** finds the last entry.

What type should ***last*** have?

28 The list must be non-empty, which means that we can use the same idea as in ***first***'s type.

```
(claim last
  (Π ((E 𝒰)
      (ℓ Nat))
    (→ (Vec E (add1 ℓ))
      E)))
```

If a list contains only one Atom, which Atom is the last one?

29 There is only one possibility.

What is the normal form of

```
(last Atom zero
  (vec:: 'flour vecnil))?
```

30 Here is a guess. The question has no meaning, because that list contains one rather than zero entries.

What is (***last*** Atom zero)'s type?

Remember Currying.

31 (***last*** Atom zero)'s type is

```
(→ (Vec Atom (add1 zero))
  Atom).
```

So the question in the preceding frame does, in fact, have a meaning!

What is the normal form of

```
(last Atom zero
  (vec:: 'flour vecnil))?
```

32 It must be 'flour.

Yes, indeed.

Using this insight, what is ***base-last***'s type?

33 The base is used when the Nat is zero.

```
(claim base-last
  (Π ((E 𝒰))
    (→ (Vec E (add1 zero))
      E)))
```

What is the definition of ***base-last***?

34 It uses **head** to obtain the only entry in a (Vec Atom (add1 zero)).

```
(define base-last
  (λ (E)
    (λ (es)
      (head es))))
```

This is the first time that the base is a function. According to the motive, both the base and the step's almost-answer are functions.

When the base is a function and the step transforms an almost-function into a function, the **ind-Nat**-expression constructs a function as well.

35 Are λ-expressions values?

Yes, because λ is a constructor.

36 Functions are indeed values.

The **ind-Nat**-expression's type is the motive applied to the target, which is the Nat being eliminated.

What is the target Nat when the base is reached?

37 It is zero. That is what it means to be the base.

The motive applied to zero should be the base's type.

Find an expression that can be used for the motive.

38 How about

```
(Π ((E 𝒰)
    (k Nat))
  (→ (Vec E (add1 k))
    E))?
```

Filling in E with the entry type and k with zero yields the base's type.

ind-Nat's Base Type

In ind-Nat, the base's type is the motive applied to the target zero.

That's close, but not quite correct.

The motive for **ind-Nat** should be applied to zero, but applying a Π-expression doesn't make sense. The motive for **ind-Nat** is a function, not a function's type.

39 Oh, so it must be

```
(λ (E k)
  (→ (Vec E (add1 k))
    E)),
```

which can be applied to the entry type and zero to obtain the base's type.

Now define the motive for ***last***.

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

40 Here it is.

```
(define mot-last
  (λ (E k)
    (→ (Vec E (add1 k))
      E)))
```

What is the type and value of

```
(mot-last Atom)?
```

41 The type is

```
(→ Nat
  𝒰)
```

and the value is

```
(λ (k)
  (→ (Vec Atom (add1 k))
    Atom)).
```

What does this resemble?

42 ***twin-Atom*** from frame 4:54. Applying ***mot-last*** to a 𝒰 results in a suitable motive for **ind-Nat**.

What is the value of the base's type, which is (***mot-last*** Atom zero)?	43	It is the type (→ (Vec Atom (add1 zero)) Atom).
What is the value of (***mot-last*** Atom (add1 *ℓ-1*))?	44	It is (→ (Vec Atom (add1 (add1 *ℓ-1*))) Atom).
What is the purpose of the step for ***last***?	45	The step for ***last*** turns the almost-answer for *ℓ-1* into the answer for (add1 *ℓ-1*). In other words, the step for ***last*** changes a function that gets the last entry in a (Vec *E* (add1 *ℓ-1*)) to a function that gets the last entry in a (Vec *E* (add1 (add1 *ℓ-1*))). Why are there two add1s?
The outer add1 is part of the type in order to ensure that the list given to ***last*** has at least one entry. The inner add1 is from the (add1 *ℓ-1*) passed to ***mot-last***.	46	The outer add1 makes the function total, and the inner add1 is due to the Law of **ind-Nat**.

What is the step's type?

47 The step's type must be

```
(→ (→ (Vec E (add1 ℓ-1))
      E)
   (→ (Vec E (add1
               (add1 ℓ-1)))
      E))
```

because the step must construct a

(***mot-last*** *E* (add1 *ℓ-1*))

from a

(***mot-last*** *E* *ℓ-1*).

How can that type be explained in prose?

48 The step transforms a

last

function for

ℓ

into a

last

function for

(add1 *ℓ*).

---

ind-Nat's Step Type

In ind-Nat, the step must take two arguments: some Nat *n* and an almost-answer whose type is the motive applied to *n*. The type of the answer from the step is the motive applied to (add1 *n*). The step's type is:

```
(Π ((n Nat))
  (→ (mot n)
    (mot (add1 n))))
```

Here is ***step-last***'s claim.

```
(claim step-last
  (Π ((E 𝒰)
      (ℓ-1 Nat))
    (→ (mot-last E ℓ-1)
      (mot-last E (add1 ℓ-1)))))
```

Now define ***step-last***.

49 $last_{\ell\text{-}1}$ is almost the right function, but only for a list with *ℓ-1* entries, so it accepts the **tail** of a list with (add1 *ℓ-1*) entries as an argument.

```
(define step-last
  (λ (E ℓ-1)
    (λ (last_ℓ-1)
      (λ (es)
        (last_ℓ-1 (tail es))))))
```

What is *es*'s type in the inner λ-expression?

50 *es* is a

```
(Vec E (add1
         (add1 ℓ-1))).
```

Why is that *es*'s type?

51 The whole inner λ-expression's type is

```
(mot-last E (add1 ℓ-1)),
```

and that type and

```
(→ (Vec E (add1
            (add1 ℓ-1)))
  E)
```

are the same type. Thus, the argument to the λ-expression, namely *es*, is a

```
(Vec E (add1
         (add1 ℓ-1))).
```

Clever.

What is (**tail** *es*)'s type?

52 (**tail** *es*)'s type is

```
(Vec E (add1 ℓ-1)),
```

which is the type of a suitable argument for the almost-ready function.

53 What is $last_{\ell\text{-}1}$'s type in the outer λ-expression in frame 49?

$last_{\ell\text{-}1}$ is an

```
(→ (Vec E (add1 ℓ-1))
   E),
```

which is the value of (***mot-last*** *ℓ-1*).

---

54 Now it is time to define ***last***. The **claim** is in frame 28 on page 151.

Here goes.

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

---

55 What is the normal form of

```
(last Atom 1
  (vec:: 'carrot
    (vec:: 'celery vecnil)))?
```

Here is the beginning.

```
1. (last Atom (add1 zero)
     (vec:: 'carrot
       (vec:: 'celery vecnil)))
2. ((ind-Nat (add1 zero)
      (mot-last Atom)
      (base-last Atom)
      (step-last Atom))
    (vec:: 'carrot
      (vec:: 'celery vecnil)))
3. ((step-last Atom zero
      (ind-Nat zero
        (mot-last Atom)
        (base-last Atom)
        (step-last Atom)))
    (vec:: 'carrot
      (vec:: 'celery vecnil)))
```

Thanks for the help. There is more.

```
4. ((λ (es)
      ((ind-Nat zero
         (mot-last Atom)
         (base-last Atom)
         (step-last Atom))
       (tail es)))
    (vec:: 'carrot
      (vec:: 'celery vecnil)))
   ((ind-Nat zero
      (mot-last Atom)
      (base-last Atom)
      (step-last Atom))
    (tail
      (vec:: 'carrot
        (vec:: 'celery vecnil))))
5. (base-last Atom
     (tail
       (vec:: 'carrot
         (vec:: 'celery vecnil))))
```

Is that the normal form?

56 No, there are a few more steps.

```
 7. ((λ (es)
       (head es))
     (tail
       (vec:: 'carrot
         (vec:: 'celery vecnil))))
 8. (head
      (tail
        (vec:: 'carrot
          (vec:: 'celery vecnil))))
 9. (head
      (vec:: 'celery vecnil))
10. 'celery
```

Excellent.

Now take a quick break and have some fortifying mushroom pot pie.

57 That sounds like a good idea.

Guess what ***drop-last*** means.

58 Presumably, it drops the last entry in a Vec.

Good guess!

What is (***drop-last*** Atom 3 vecnil)?

59 It is not described by a type, for the same reason that

(***first*** Atom 3 vecnil),

(***last*** Atom 3 vecnil),

and

(***rest*** Atom 3 vecnil)

aren't described by types.

The type must contain a Vec with an add1 in it.

That's solid thinking.

What is ***drop-last***'s type?

60 ***drop-last*** shrinks a list by one.

```
(claim drop-last
  (Π ((E 𝒰)
      (ℓ Nat))
    (→ (Vec E (add1 ℓ))
      (Vec E ℓ))))
```

What is ***base-drop-last***?

61 The base finds the

drop-last

of a single-entry list, which is

vecnil

because the last entry is the only entry.

```
(claim base-drop-last
  (Π ((E 𝒰))
    (→ (Vec E (add1 zero))
      (Vec E zero))))
(define base-drop-last
  (λ (E)
    (λ (es)
      vecnil)))
```

Would this definition of ***base-drop-last*** also work?

```
(define base-drop-last
  (λ (E)
    (λ (es)
      (tail es))))
```

62 It always has the same value, but it does not convey the idea as clearly.

The intention is that ***base-drop-last*** ignores the last entry in the list.

That sounds right.

Why doesn't it deserve a solid box?

63 Getting the right answer is worthless if we do not *know* that it is correct. Understanding the answer is at least as important as having the correct answer.

Readable Expressions

Getting the right answer is worthless if we do not *know* that it is correct. Understanding the answer is at least as important as having the correct answer.

Someone has been paying attention!

What is ***mot-drop-last***?

64 ***mot-drop-last*** says that ***drop-last*** constructs a Vec with one fewer entries.

```
(claim mot-drop-last
  (→ 𝒰 Nat
    𝒰))
(define mot-drop-last
  (λ (E k)
    (→ (Vec E (add1 k))
      (Vec E k))))
```

That was fast. Please explain.

65 During ind-Nat, the motive applied to zero is the base's type. That means that we can work backwards by replacing the zeros in the base's type with the argument k to ***mot-drop-last*** in the base's type from frame 61.

That is a keen observation. This approach does not always work, but it is a good starting point.

Replacing a particular constant with a variable and wrapping a λ of the variable is called *abstracting over constants*, and it is used often. Here, the motive abstracts over zero in ***base-drop-last***.

66 ***step-drop-last***'s type follows the Law of ind-Nat.

```
(claim step-drop-last
  (Π ((E 𝒰)
      (ℓ-1 Nat))
    (→ (mot-drop-last E ℓ-1)
      (mot-drop-last E (add1 ℓ-1)))))
```

How should ***step-drop-last*** be defined?

67 ***step-drop-last*** keeps the head around.

```
(define step-drop-last
  (λ (E ℓ-1)
    (λ (drop-last_ℓ-1)
      (λ (es)
        (vec:: (head es)
          (drop-last_ℓ-1 (tail es)))))))
```

This is the familiar pattern of induction:

step-drop-last

transforms a

drop-last

that works for

(Vec *E* *ℓ-1*)

into a

drop-last

that works for

(Vec *E* (add1 *ℓ-1*)).

How does the transformation work?

68 Just as

step-last

uses its almost-answer, namely $\mathit{last}_{\ell\text{-}1}$, to find the ***last*** of its own (**tail** *es*),

step-drop-last

uses its almost-answer, $\mathit{drop\text{-}last}_{\ell\text{-}1}$, to find the ***drop-last*** of its own (**tail** *es*).

Based on ***mot-drop-last***, the function produced by ***step-drop-last*** must add an entry to that list. Thus, the inner λ-expression in frame 66 "cons"es (using vec::) the **head** of *es* onto the ($\mathit{drop\text{-}last}_{\ell\text{-}1}$ (**tail** *es*)).

The **claim** for ***drop-last*** is in frame 60 on page 159.

Now define ***drop-last***.

69 It's a matter of putting the pieces together.

```
(define drop-last
  (λ (E ℓ)
    (ind-Nat ℓ
      (mot-drop-last E)
      (base-drop-last E)
      (step-drop-last E))))
```

Yes, ***drop-last*** is now defined.

Sometimes, it can be convenient to find a function that can be used later. For example, (***drop-last*** Atom 2) finds the first two entries in any three-entry list of Atoms.

Show how this works by finding the value of

```
(drop-last Atom
  (add1
    (add1 zero))).
```

70 Here's the chart to find the value.

```
1. (drop-last Atom
     (add1
       (add1 zero)))
2. (ind-Nat (add1
              (add1 zero))
     (mot-drop-last Atom)
     (base-drop-last Atom)
     (step-drop-last Atom))
3. (step-drop-last Atom (add1 zero)
     (ind-Nat (add1 zero)
       (mot-drop-last Atom)
       (base-drop-last Atom)
       (step-drop-last Atom)))
4. (λ (es)
     (vec:: (head es)
       ((ind-Nat (add1 zero)
          (mot-drop-last Atom)
          (base-drop-last Atom)
          (step-drop-last Atom))
        (tail es))))
```

That's right—λ-expressions are values. To find the normal form, more steps are necessary. Here's the first one.

```
5. (λ (es)
     (vec:: (head es)
       ((step-drop-last Atom zero
          (ind-Nat zero
            (mot-drop-last Atom)
            (base-drop-last Atom)
            (step-drop-last Atom)))
        (tail es))))
```

Now find the normal form.

71 In step 6, *es* has been consistently renamed to *ys* to make it clear that the inner λ-expression has its own variable.

```
6. (λ (es)
     (vec:: (head es)
       ((λ (ys)
          (vec:: (head ys)
            ((ind-Nat zero
               (mot-drop-last Atom)
               (base-drop-last Atom)
               (step-drop-last Atom))
             (tail ys))))
        (tail es))))
```

It is not necessary to rename *es* to *ys*, because variable names always refer to their closest surrounding λ, but it's always a good idea to make expressions easier to understand.

Here are two more steps.

```
7. (λ (es)
     (vec:: (head es)
       (vec:: (head (tail es))
         ((ind-Nat zero
             (mot-drop-last Atom)
             (base-drop-last Atom)
             (step-drop-last Atom))
           (tail (tail es))))))
8. (λ (es)
     (vec:: (head es)
       (vec:: (head (tail es))
         (base-drop-last Atom
           (tail (tail es))))))
```

72 Almost there.

```
9. (λ (es)
     (vec:: (head es)
       (vec:: (head (tail es))
         ((λ (ys)
             vecnil)
           (tail (tail es))))))
10. (λ (es)
      (vec:: (head es)
        (vec:: (head (tail es))
          vecnil)))
```

The normal form is much easier to understand than the starting expression!

C'est magnifique! Bet you're tired.

73 Indeed. And hungry, too.

Eat the rest of that pot pie,

and head to a café if you're still hungry,
and re-read this chapter in a relaxed ambience.

One Piece at a Time

Sometimes, it is not immediately apparent how to write a Pie expression.

1 That's what empty boxes are for, right?

That's right. Most keyboards, however, do not make it particularly easy to type empty boxes.

Instead of typing empty boxes, it is possible to leave part of an expression to be written later using the `TODO` form.

2 What is `TODO`?

`TODO` is an expression that is a placeholder for another expression. A `TODO` can have any type, and Pie keeps track of which `TODO`s have which types.

3 How can `TODO` be used?

Each `TODO` comes from somewhere specific. Here, we refer to them by frame number; when using Pie outside of a book, this will be somewhere else that is appropriate.

Try typing

```
(claim peas
  TODO)
```

and see what happens.

4 Pie responds with

```
Frame 4:2.3: TODO: U
```

and the `TODO` it's mentioning is indeed a $\mathcal{U}$ on the second line and third column of an expression in frame 4.

Now try

```
(claim peas
  (Pi ((n Nat))
    TODO))
```

which is closer to the type for ***peas*** in chapter 7.

5 Pie responds with

```
Frame 5:3.5: TODO:
 n : Nat
--------------
 U
```

but what does that horizontal line mean?

When Pie replies with the type that is expected for a `TODO`, it also includes the types of the variables that can be used at the `TODO`'s position.

6 The `n : Nat` above the line means that the variable *n* is a Nat.

That's right.

Now try

```
(claim peas
  (Pi ((n Nat))
    (Vec Atom n)))
(define peas
  TODO)
```

where the `TODO` is in a definition.

7 Pie replies with

```
Frame 7:5.3: TODO:
 (Π ((n Nat))
   (Vec Atom n))
```

which is the type that was **claim**ed.

How does Pie respond when provided with a λ around the `TODO`?

```
(claim peas
  (Pi ((n Nat))
    (Vec Atom n)))
(define peas
  (λ (n)
    TODO))
```

8 There will be a line for `n` above the horizontal line.

Try it and see.

9 That's what happened.

```
Frame 8:6.5: TODO:
 n : Nat
--------------
 (Vec Atom n)
```

What's next?

10 The number of `'`peas depends on *n*, so **ind-Nat** is needed.

How does Pie respond to this version of ***peas***?

```
(claim peas
  (Pi ((n Nat))
    (Vec Atom n)))
(define peas
  (λ (n)
    (ind-Nat n
      (λ (k)
        (Vec Atom k))
      TODO
      TODO)))
```

11 Each `TODO` has the type that would be expected by the Law of **ind-Nat**.

```
Frame 11:9.7: TODO:
 n : Nat
--------------
 (Vec Atom 0)

Frame 11:10.7: TODO:
 n : Nat
--------------
 (Π ((n-1 Nat))
   (→ (Vec Atom n-1)
     (Vec Atom
       (add1 n-1))))
```

How should the `TODO`s be replaced?

12 The first `TODO` should be a (Vec Atom 0), so vecnil is appropriate. The second `TODO` should be a two-argument function, built with λ, that uses vec:: to add a 'pea to *n-1* peas.

Nicely chosen. How does Pie respond to this version?

```
(claim peas
  (Pi ((n Nat))
    (Vec Atom n)))
(define peas
  (λ (n)
    (ind-Nat n
      (λ (k)
        (Vec Atom k))
      vecnil
      (λ (n-1 peas-of-n-1)
        (vec:: TODO TODO)))))
```

13 The Law of vec:: determines the type of each `TODO`.

```
Frame 13:11.16: TODO:
           n : Nat
         n-1 : Nat
 peas-of-n-1 : (Vec Atom n-1)
------------------------------
 Atom

Frame 13:11.21: TODO:
           n : Nat
         n-1 : Nat
 peas-of-n-1 : (Vec Atom n-1)
------------------------------
 (Vec Atom n-1)
```

Now replace the final TODOs.

[14] Here is the final definition.

```
(claim peas
  (Pi ((n Nat))
    (Vec Atom n)))

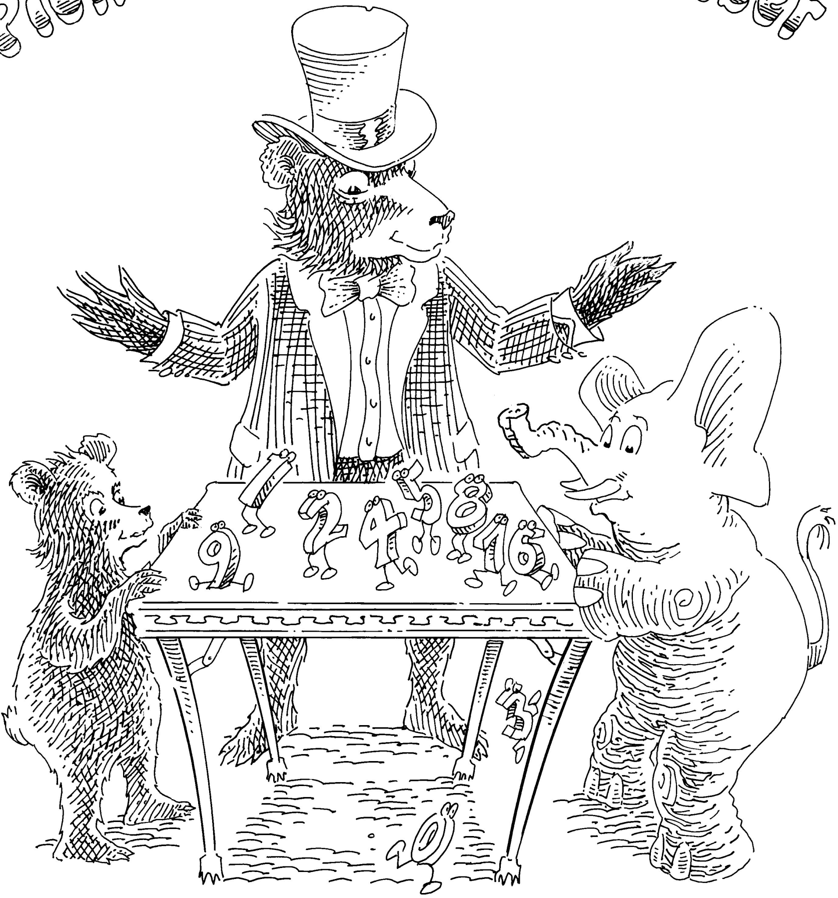
(define peas
  (λ (n)
    (ind-Nat n
      (λ (k)
        (Vec Atom k))
      vecnil
      (λ (n-1 peas-of-n-1)
        (vec:: 'pea
          peas-of-n-1)))))
```

Go eat a mushroom pot pie that contains `n` peas,

one delicious bite at a time.

8

Pick a Number, Any Number



1 How was that mushroom pot pie?

Delicious, though very filling. How about something less filling here?

2 How about a (***sandwich*** 'hoagie)?

That should be manageable.

3 What is the normal form of (**+** 1)?

Let's find out.

4 Let's start.

```
1. (+ (add1 zero))
2. (λ (j)
     (iter-Nat (add1 zero)
       j
       step-+))
```

This is the value.

The normal form needs a bit more work.

```
3. (λ (j)
     (step-+
       (iter-Nat zero
         j
         step-+)))
4. (λ (j)
     (add1
       (iter-Nat zero
         j
         step-+)))
5. (λ (j)
     (add1 j))
```

5 Here is another definition.

```
(claim incr
  (→ Nat
    Nat))
(define incr
  (λ (n)
    (iter-Nat n
      1
      (+ 1))))
```

What is the normal form of (***incr*** 0)?

That's just three steps.

```
1. (incr zero)
2. (iter-Nat zero
     1
     (+ 1))
3. 1
```

It is 1, also known as (add1 zero).

What is the normal form of (*incr* 3)?

6 That normal form takes a few more steps. The first steps are to find the value.

```
1. (iter-Nat 3
     1
     (+ 1))
2. (+ 1
     (iter-Nat 2
       1
       (+ 1)))
3. (add1
     (iter-Nat 2
       1
       (+ 1)))
```

That is indeed the value. But what is the normal form? Here's a few more steps.

```
4. (add1
     (+ 1
       (iter-Nat (add1 zero)
         1
         (+ 1))))
5. (add1
     (add1
       (iter-Nat (add1 zero)
         1
         (+ 1))))
```

7 The normal form is 4.

```
6. (add1
     (add1
       (+ 1
         (iter-Nat zero
           1
           (+ 1)))))
7. (add1
     (add1
       (add1
         (iter-Nat zero
           1
           (+ 1)))))
8. (add1
     (add1
       (add1
         1)))
```

What is the relationship between

(**+** 1)

and

incr?

8 They both find the same answer, no matter what the argument is.

Does this mean that (**+** 1) is the same

```
(→ Nat
  Nat)
```

as ***incr***?

9 They are the same if they have the same normal form.

The normal form of (**+** 1) is

```
(λ (n)
  (add1 n)),
```

while the normal form of ***incr*** is

```
(λ (n)
  (iter-Nat n
    1
    (λ (j)
      (add1 j)))).
```

They are not the same.

That's right.

Even though they are not the same, the fact that they always find the same answer can be written as a type.

10 But isn't sameness a judgment, not a type?

Sameness is indeed a judgment. But, with a new type constructor, types can express a new idea called *equality*.

Writing

"***incr*** and (**+** 1) always find the same answer."

as a type is hungry work. You'd better have this

```
(sandwich 'grinder)
```

to keep your energy up.

11 Another sandwich?

Okay.

An expression
(**=** X *from* *to*)
is a type if
X
is a type,
from
is an X, and
to
is an X.

12 Is this another way to construct a dependent type?

The Law of =

An expression
(= X *from* *to*)
is a type if X is a type, *from* is an X, and *to* is an X.

Yes, **=** is another way to construct a dependent type, because *from* and *to* need not be types.

Because *from* and *to* are convenient names, the corresponding parts of an **=**-expression are called the FROM and the TO.

13 Okay.

Reading FROM and TO as Nouns

Because *from* and *to* are convenient names, the corresponding parts of an =-expression are referred to as the FROM and the TO.

14

Is

(**=** Atom 'kale 'blackberries)

a type?

Yes, because Atom is a type and both 'kale and 'blackberries are Atoms.

15

Is

(**=** Nat (**+** 1 1) 2)†

a type?

†Thank you, Alfred North Whitehead (1861–1947) and again Bertrand Russell. Page 379 of *Principia Mathematica*, their 3-volume work published respectively, in 1910, 1912, and 1913, states, "From this proposition it will follow, when arithmetical addition has been defined, that $1 + 1 = 2$."

Yes, because Nat is a type and both (**+** 1 1) and 2 are Nats.

16

Is

```
(= (car (cons Nat 'kale))
   17
   (+ 14 3))
```

a type?

Yes, it is, because

(**car** (cons Nat 'kale))

and Nat are the same type, and the FROM and the TO are both Nats.

17

Is

```
(= (car (cons Nat 'kale))
   15
   (+ 14 3))
```

a type?

Yes, it is. Frame 12 requires only that the FROM and the TO are Nats, not that they are the same Nat.

But what is the purpose of **=**?

18

To understand **=**, it is first necessary to gain a new perspective on types.

Types can be read as *statements*.†

†Thank you Robert Feys (1889–1961), Nicolaas Govert de Bruijn (1918–2012), and again Haskell B. Curry. Thanks William Alvin Howard (1926–). Statements are sometimes called *propositions*.

How can

(**=** Atom 'apple 'apple)

be read as a statement?

The type

(= Atom 'apple 'apple)

can be read:

"The expressions 'apple and 'apple are equal Atoms."

How can

(= Nat (+ 2 2) 4)

be read as a statement?

19 Does

"Two plus two equals four"

make sense?

Yes, it does.

20 In what way do

"Three plus four equals seven"

and

(+ 3 4) is the same Nat as 7

differ?

The statement

"Three plus four equals seven"

is another way of writing the type

(= Nat (+ 3 4) 7),

which *is* an expression, but

(+ 3 4) is the same Nat as 7

is a judgment *about* expressions.

Frame 1:12 describes judgments. A judgment is not an expression—rather, a judgment is an attitude that a person takes when thinking about expressions.

21 Here's a judgment:

"Three plus four equals seven" is a type.

Well-spotted.

=-expressions are not the only types that can be read as statements.

22 What are some others?

A Π-expression can be read as "for every." Consider this example,

(Π ((*n* Nat))
  (= Nat (+ 1 *n*) (add1 *n*)))

can be read as

"For every Nat *n*, (+ 1 *n*) equals (add1 *n*)."

23 Okay. But what is the point of reading types as statements?

If a type can be read as a statement, then judging the statement to be true means that there is an expression with that type. So, saying

"(+ *n* 0) and *n* are equal Nats."

means

"There is an expression with type
  (= Nat (+ *n* 0) *n*)."

24 Does this mean that truth requires evidence?

It goes further. Truth *means* that we have evidence.† This evidence is called a *proof*.

†Thank you, BHK: L. E. J. Brouwer (1881–1966), Arend Heyting (1898–1980), and Andrey Kolmogorov (1903–1987).

25 Can every type be read as a statement?

In principle, they could be, but many types would be very uninteresting as statements.

26 What makes a statement interesting?

A person does, by being interested in it. But most interesting statements come from dependent types. Nat is not an interesting statement because it is too easy to prove.

27 How can Nat be proved?

Pick a number, any number.

28 Okay,

15.

Good job. You have a proof.

29 That isn't very interesting.

Right.

30 That explains it.

Another way to think about statements is as an expectation of a proof, or as a problem to be solved.

31 Having seen a **claim**, it makes sense to expect a definition.

Frame 12 describes when an **=**-expression is a type, but it says nothing about what the values of such a type are.

32 Here, "values" means the same thing as "proofs," right?

Exactly right.

There is only one constructor for **=**, and it is called same. same takes one argument.

33 How is same used?

The expression

(same e)

is an

(**=** X e e)

if e is an X.

34 What is an example of this?

The Law of same

The expression (same e) is an (= X e e) if e is an X.

The expression

(same 21)

is an

(**=** Nat (**+** 17 4) (**+** 11 10)).

35 That doesn't seem right.

In frame 34, same's argument as well as the FROM and TO arguments of **=** have to be identical, but here, 21,

(**+** 17 4)

and

(**+** 11 10)

look rather different.

Both

(**+** 17 4)

and

(**+** 11 10)

are the same Nat as 21, so they are the same.

36 Does this mean that

(same (***incr*** 3))

is an

(**=** Nat (**+** 2 2) 4)?

Yes,

(same (***incr*** 3))

is a proof of

(**=** Nat (**+** 2 2) 4).

The Law of same uses e twice to require that

the FROM is the same X as the TO.

With the type constructor **=** and its constructor same, expressions can now state ideas that previously could only be judged.†

†Creating expressions that capture the ideas behind a form of judgment is sometimes called *internalizing* the form of judgment.

37 Why is this so important?

Expressions can be used together with other expressions.

By combining Π with =, we can write statements that are true for arbitrary Nats, while we could only make judgments about particular Nats. Here's an example:

```
(claim +1=add1
  (Π ((n Nat))
    (= Nat (+ 1 n) (add1 n))))
```

38 The definition of ***+1=add1*** clearly has a λ at the top because its type has a Π at the top.

```
(define +1=add1
  (λ (n)
                ))
```

That's a solid start. What goes in the box?

39 Following the Law of λ,

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

is the type of the body of the λ-expression.

Right, the box should contain an

(= Nat (+ 1 *n*) (add1 *n*)).

What is the normal form of the box's type?

40 The normal form of the box's type is

(= Nat (add1 *n*) (add1 *n*))

because the normal form of

(+ 1 *n*)

is

(add1 *n*).

Okay, so the expression in the box in frame 38 is

(same (add1 *n*)).

That's right.

Now finish the definition.

41 Here it is.

```
(define +1=add1
  (λ (n)
    (same (add1 n))))
```

What statement does ***+1=add1*** prove?

42 The statement is

"For every Nat n, (**+** 1 n) equals (add1 n)."

Here is another statement.

"For every Nat n, (***incr*** n) is equal to (add1 n)."

Translate it to a type.

43 Let's call it ***incr=add1***.

```
(claim incr=add1
  (Π ((n Nat))
    (= Nat (incr n) (add1 n))))
```

Now define ***incr=add1***.

44 Isn't it just like ***+1=add1***?

```
(define incr=add1
  (λ (n)
    (same (add1 n))))
```

Not quite. What is the normal form of (***incr*** n)?

45

```
1. (incr n)
2. (iter-Nat n
     1
     (+ 1))
```

The normal form is *not* the same Nat as (add1 n).

That's right. This normal form is neutral.

What is a neutral expression?

46 Neutral expressions are described in frame 2:24 on page 39.

Neutral expressions are those that cannot yet be evaluated.

Why is

```
(iter-Nat n
  1
  (+ 1))
```

neutral?

47 Because **iter-Nat** chooses the base when the target is zero, or the step when the target has add1 at the top. But n is neither.

48 A more precise way to define *neutral expressions* is to start with the simplest neutral expressions and build from there.

Variables are neutral, unless they refer to definitions, because a defined name is the same as its definition (see page 43).

Okay.

49 Also, if the target of an eliminator expression is neutral, then the entire expression is neutral.

So,

```
(iter-Nat n
  1
  (+ 1))
```

is neutral because **iter-Nat** is an eliminator and its target, *n*, is a variable.

Is every expression that contains a variable neutral?

Neutral Expressions

Variables that are not defined are neutral. If the target of an eliminator expression is neutral, then the eliminator expression is neutral.

50 No.

The body of the λ-expression

```
(λ (x)
  (add1 x))
```

contains the variable *x*, but λ-expressions are values, not neutral expressions.

But if the *whole* expression were just (add1 *x*), then it would be neutral because it would contain the neutral *x*.

Are neutral expressions normal?

Not always.

Some types have ways of making neutral expressions into values, and in these cases, the neutral expression is not considered normal, because it can be made into a value.

51 Which types work this way?

A neutral expression whose type has Π at the top is not normal. This is because a neutral expression f is the same as

```
(λ (x)
  (f x)),
```

which is a value.

52 Why does this mean that f is not normal?

What does it mean for an expression to be normal?

53 The big box on page 13 states that if two expressions are the same, then they have identical normal forms.

By the Second Commandment of λ[†] on page 140, f is the same as

```
(λ (x)
  (f x)),
```

but they are *not* written identically.

[†] Commandments such as this one are often called η- (pronounced "eta") rules. These normal forms in which all possible η-rules have been applied to make values are called *η-long* normal forms.

54 One is wrapped in a λ, the other is not.

At most one of them can be the normal form. The one wrapped in λ is the normal form. Because expressions with λ at the top are values, they are not neutral. Neutral expressions do not have a constructor at the top.

55 Are there any others?

Yes.

Because of the Second Commandment of cons from page 44, if p is a

  (Pair A D),

then p is the same as

  (cons (**car** p) (**cdr** p)).

For the very same reason, the only normal forms for pairs are expressions with cons at the top, so there are no neutral pairs that are normal.

56 Where do neutral expressions come from?

Neutral expressions, like (***incr*** n)'s normal form in frame 45, occur frequently when **=**-expressions mention argument names in Π-expressions.

57 How can we find a definition for ***incr=add1***? same does not do the job, after all, and ***incr=add1***'s type has a neutral expression in it.

Judgments, like

  (***incr*** n) is the same Nat as (add1 n),

can be mechanically checked using relatively simple rules. This is why judgments are a suitable basis for knowledge.

Expressions, however, can encode interesting patterns of reasoning, such as using induction to try each possibility for the variable in a neutral expression.

58 Does this mean that induction can be used to prove that (***incr*** n) and (add1 n) are equal, even though they are not the same?

Yes, using **ind-Nat** because the type depends on the target.

```
(define incr=add1
  (λ (n)
    (ind-Nat n
      mot-incr=add1
      base-incr=add1†
      step-incr=add1)))
```

What is the type of ***base-incr=add1***?

†Names like ***base-incr=add1*** should be read "the base for ***incr=add1***," not as "*base-incr* equals add1."

59 The base's type in an **ind-Nat**-expression is the motive applied to zero. (***incr*** zero) is *not* neutral, and its normal form is (add1 zero) as seen in frame 5, so it is the same Nat as (add1 zero).

```
(claim base-incr=add1
  (= Nat (incr zero) (add1 zero)))
(define base-incr=add1
  (same (add1 zero)))
```

Now abstract over the constant zero in ***base-incr=add1***'s type to define ***mot-incr=add1***.

60 Each zero becomes *k*.

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

Following the Law of **ind-Nat**, what is ***step-incr=add1***'s type?

Use a dashed box for now.

61 It is found using ***mot-incr=add1***. But why is it in a dashed box?

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

Solid boxes are used when the final version of a claim or definition is ready. Even though this is the correct type, it can be written in a way that is easier to understand.

62 What is that easier way of writing it?

Here is another way to write ***step-incr=add1***'s type.

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

63 Why is that the same type?

Because

(***mot-incr=add1*** *n-1*)

and

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

are the same type.†

What is the value of

(***mot-incr=add1*** (add1 *n-1*))?

†This uses the fourth form of judgment.

64 The value is

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

which is the other type in the →-expression in frame 63.

How can that type be read as a statement?

65 Hard to say.

How can →-expressions be read as statements?

The expression

$(\rightarrow X$
$\quad Y)$

can be read as the statement,

"if X, then Y."

This works because its values are total functions that transform *any* proof of X into a proof of Y.

66 Here goes.

The step's type is a Π-expression, which means that the statement starts with "every." After that is an →, which can be read as "if" and "then." And = can be read as "equals."

"If" and "Then" as Types

The expression

$(\rightarrow X$
$\quad Y)$

can be read as the statement,

"if X then Y."

How can ***step-incr=add1***'s type be read as a statement?

67 "For every Nat n,
if
(***incr*** n) equals (add1 n),
then
(***incr*** (add1 n))
equals
(add1 (add1 n))."

Unlike previous statements, to prove *this* statement, we must observe something about ***incr***.

What is the normal form of

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

68 The **iter-Nat** gets stuck on *n-1*, but an add1 does make it to the top.

1. ```
   (incr
     (add1 n-1))
   ```
2. ```
   (iter-Nat (add1 n-1)
     1
     (+ 1))
   ```
3. ```
   (+ 1
     (iter-Nat n-1
       1
       (+ 1)))
   ```
4. ```
   (add1
     (iter-Nat n-1
       1
       (+ 1)))
   ```
5. ```
   (add1
     (iter-Nat n-1
       1
       (λ (x)
         (add1 x))))
   ```

In other words,

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

is the same Nat as

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

because (***incr*** *n-1*) is the same Nat as

```
(iter-Nat n-1
  1
  (λ (x)
    (add1 x))).
```

This is the observation.

69 Okay, so the type of ***step-incr=add1*** can also be written this way. There is a gray box around the part that is different from the version in frame 63.

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

The box is now solid because it is easy to see why this type makes sense. If two Nats are equal, then one greater than both of them are also equal.

70 Okay. But how can it be made true with a proof?

Observation about *incr*

No matter which Nat n **is,**

(***incr*** (add1 n))

is the same Nat **as**

(add1 (***incr*** n)).

Here's the start of a definition of ***step-incr=add1***.

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

71 The almost-proof for *n-1* is an

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

What can be used in the white box to turn an almost-proof into a proof of

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

cong[†] is an eliminator for **=** that is useful here.

[†]Short for "congruence."

72 What is a **cong**-expression?

First things first. It's time to sit back and have a

(***sandwich*** **'**submarine).

73 Another sandwich?

This is a bit too much to eat.

Returning to the problem at hand,

(**cong** *target f*)

is used to transform both expressions that *target* equates using *f*.

74 Is there another way to look at **cong**?

If *f* is an

$(\rightarrow X$
$\quad Y)$

and *target* is an

(**=** *X from to*),

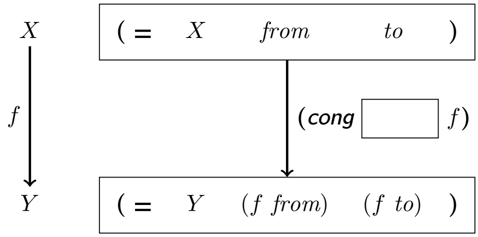
then

(**cong** *target f*)

is an

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

This diagram shows how **cong** is used.



75 How can **cong** be used to complete the definition of ***step-incr=add1***?

The Law of cong

If *f* is an

$(\rightarrow X$
$\quad Y)$

and *target* is an (= *X from to*),

then (cong *target f*) is an (= *Y* (*f from*) (*f to*)).

In this case, X is Nat, Y is Nat, and *target* is *incr=add1*$_{n\text{-}1}$.

What are *from* and *to*?

76 *incr=add1*$_{n\text{-}1}$'s type is

(**=** Nat (***incr*** *n-1*) (add1 *n-1*)),

so *from* is (***incr*** *n-1*) and *to* is (add1 *n-1*).

What function f transforms

(***incr*** *n-1*) into (add1 (***incr*** *n-1*))

and

(add1 *n-1*) into (add1 (add1 *n-1*))?

77 In each case, an add1 is added to the top. How about using add1 for f?

---

add1 is a constructor, but it is not an expression when it is not used as the top of a Nat tucked under it.

An add1-expression must have an argument.

78 How about using ***incr*** for f?

While ***incr*** does indeed add one to its argument, it does not result in an add1 immediately when its argument is neutral.

79 How about using (**+** 1) for f?

An excellent choice. There is now an expression for the white box.

```
(cong incr=add1_{n-1} (+ 1))
```

80 Okay.

```
(define step-incr=add1
  (λ (n-1)
    (λ (incr=add1_{n-1})
      (cong incr=add1_{n-1} (+ 1)))))
```

It is now possible to define ***incr=add1***.

81 The motive, the base, and the step are now defined, so the previous definition of ***incr=add1*** in frame 59 is now solid.

```
(define incr=add1
  (λ (n)
    (ind-Nat n
      mot-incr=add1
      base-incr=add1
      step-incr=add1)))
```

It's time for another sandwich:
(***sandwich*** **'**hero).

82 Another one!

Yes, another one.

Why is **ind-Nat** needed in the definition of ***incr=add1***, but not in the definition of ***+1=add1***?

83 Because the normal form of (***incr*** *n*) is the neutral expression in frame 45, but based on the definition of **+**, the normal form of (**+** 1 *n*) is (add1 *n*).

Neutral expressions are those that cannot yet be evaluated, but replacing their variables with values could allow evaluation.

What is the type of
(***incr=add1*** 2)?

84 The expression
(***incr=add1*** 2)
is an
(**=** Nat (***incr*** 2) (add1 2)).

In other words, it is an
(**=** Nat 3 3)
because (***incr*** 2) is not neutral.

What is the normal form of

(***incr=add1*** 2)?

85 Here's the start of the chart.

1. (***incr=add1*** 2)
2. (**ind-Nat** (add1 1)
   mot-incr=add1
   base-incr=add1
   step-incr=add1)
3. (***step-incr=add1*** 1
   (**ind-Nat** 1
   mot-incr=add1
   base-incr=add1
   step-incr=add1))
4. (**cong** (**ind-Nat** (add1 0)
   mot-incr=add1
   base-incr=add1
   step-incr=add1)
   (**+** 1))

How is a **cong**-expression evaluated?

Like other eliminators, the first step in evaluating a **cong**-expression is to evaluate its target. If the target is neutral, the whole **cong**-expression is neutral, and thus there is no more evaluation.

86 What if the target is not neutral?

If the target is not neutral, then its value has same at the top because same is the only constructor for **=**-expressions.

The value of

(**cong** (same x) f)

is

(same (f x)).

87 Okay, the next step in finding the normal form is to find the value of **cong**'s target.

ind-Nat's target has add1 at the top, so the next step is to use the step.

```
5.  (cong (step-incr=add1 0
              (ind-Nat zero
                mot-incr=add1
                base-incr=add1
                step-incr=add1))
      (+ 1))
```

88 The next **ind-Nat**'s target is zero.

```
6.  (cong (cong base-incr=add1
                (+ 1))
      (+ 1))
7.  (cong (cong (same (add1 zero))
                (+ 1))
      (+ 1))
8.  (cong (same ((+ 1) (add1 zero)))
      (+ 1))
9.  (cong (same (add1 (add1 zero)))
      (+ 1))
10. (same
      ((+ 1)
       (add1
         (add1 zero))))
11. (same
      (add1
        (add1
          (add1 zero))))
```

The Commandment of cong

If x is an X, and f is an

($\to$ X
  Y),

then (cong (same x) f) is the same

(= Y (f x) (f x))

as

(same (f x)).

The interplay between judging sameness and stating equality is at the heart of working with dependent types. This first taste only scratches the surface.

89 But what about my stomach? There's really only space for one sandwich.

Today's your lucky day!

```
(claim sandwich
  (→ Atom
    Atom))
(define sandwich
  (λ (which-sandwich)
    'delicious))
```

90 Oh, what a relief! There *is* just one sandwich:

(same 'delicious)

is a proof that

(*sandwich* 'hoagie),

(*sandwich* 'grinder),

(*sandwich* 'submarine), and

(*sandwich* 'hero),

are all equal.

Enjoy your sandwich, but if you're full, wrap it up for later.

This page makes an excellent sandwich wrapper.

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.†

†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).

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

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

($\rightarrow$ *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=add1_{n-1})
      (replace incr=add1_{n-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-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

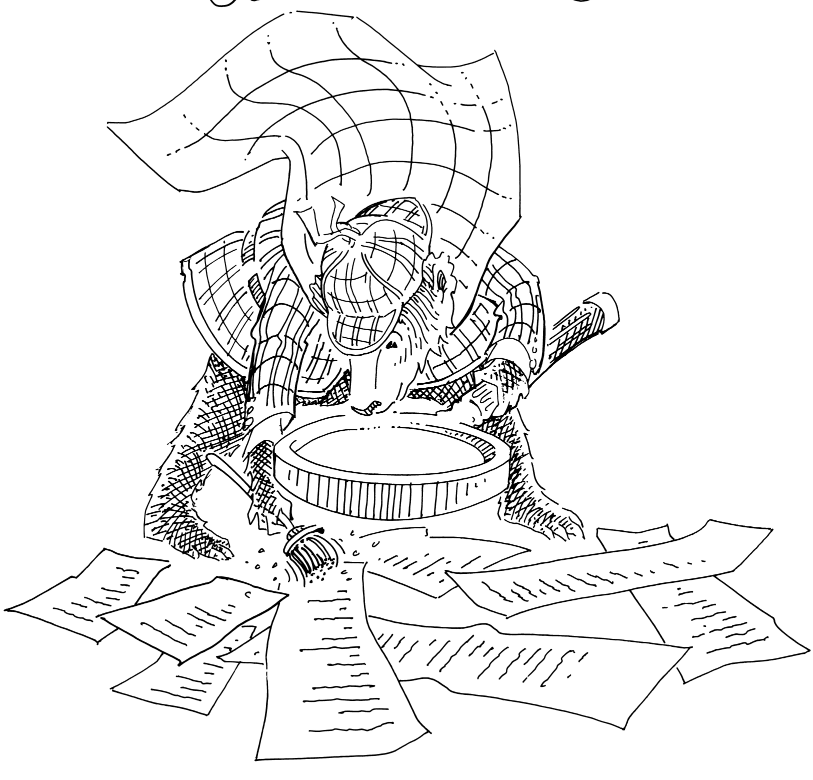
  (same x).

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.

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

6 When is

(Σ ((x A))
D)

a type?

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*.[†]

7 Yes, because Atom is a type, and

(= Atom *bread* 'bagel)

is a type when *bread* is an Atom.

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.

What expression has the type

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

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

Indeed.

Is

(Σ ((*A* $\mathcal{U}$))
  A)

a type?

9 $\mathcal{U}$ is a type, and *A* is certainly a type when *A* is a $\mathcal{U}$.

The Law of Σ

The expression

(**Σ** ((x A))
  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

(**Σ** ((x A))
  D),

then p is the same as

(cons (**car** p) (**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 $\mathcal{U}$))
  A).

Two more expressions with that type are

(cons Atom 'porridge),

and

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

11 Is

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

a

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

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?

---

12 (Pair A D) is a short way of writing

```
(Σ ((x A))
  D)
```

where x is not used in D.

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?

---

13 Like this:

```
(Σ ((ℓ Nat))
  (Vec Atom ℓ)).
```

What values have that type?

---

14 Here are seventeen 'peas:

(cons 17 (***peas*** 17)).

Now give another.

How about a nice breakfast?

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

It's good to start the day off right.

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

Types built with →, 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?

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**.

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**.

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

(Σ ((x A))
  D),

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

30 That is just like (Pair A D).

But **cdr** is slightly different.

If p is a

(Σ ((x A))
  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

(Σ ((ℓ Nat))
  (Vec Atom ℓ)),

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

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

If p is a

(Σ ((ℓ Nat))
  (Vec Atom ℓ)),

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
  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 (**+** ℓ 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.†

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

30 What is an example of such a proof?

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.

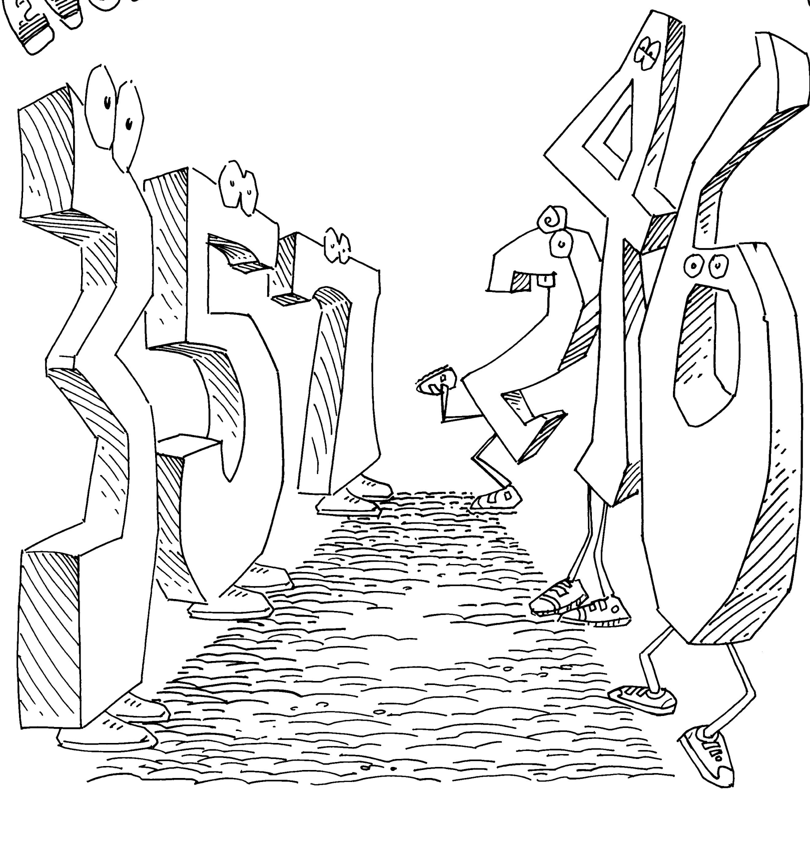
62 If they also reversed lists, then that should have been part of their names!

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))))).
```

25 The expression

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

has that type because

```
(+ 2 n)
```

is the same Nat as

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

That is precisely what is needed to complete the proof.

26 Thanks for the hints.

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

Is two even?

27 Yes, it is.

---

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

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))
```

29 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.

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))
```

30 What is an odd number?

An odd number is not even.

31 Is there a more explicit way to say that?

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

32 How can that description be written as a type?

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

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 →-expression, because the n means "for every n," and if-then statements are translated to →-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)))))
```

50 Now it's time to make that claim true.

What is "haf" of 25?

It is 12 because
  (add1 (***double*** 12))
is the same Nat as 25.

---

51 What is half of 26?

It is 13 because
  (***double*** 13)
is the same Nat as 26.

---

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

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

---

53 Now start the definition, using this "haf."

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)))).
```

---

54 Where did that type come from?

It came from the definition of ***Even*** combined with the Commandment of **cdr**.

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

55 In the box,

(**cdr** o_n)

is an

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

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

```
(= Nat
  (add1 n)
  (double (add1 (car oₙ))))?
```

56 **cong** does the trick, because

```
(double (add1 (car oₙ)))
```

is the same Nat as

```
(add1
  (add1
    (double (car oₙ)))).
```

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

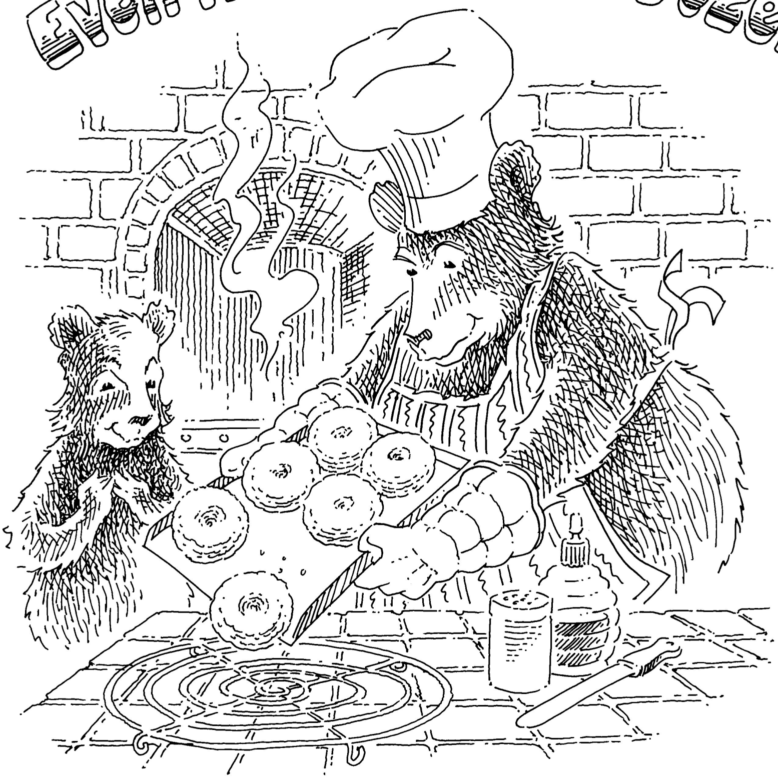
Behold! Ackermann!

```
(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 Haf 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**

```
(→ (Either L R)
   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-or-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))))
```

37

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))))
```

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))))
```

38

13.
```
((λ (o_n-1)
   (left
     (add1-odd→even 1 o_n-1)))
 (add1-even→odd 0 zero-is-even))
```

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 (oₙ)
          (cons (add1 (car oₙ))
            (cong (cdr oₙ) (+ 1))))
        ((λ (eₙ)
            (cons (car eₙ)
              (cong (cdr eₙ) (+ 1))))
          zero-is-even)))
18. (left
      ((λ (oₙ)
          (cons (add1 (car oₙ))
            (cong (cdr oₙ) (+ 1))))
        ((λ (eₙ)
            (cons (car eₙ)
              (cong (cdr eₙ) (+ 1))))
          (cons 0 (same 0)))))
```

What's next?

42 Next, find the **car** and **cdr** of e_n.

```
19. (left
      ((λ (oₙ)
          (cons (add1 (car oₙ))
            (cong (cdr oₙ) (+ 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).

Here's the next step.

```
20. (left
      ((λ (oₙ)
          (cons (add1 (car oₙ))
            (cong (cdr oₙ) (+ 1))))
        (cons 0
          (same 1))))
```

What remains?

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)))
```

What can be learned from this normal form?

44 From the value, we see that 2 is even. The normal form also has the *proof* that 2 is even tucked under left.

Each step in the "same as" chart is the same as the previous step, so the value also contains the proof.

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.[†]

[†] Sometimes called the *unit* type.

4 What is Trivial?

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.

53 This is because there are no entries when the length is zero.

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))))
```

54

Take another look at the definition of ***Fin*** in frame 50. What is another way of writing ***fzero***'s type?

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.

55

In that type, what are (***Fin*** *n*)'s values?

That depends on *n*'s values.

56

This means that a good choice for ***fzero***'s definition is . . .

. . . (***nothing*** (***Fin*** *n*)), even though it is something rather than nothing.

57

Good choice. Now define ***fzero***.

Here it is.

```
(define fzero
  (λ (n)
    (nothing (Fin n))))
```

58 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)))))
```

Why do the two ***Fin***s have different arguments?

59 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?

The other is

(***just*** (***Maybe*** Absurd)
  (***nothing*** Absurd)).

60 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***.

It adds the extra ***just***.

```
(define fadd1
  (λ (n)
    (λ (i-1)
      (just (Fin n) i-1))))
```

61 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)))
```

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 *E*s.

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-refℓ-1)
          (λ (f es)
            (ind-Either f
              (λ (i)
                E)
              (λ (i-1)
                (vec-refℓ-1
                  i-1 (tail es)))
              (λ (triv)
                (head es))))))
      Atom (add1 2)
      (ind-Nat (add1 2) ...))
    (fadd1 3
      (fzero 2))
    menu)
6. (((λ (vec-refℓ-1)
        (λ (f es)
          (ind-Either f
            (λ (i)
              Atom)
            (λ (i-1)
              (vec-refℓ-1
                i-1 (tail es)))
            (λ (triv)
              (head es)))))
      (ind-Nat (add1 2) ...))
    (fadd1 3
      (fzero 2))
    menu)
```

75

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))
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))
```

76

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))
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)))))

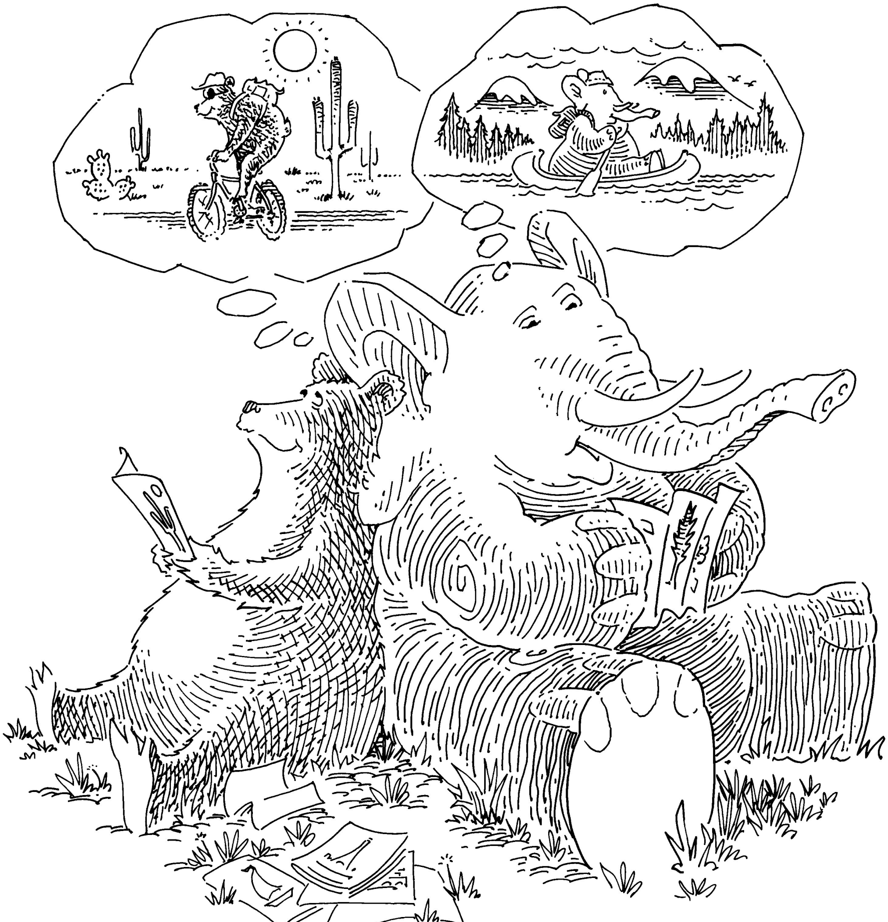
(claim both-left
  (→ Two Two
    Two))
(define both-left
  (λ (a b)
    (ind-Either a
      (λ (c)
        Two)
      (λ (left-sole)
        b)
      (λ (right-sole)
        (right sole)))))

(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)
```

[†] Also known as *the Principle of Explosion* or *ex falso quodlibet*, which means "from false, anything."

6 Why does

```
(→ X
  Absurd)
```

capture the meaning of "not *X*?"

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.

32 Are more things equal than are the same?

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.

33 How is this expressive power useful?

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.

34 Even the Absurd consequences?

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.

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

```
1. (front Atom 2
     (vec:: 'sprinkles
       (vec:: 'chocolate
         (vec:: 'maple vecnil))))?
```

76 The first step is to apply ***front*** to its arguments.

```
2. ((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.

```
3. ((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)))
4. ((λ (j eq)
      'sprinkles)
    2 (same (add1 2)))
5. '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

```
(→ X
  Absurd).
```

What is evidence for an →?

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.

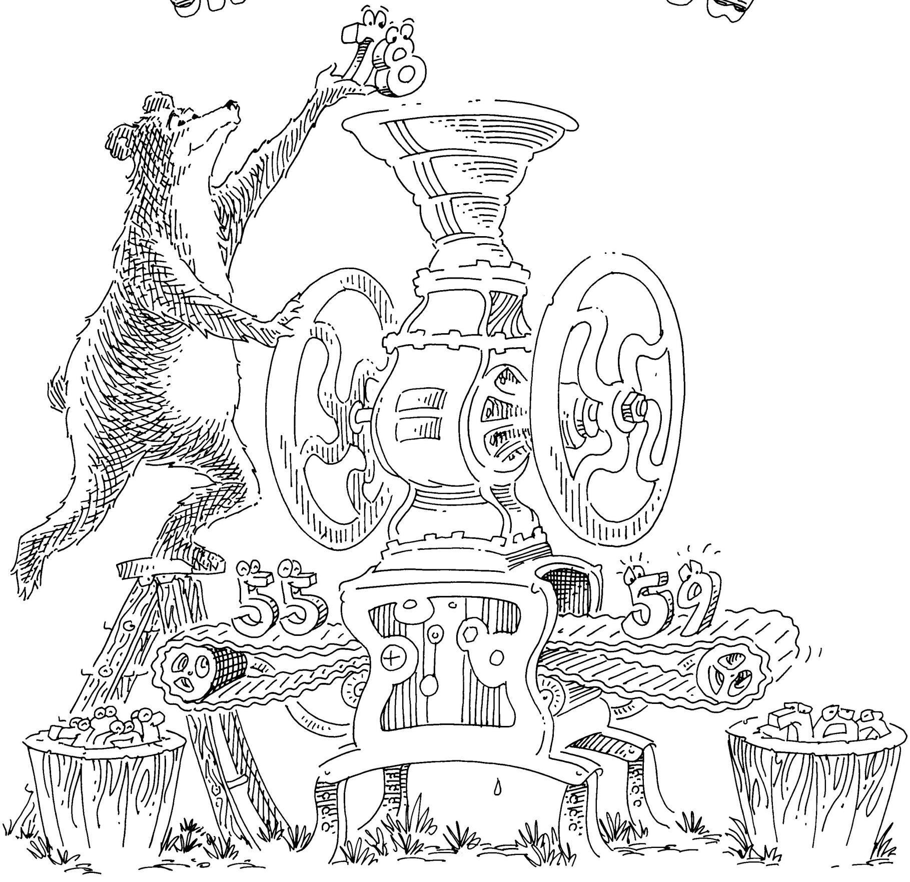
110 It's a good thing there are more donuts.

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 $\textit{zero?}_{n\text{-}1}$ 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.

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

[3] Thanks, Thierry Coquand (1961–).

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 $\leadsto$, 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.
$\Gamma \vdash \textbf{fresh} \leadsto x$	Γ does not bind x.
$\Gamma \vdash x\ \textbf{lookup} \leadsto c_t$	Looking up x in Γ yields the type c_t.
$\Gamma \vdash e_t\ \ \textbf{type} \leadsto c_t$	e_t represents the type c_t.
$\Gamma \vdash c_1 \equiv c_2\ \ \textbf{type}$	c_1 and c_2 are the same type.
$\Gamma \vdash e \in c_t \leadsto c_e$	Checking that e can have type c_t results in c_e.
$\Gamma \vdash e\ \ \textbf{synth} \leadsto (\texttt{the}\ c_t\ c_e)$	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
$\Gamma \vdash \textbf{fresh} \leadsto x$	Γ is a context.
$\Gamma \vdash x\ \textbf{lookup} \leadsto c_t$	Γ is a context.
$\Gamma \vdash e_t\ \ \textbf{type} \leadsto c_t$	Γ is a context.
$\Gamma \vdash c_1 \equiv c_2\ \ \textbf{type}$	Γ is a context, and c_1 and c_2 are both types.
$\Gamma \vdash e \in c_t \leadsto c_e$	Γ is a context and c_t is a type.
$\Gamma \vdash e\ \ \textbf{synth} \leadsto (\texttt{the}\ c_t\ c_e)$	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 $\Gamma \vdash \textbf{fresh} \leadsto x$. 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	car
A	Type of car of a pair	CAR
arg	Argument to a function	argument
Arg	Type of argument to a function	Argument
b	Base	b is for base
B	Type of base	B is for base
d	cdr of a pair	cdr
D	Type of cdr of a pair	CDR
e	Entry in a list or vector	e is for entry
E	Type of entries in a list or vector	ENTRY
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 Port 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 Starboard 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 $\Gamma \vdash x$ **lookup** $\rightsquigarrow c_t$. It has two rules: LookupStop and LookupPop.

$$\frac{}{\Gamma, x : X \vdash x\ \textbf{lookup} \rightsquigarrow X}\ [\textsc{LookupStop}]$$

$$\frac{x \neq y \qquad \Gamma \vdash x\ \textbf{lookup} \rightsquigarrow X}{\Gamma, y : Y \vdash x\ \textbf{lookup} \rightsquigarrow X}\ [\textsc{LookupPop}]$$

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.

$$\frac{\Gamma \vdash x\ \textbf{lookup} \rightsquigarrow X}{\Gamma \vdash x\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ X\ x)}\ [\textsc{Hypothesis}]$$

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.

$$\frac{\Gamma \vdash x\ \textbf{lookup} \leadsto X}{\Gamma \vdash x \equiv x : X}\ [\textsc{HypothesisSame}]$$

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.

$$\frac{}{\Gamma \vdash \mathsf{Atom}\ \ \textbf{type} \leadsto \mathsf{Atom}}\ [\textsc{AtomF}] \qquad \frac{}{\Gamma \vdash \mathsf{Atom} \equiv \mathsf{Atom}\ \ \textbf{type}}\ [\textsc{AtomSame}\text{-}\mathsf{Atom}]$$

$$\frac{}{\Gamma \vdash \texttt{'}\lceil sym \rceil\ \ \textbf{synth} \leadsto (\texttt{the}\ \mathsf{Atom}\ \texttt{'}\lceil sym \rceil)}\ [\textsc{AtomI}]$$

$$\frac{}{\Gamma \vdash \texttt{'}\lceil sym \rceil \equiv \texttt{'}\lceil sym \rceil : \mathsf{Atom}}\ [\textsc{AtomSame-tick}]$$

Pairs

$$\frac{\Gamma \vdash A\ \ \textbf{type} \leadsto A^o \qquad \Gamma, x : A^o \vdash D\ \ \textbf{type} \leadsto D^o}{\Gamma \vdash (\Sigma\ ((x\ A))\ D)\ \ \textbf{type} \leadsto (\Sigma\ ((x\ A^o))\ D^o)}\ [\Sigma\text{F-1}]$$

$$\frac{\begin{array}{l}\Gamma \vdash A\ \ \textbf{type} \leadsto A^o \\ \Gamma, x : A^o \vdash (\Sigma\ ((x_1\ A_1)\ \ \ldots\ \ (x_n\ A_n))\ D)\ \ \textbf{type} \leadsto X\end{array}}{\Gamma \vdash (\Sigma\ ((x\ A)\ (x_1\ A_1)\ \ \ldots\ \ (x_n\ A_n))\ D)\ \ \textbf{type} \leadsto (\Sigma\ ((x\ A^o))\ X)}\ [\Sigma\text{F-2}]$$

$$\frac{\Gamma \vdash A \ \textbf{type} \rightsquigarrow A^o \qquad \Gamma \vdash \textbf{fresh} \rightsquigarrow x \qquad \Gamma, x : A^o \vdash D \ \textbf{type} \rightsquigarrow D^o}{\Gamma \vdash (\mathsf{Pair}\ A\ D) \ \textbf{type} \rightsquigarrow (\Sigma\ ((x\ A^o))\ D^o)} \text{[}\Sigma\text{F-Pair]}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 \ \textbf{type} \qquad \Gamma, x : A_1 \vdash D_1 \equiv D_2 \ \textbf{type}}{\Gamma \vdash (\Sigma\ ((x\ A_1))\ D_1) \equiv (\Sigma\ ((x\ A_2))\ D_2) \ \textbf{type}} \text{[}\Sigma\textsc{Same}\text{-}\Sigma\text{]}$$

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.

$$\frac{\Gamma \vdash a \in A \rightsquigarrow a^o \qquad \Gamma \vdash d \in D[a^o/x] \rightsquigarrow d^o}{\Gamma \vdash (\mathsf{cons}\ a\ d) \in (\Sigma\ ((x\ A))\ D) \rightsquigarrow (\mathsf{cons}\ a^o\ d^o)} \text{[}\Sigma\text{I]}$$

Try reading ΣI aloud.

$$\frac{\Gamma \vdash a_1 \equiv a_2 : A \qquad \Gamma \vdash d_1 \equiv d_2 : D[a_1/x]}{\Gamma \vdash (\mathsf{cons}\ a_1\ d_1) \equiv (\mathsf{cons}\ a_2\ d_2) : (\Sigma\ ((x\ A))\ D)} \text{[}\Sigma\textsc{Same}\text{-}\mathsf{cons}\text{]}$$

$$\frac{\Gamma \vdash pr \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\Sigma\ ((x\ A))\ D)\ pr^o)}{\Gamma \vdash (\mathsf{car}\ pr) \ \textbf{synth} \rightsquigarrow (\texttt{the}\ A\ (\mathsf{car}\ pr^o))} \text{[}\Sigma\text{E-1]}$$

$$\frac{\Gamma \vdash pr_1 \equiv pr_2 : (\Sigma\ ((x\ A))\ D)}{\Gamma \vdash (\mathsf{car}\ pr_1) \equiv (\mathsf{car}\ pr_2) : A} \text{[}\Sigma\textsc{Same}\text{-}\mathsf{car}\text{]}$$

$$\frac{\Gamma \vdash a_1 \equiv a_2 : A \qquad \Gamma, x : A \vdash d \equiv d : D}{\Gamma \vdash (\mathsf{car}\ (\mathsf{cons}\ a_1\ d)) \equiv a_2 : A} \text{[}\Sigma\textsc{Same}\text{-}\iota 1\text{]}$$

$$\frac{\Gamma \vdash pr \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\Sigma\ ((x\ A))\ D)\ pr^o)}{\Gamma \vdash (\mathsf{cdr}\ pr) \ \textbf{synth} \rightsquigarrow (\texttt{the}\ D[(\mathsf{car}\ pr^o)/x]\ (\mathsf{cdr}\ pr^o))} \text{[}\Sigma\text{E-2]}$$

$$\frac{\Gamma \vdash pr_1 \equiv pr_2 : (\Sigma\ ((x\ A))\ D)}{\Gamma \vdash (\mathsf{cdr}\ pr_1) \equiv (\mathsf{cdr}\ pr_2) : D[(\mathsf{car}\ pr_1)/x]} \text{[}\Sigma\textsc{Same}\text{-}\mathsf{cdr}\text{]}$$

$$\frac{\Gamma \vdash a_1 \equiv a_2 : A \qquad \Gamma, x : A \vdash d_1 \equiv d_2 : D}{\Gamma \vdash (\mathsf{cdr}\ (\mathsf{cons}\ a_1\ d_1)) \equiv d_2 : D[a_2/x]} \text{[}\Sigma\textsc{Same}\text{-}\iota 2\text{]}$$

$$\frac{\Gamma \vdash pr_1 \equiv pr_2 : (\Sigma\ ((x\ A))\ D)}{\Gamma \vdash pr_1 \equiv (\mathsf{cons}\ (\mathsf{car}\ pr_2)\ (\mathsf{cdr}\ pr_2)) : (\Sigma\ ((x\ A))\ D)} \text{[}\Sigma\textsc{Same}\text{-}\eta\text{]}$$

Functions

$$\frac{\Gamma \vdash \mathit{Arg}\ \ \textbf{type} \rightsquigarrow \mathit{Arg}^o \qquad \Gamma, x : \mathit{Arg}^o \vdash R\ \ \textbf{type} \rightsquigarrow R^o}{\Gamma \vdash (\Pi\ ((x\ \mathit{Arg}))\ R)\ \ \textbf{type} \rightsquigarrow (\Pi\ ((x\ \mathit{Arg}^o))\ R^o)}\ [\textsc{FunF-1}]$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{Arg}\ \ \textbf{type} \rightsquigarrow \mathit{Arg}^o \\ \Gamma, x : \mathit{Arg}^o \vdash (\Pi\ ((x_1\ \ \mathit{Arg}_1)\ \ \ldots\ \ (x_n\ \ \mathit{Arg}_n))\ R)\ \ \textbf{type} \rightsquigarrow X\end{array}}{\Gamma \vdash (\Pi\ ((x\ \mathit{Arg})\ (x_1\ \ \mathit{Arg}_1)\ \ \ldots\ \ (x_n\ \ \mathit{Arg}_n))\ R)\ \ \textbf{type} \rightsquigarrow (\Pi\ ((x\ \mathit{Arg}^o))\ X)}\ [\textsc{FunF-2}]$$

$$\frac{\Gamma \vdash \mathit{Arg}\ \ \textbf{type} \rightsquigarrow \mathit{Arg}^o \qquad \Gamma \vdash \textbf{fresh} \rightsquigarrow x \qquad \Gamma, x : \mathit{Arg}^o \vdash R\ \ \textbf{type} \rightsquigarrow R^o}{\Gamma \vdash (\rightarrow \mathit{Arg}\ R)\ \ \textbf{type} \rightsquigarrow (\Pi\ ((x\ \mathit{Arg}^o))\ R^o)}\ [\textsc{FunF}{\rightarrow}1]$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{Arg}\ \ \textbf{type} \rightsquigarrow \mathit{Arg}^o \\ \Gamma \vdash \textbf{fresh} \rightsquigarrow x \\ \Gamma, x : \mathit{Arg}^o \vdash (\rightarrow \mathit{Arg}_1\ \ \ldots\ \ \mathit{Arg}_n\ R)\ \ \textbf{type} \rightsquigarrow X\end{array}}{\Gamma \vdash (\rightarrow \mathit{Arg}\ \mathit{Arg}_1\ \ \ldots\ \ \mathit{Arg}_n\ R)\ \ \textbf{type} \rightsquigarrow (\Pi\ ((x\ \mathit{Arg}^o))\ X)}\ [\textsc{FunF}{\rightarrow}2]$$

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 $\mathit{Arg}_1 \ldots \mathit{Arg}_n$ 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 $(\Pi\ ((x\ \mathit{Arg}^o))\ X)$.

$$\frac{\Gamma \vdash \mathit{Arg}_1 \equiv \mathit{Arg}_2\ \ \textbf{type} \qquad \Gamma, x : \mathit{Arg}_1 \vdash R_1 \equiv R_2\ \ \textbf{type}}{\Gamma \vdash (\Pi\ ((x\ \mathit{Arg}_1))\ R_1) \equiv (\Pi\ ((x\ \mathit{Arg}_2))\ R_2)\ \ \textbf{type}}\ [\textsc{FunSame-}\Pi]$$

$$\frac{\Gamma, x : \mathit{Arg} \vdash r \in R \rightsquigarrow r^o}{\Gamma \vdash (\lambda\ (x)\ r) \in (\Pi\ ((x\ \mathit{Arg}))\ R) \rightsquigarrow (\lambda\ (x)\ r^o)}\ [\textsc{FunI-1}]$$

$$\frac{\Gamma, x : \mathit{Arg} \vdash (\lambda\ (y\ z \ldots)\ r) \in R \rightsquigarrow r^o}{\Gamma \vdash (\lambda\ (x\ y\ z \ldots)\ r) \in (\Pi\ ((x\ \mathit{Arg}))\ R) \rightsquigarrow (\lambda\ (x)\ r^o)}\ [\textsc{FunI-2}]$$

$$\frac{\Gamma, x : Arg \vdash r_1 \equiv r_2 : R}{\Gamma \vdash (\mathsf{\lambda}\ (x)\ r_1) \equiv (\mathsf{\lambda}\ (x)\ r_2) : (\mathsf{\Pi}\ ((x\ Arg))\ R)}\ [\textsc{FunSame-}\mathsf{\lambda}]$$

$$\frac{\Gamma \vdash f\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\mathsf{\Pi}\ ((x\ Arg))\ R)\ f^o) \qquad \Gamma \vdash arg \in Arg \rightsquigarrow arg^o}{\Gamma \vdash (f\ \ arg)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ R[arg^o/x]\ (f^o\ \ arg^o))}\ [\textsc{FunE-1}]$$

$$\frac{\begin{array}{l}\Gamma \vdash (f\ \ arg\ \ldots\ arg_{n-1})\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\mathsf{\Pi}\ ((x\ Arg))\ R)\ f^o) \\ \Gamma \vdash arg_n \in Arg \rightsquigarrow arg_n^o\end{array}}{\Gamma \vdash (f\ \ arg\ \ldots\ arg_{n-1}\ \ arg_n)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ R[arg_n^o/x]\ (f^o\ \ arg_n^o))}\ [\textsc{FunE-2}]$$

$$\frac{\Gamma \vdash f_1 \equiv f_2 : (\mathsf{\Pi}\ ((x\ Arg))\ R) \qquad \Gamma \vdash arg_1 \equiv arg_2 : Arg}{\Gamma \vdash (f_1\ \ arg_1) \equiv (f_2\ \ arg_2) : R[arg_1/x]}\ [\textsc{FunSame-}\mathsf{apply}]$$

$$\frac{\Gamma, x : Arg \vdash r_1 \equiv r_2 : R \qquad \Gamma \vdash arg_1 \equiv arg_2 : Arg}{\Gamma \vdash ((\mathsf{\lambda}\ (x)\ r_1)\ \ arg_1) \equiv r_2[arg_2/x] : R[arg_2/x]}\ [\textsc{FunSame-}\beta]$$

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.

$$\frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash f_1 \equiv f_2 : (\mathsf{\Pi}\ ((x\ Arg))\ R)}{\Gamma \vdash f_1 \equiv (\mathsf{\lambda}\ (x)\ (f_2\ x)) : (\mathsf{\Pi}\ ((x\ Arg))\ R)}\ [\textsc{FunSame-}\eta]$$

Natural Numbers

$$\frac{}{\Gamma \vdash \mathsf{Nat}\ \ \textbf{type} \rightsquigarrow \mathsf{Nat}}\ [\textsc{NatF}] \qquad \frac{}{\Gamma \vdash \mathsf{Nat} \equiv \mathsf{Nat}\ \ \textbf{type}}\ [\textsc{NatSame-}\mathsf{Nat}]$$

$$\frac{}{\Gamma \vdash \mathsf{zero}\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathsf{Nat}\ \mathsf{zero})}\ [\textsc{NatI-1}]$$

$$\frac{}{\Gamma \vdash \mathsf{zero} \equiv \mathsf{zero} : \mathsf{Nat}}\ [\textsc{NatSame-}\mathsf{zero}]$$

$$\frac{\Gamma \vdash n \in \textsf{Nat} \rightsquigarrow n^o}{\Gamma \vdash (\textsf{add1}\ n)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \textsf{Nat}\ (\textsf{add1}\ n^o))}\ [\textsc{NatI-2}]$$

In these rules, $\lceil n \rceil$ stands for a literal Scheme natural number.

$$\frac{}{\Gamma \vdash \lceil 0 \rceil\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \textsf{Nat}\ \textsf{zero})}\ [\textsc{NatI-3}]$$

$$\frac{\Gamma \vdash \lceil k \rceil \in \textsf{Nat} \rightsquigarrow n}{\Gamma \vdash \lceil k+1 \rceil\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \textsf{Nat}\ (\textsf{add1}\ n))}\ [\textsc{NatI-4}]$$

$$\frac{\Gamma \vdash n_1 \equiv n_2 : \textsf{Nat}}{\Gamma \vdash (\textsf{add1}\ n_1) \equiv (\textsf{add1}\ n_2) : \textsf{Nat}}\ [\textsc{NatSame-}\textsf{add1}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t \in \textsf{Nat} \rightsquigarrow t^o \\ \Gamma \vdash b\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ B\ b^o) \\ \Gamma \vdash s \in (\Pi\ ((\times\ \textsf{Nat}))\ B) \rightsquigarrow s^o\end{array}}{\Gamma \vdash (\textsf{\textbf{which-Nat}}\ t\ b\ s)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ B\ (\textsf{\textbf{which-Nat}}\ t^o\ (\texttt{the}\ B\ b^o)\ s^o))}\ [\textsc{NatE-1}]$$

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.

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : \textsf{Nat} \\ \Gamma \vdash B_1 \equiv B_2\ \ \textbf{type} \\ \Gamma \vdash b_1 \equiv b_2 : B_1 \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((\times\ \textsf{Nat}))\ B_1)\end{array}}{\Gamma \vdash \begin{array}{c}(\textsf{\textbf{which-Nat}}\ t_1\ (\texttt{the}\ B_1\ b_1)\ s_1) \\ \equiv \\ (\textsf{\textbf{which-Nat}}\ t_2\ (\texttt{the}\ B_2\ b_2)\ s_2)\end{array} : B_1}\ [\textsc{NatSame-}\textsf{\textbf{w-N}}]$$

$$\frac{\Gamma \vdash b_1 \equiv b_2 : B \qquad \Gamma \vdash s \equiv s : (\Pi\ ((\times\ \textsf{Nat}))\ B)}{\Gamma \vdash (\textsf{\textbf{which-Nat}}\ \textsf{zero}\ (\texttt{the}\ B\ b_1)\ s) \equiv b_2 : B}\ [\textsc{NatSame-}\textsf{\textbf{w-N}}\iota 1]$$

$$\frac{\begin{array}{l}\Gamma \vdash n_1 \equiv n_2 : \textsf{Nat} \\ \Gamma \vdash b \equiv b : B \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((\times\ \textsf{Nat}))\ B)\end{array}}{\Gamma \vdash (\textsf{\textbf{which-Nat}}\ (\textsf{add1}\ n_1)\ (\texttt{the}\ B\ b)\ s_1) \equiv (s_2\ n_2) : B}\ [\textsc{NatSame}\text{-}\textsf{\textbf{w-N}}\iota 2]$$

$$\frac{\begin{array}{l}\Gamma \vdash t \in \textsf{Nat} \rightsquigarrow t^o \\ \Gamma \vdash b\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ B\ b^o) \\ \Gamma \vdash s \in (\Pi\ ((\times\ B))\ B) \rightsquigarrow s^o\end{array}}{\Gamma \vdash (\textsf{\textbf{iter-Nat}}\ t\ b\ s)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ B\ (\textsf{\textbf{iter-Nat}}\ t^o\ (\texttt{the}\ B\ b^o)\ s^o))}\ [\textsc{Nat}\text{E-2}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : \textsf{Nat} \\ \Gamma \vdash B_1 \equiv B_2\ \ \textbf{type} \\ \Gamma \vdash b_1 \equiv b_2 : B_1 \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((\times\ B_1))\ B_1)\end{array}}{\Gamma \vdash \begin{array}{c}(\textsf{\textbf{iter-Nat}}\ t_1\ (\texttt{the}\ B_1\ b_1)\ s_1) \\ \equiv \\ (\textsf{\textbf{iter-Nat}}\ t_2\ (\texttt{the}\ B_2\ b_2)\ s_2)\end{array} : B_1}\ [\textsc{NatSame}\text{-}\textsf{\textbf{iter-Nat}}]$$

$$\frac{\begin{array}{l}\Gamma \vdash b_1 \equiv b_2 : B \\ \Gamma \vdash s \equiv s : (\Pi\ ((\times\ B))\ B)\end{array}}{\Gamma \vdash (\textsf{\textbf{iter-Nat}}\ \textsf{zero}\ (\texttt{the}\ B\ b_1)\ s) \equiv b_2 : B}\ [\textsc{NatSame}\text{-}\textsf{\textbf{it-N}}\iota 1]$$

$$\frac{\begin{array}{l}\Gamma \vdash n_1 \equiv n_2 : \textsf{Nat} \\ \Gamma \vdash B_1 \equiv B_2\ \ \textbf{type} \\ \Gamma \vdash b_1 \equiv b_2 : B_1 \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((\times\ B_1))\ B_1)\end{array}}{\Gamma \vdash \begin{array}{c}(\textsf{\textbf{iter-Nat}}\ (\textsf{add1}\ n_1)\ (\texttt{the}\ B_1\ b_1)\ s_1) \\ \equiv \\ (s_2\ (\textsf{\textbf{iter-Nat}}\ n_2\ (\texttt{the}\ B_2\ b_2)\ s_2))\end{array} : B_1}\ [\textsc{NatSame}\text{-}\textsf{\textbf{it-N}}\iota 2]$$

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.

$$\frac{\begin{array}{l}\Gamma \vdash t \in \textsf{Nat} \rightsquigarrow t^o \\ \Gamma \vdash b\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ B\ b^o) \\ \Gamma \vdash s \in (\Pi\ ((n\ \textsf{Nat}))\ (\Pi\ ((\times\ B))\ B)) \rightsquigarrow s^o\end{array}}{\Gamma \vdash (\textsf{\textbf{rec-Nat}}\ t\ b\ s)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ B\ (\textsf{\textbf{rec-Nat}}\ t^o\ (\texttt{the}\ B\ b^o)\ s^o))}\ [\textsc{Nat}\text{E-3}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : \textsf{Nat} \\ \Gamma \vdash B_1 \equiv B_2 \ \ \textbf{type} \\ \Gamma \vdash b_1 \equiv b_2 : B_1 \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((n\ \textsf{Nat}))\ (\Pi\ ((x\ B_1))\ B_1))\end{array}}{\Gamma \vdash \begin{array}{c}(\textbf{rec-Nat}\ t_1\ (\texttt{the}\ B_1\ b_1)\ s_1) \\ \equiv \\ (\textbf{rec-Nat}\ t_2\ (\texttt{the}\ B_2\ b_2)\ s_2)\end{array} : B_1}\ [\textsc{NatSame}\text{-}\textbf{rec-Nat}]$$

$$\frac{\Gamma \vdash b_1 \equiv b_2 : B \qquad \Gamma \vdash s \equiv s : (\Pi\ ((n\ \textsf{Nat}))\ (\Pi\ ((x\ B))\ B))}{\Gamma \vdash (\textbf{rec-Nat}\ \textsf{zero}\ (\texttt{the}\ B\ b_1)\ s) \equiv b_2 : B}\ [\textsc{NatSame}\text{-}\textbf{r-N}\iota 1]$$

$$\frac{\begin{array}{l}\Gamma \vdash n_1 \equiv n_2 : \textsf{Nat} \\ \Gamma \vdash B_1 \equiv B_2 \ \ \textbf{type} \\ \Gamma \vdash b_1 \equiv b_2 : B_1 \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((n\ \textsf{Nat}))\ (\Pi\ ((x\ B_1))\ B_1))\end{array}}{\Gamma \vdash \begin{array}{c}(\textbf{rec-Nat}\ (\textsf{add1}\ n_1)\ (\texttt{the}\ B_1\ b_1)\ s_1) \\ \equiv \\ ((s_2\ n_2)\ (\textbf{rec-Nat}\ n_2\ (\texttt{the}\ B_2\ b_2)\ s_2))\end{array} : B_1}\ [\textsc{NatSame}\text{-}\textbf{r-N}\iota 2]$$

$$\frac{\begin{array}{l}\Gamma \vdash t \in \textsf{Nat} \rightsquigarrow t^o \\ \Gamma \vdash m \in (\Pi\ ((x\ \textsf{Nat}))\ \mathcal{U}) \rightsquigarrow m^o \\ \Gamma \vdash b \in (m^o\ \textsf{zero}) \rightsquigarrow b^o \\ \Gamma \vdash s \in (\Pi\ ((k\ \textsf{Nat}))\ (\Pi\ ((\mathit{almost}\ (m^o\ k)))\ (m^o\ (\textsf{add1}\ k)))) \rightsquigarrow s^o\end{array}}{\Gamma \vdash (\textbf{ind-Nat}\ t\ m\ b\ s)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (m^o\ t^o)\ (\textbf{ind-Nat}\ t^o\ m^o\ b^o\ s^o))}\ [\textsc{NatE-4}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : \textsf{Nat} \\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((x\ \textsf{Nat}))\ \mathcal{U}) \\ \Gamma \vdash b_1 \equiv b_2 : (m_1\ \textsf{zero}) \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((k\ \textsf{Nat})) \\ \qquad\qquad\qquad (\Pi\ ((\mathit{almost}\ (m_1\ k))) \\ \qquad\qquad\qquad\quad (m_1\ (\textsf{add1}\ k))))\end{array}}{\Gamma \vdash \begin{array}{c}(\textbf{ind-Nat}\ t_1\ m_1\ b_1\ s_1) \\ \equiv \\ (\textbf{ind-Nat}\ t_2\ m_2\ b_2\ s_2)\end{array} : (m_1\ t_1)}\ [\textsc{NatSame}\text{-}\textbf{ind-Nat}]$$

$$\frac{\begin{array}{l}\Gamma \vdash m \equiv m : (\Pi\ ((x\ \mathsf{Nat}))\ \mathcal{U}) \\ \Gamma \vdash b_1 \equiv b_2 : (m\ \mathsf{zero}) \\ \Gamma \vdash s \equiv s : (\Pi\ ((k\ \mathsf{Nat})) \\ \qquad\qquad (\Pi\ ((\mathit{almost}\ (m\ k))) \\ \qquad\qquad\quad (m\ (\mathsf{add1}\ k))))\end{array}}{\Gamma \vdash (\textbf{ind-Nat}\ \mathsf{zero}\ m\ b_1\ s) \equiv b_2 : (m\ \mathsf{zero})}\ [\text{NatSame-in-N}\iota 1]$$

$$\frac{\begin{array}{l}\Gamma \vdash n_1 \equiv n_2 : \mathsf{Nat} \\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((x\ \mathsf{Nat}))\ \mathcal{U}) \\ \Gamma \vdash b_1 \equiv b_2 : (m_1\ \mathsf{zero}) \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((k\ \mathsf{Nat})) \\ \qquad\qquad (\Pi\ ((\mathit{almost}\ (m_1\ k))) \\ \qquad\qquad\quad (m_1\ (\mathsf{add1}\ k))))\end{array}}{\Gamma \vdash \begin{array}{c}(\textbf{ind-Nat}\ (\mathsf{add1}\ n_1)\ m_1\ b_1\ s_1) \\ \equiv \\ ((s_2\ n_2)\ (\textbf{ind-Nat}\ n_2\ m_2\ b_2\ s_2))\end{array} : (m_1\ (\mathsf{add1}\ n_1))}\ [\text{NatSame-in-N}\iota 2]$$

Lists

$$\frac{\Gamma \vdash E\ \ \textbf{type} \rightsquigarrow E^o}{\Gamma \vdash (\mathsf{List}\ E)\ \ \textbf{type} \rightsquigarrow (\mathsf{List}\ E^o)}\ [\text{ListF}]$$

$$\frac{\Gamma \vdash E_1 \equiv E_2\ \ \textbf{type}}{\Gamma \vdash (\mathsf{List}\ E_1) \equiv (\mathsf{List}\ E_2)\ \ \textbf{type}}\ [\text{ListSame-List}]$$

$$\frac{}{\Gamma \vdash \mathsf{nil} \in (\mathsf{List}\ E) \rightsquigarrow \mathsf{nil}}\ [\text{ListI-1}]$$

$$\frac{}{\Gamma \vdash \mathsf{nil} \equiv \mathsf{nil} : (\mathsf{List}\ E)}\ [\text{ListSame-nil}]$$

$$\frac{\Gamma \vdash e\ \ \textbf{synth} \rightsquigarrow (\mathtt{the}\ E\ e^o) \qquad \Gamma \vdash es \in (\mathsf{List}\ E) \rightsquigarrow es^o}{\Gamma \vdash (::\ e\ es)\ \ \textbf{synth} \rightsquigarrow (\mathtt{the}\ (\mathsf{List}\ E)\ (::\ e^o\ es^o))}\ [\text{ListI-2}]$$

$$\frac{\Gamma \vdash e_1 \equiv e_2 : E \qquad \Gamma \vdash es_1 \equiv es_2 : (\mathsf{List}\ E)}{\Gamma \vdash (::\ e_1\ es_1) \equiv (::\ e_2\ es_2) : (\mathsf{List}\ E)}\ [\text{ListSame-::}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textsf{List}\ E)\ t^o) \\ \Gamma \vdash b\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ B\ b^o) \\ \Gamma \vdash s \in (\Pi\ ((\textit{x}\ E))\ (\Pi\ ((\textit{xs}\ (\textsf{List}\ E)))\ (\Pi\ ((\textit{almost}\ B))\ B))) \rightsquigarrow s^o\end{array}}{\Gamma \vdash (\textsf{\textbf{rec-List}}\ t\ b\ s)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ B\ (\textsf{\textbf{rec-List}}\ t^o\ (\texttt{the}\ B\ b^o)\ s^o))}\ [\textsc{ListE-1}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : (\textsf{List}\ E) \\ \Gamma \vdash B_1 \equiv B_2\ \ \textbf{type} \\ \Gamma \vdash b_1 \equiv b_2 : B_1 \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((\textit{x}\ E)) \\ \qquad\qquad\qquad (\Pi\ ((\textit{xs}\ (\textsf{List}\ E))) \\ \qquad\qquad\qquad\quad (\Pi\ ((\textit{almost}\ B_1)) \\ \qquad\qquad\qquad\qquad B_1)))\end{array}}{\Gamma \vdash (\textsf{\textbf{rec-List}}\ t_1\ (\texttt{the}\ B_1\ b_1)\ s_1) \equiv (\textsf{\textbf{rec-List}}\ t_2\ (\texttt{the}\ B_2\ b_2)\ s_2) : B_1}\ [\textsc{ListSame}\text{-}\textsf{\textbf{rec-List}}]$$

$$\frac{\begin{array}{l}\Gamma \vdash \textsf{nil} \equiv \textsf{nil} : (\textsf{List}\ E) \\ \Gamma \vdash b_1 \equiv b_2 : B \\ \Gamma \vdash s \equiv s : (\Pi\ ((\textit{x}\ E)) \\ \qquad\qquad\qquad (\Pi\ ((\textit{xs}\ (\textsf{List}\ E))) \\ \qquad\qquad\qquad\quad (\Pi\ ((\textit{almost}\ B)) \\ \qquad\qquad\qquad\qquad B)))\end{array}}{\Gamma \vdash (\textsf{\textbf{rec-List}}\ \textsf{nil}\ (\texttt{the}\ B\ b_1)\ s_1) \equiv b_2 : B}\ [\textsc{ListSame}\text{-}\textsf{\textbf{r-L}}\iota 1]$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \equiv e_2 : E \\ \Gamma \vdash es_1 \equiv es_2 : (\textsf{List}\ E) \\ \Gamma \vdash B_1 \equiv B_2\ \ \textbf{type} \\ \Gamma \vdash b_1 \equiv b_2 : B_1 \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((\textit{x}\ E)) \\ \qquad\qquad\qquad (\Pi\ ((\textit{xs}\ (\textsf{List}\ E))) \\ \qquad\qquad\qquad\quad (\Pi\ ((\textit{almost}\ B_1)) \\ \qquad\qquad\qquad\qquad B_1)))\end{array}}{\Gamma \vdash \begin{array}{c}(\textsf{\textbf{rec-List}}\ (\textsf{\textbf{::}}\ e_1\ \ es_1)\ (\texttt{the}\ B_1\ b_1)\ s_1) \\ \equiv \\ (((s_2\ e_2)\ es_2)\ (\textsf{\textbf{rec-List}}\ es_2\ (\texttt{the}\ B_2\ b_2)\ s_2))\end{array} : B_1}\ [\textsc{ListSame}\text{-}\textsf{\textbf{r-L}}\iota 2]$$

$$\frac{\begin{array}{l}\Gamma \vdash t\ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\mathsf{List}\ E)\ t^o)\\ \Gamma \vdash m \in (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E)))\ \mathcal{U}) \rightsquigarrow m^o\\ \Gamma \vdash b \in (m^o\ \mathsf{nil}) \rightsquigarrow b^o\\ \Gamma \vdash s \in (\Pi\ ((x\ E))\\ \qquad\qquad (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E)))\\ \qquad\qquad\quad (\Pi\ ((\mathit{almost}\ (m^o\ \mathit{xs})))\\ \qquad\qquad\qquad (m^o\ (::\ x\ \mathit{xs}))))) \quad \rightsquigarrow s^o\end{array}}{\Gamma \vdash (\textsf{ind-List}\ t\ m\ b\ s)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (m^o\ t^o)\ (\textsf{ind-List}\ t^o\ m^o\ b^o\ s^o))}\ [\text{ListE-2}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : (\mathsf{List}\ E)\\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E)))\ \mathcal{U})\\ \Gamma \vdash b_1 \equiv b_2 : (m_1\ \mathsf{nil})\\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((x\ E))\\ \qquad\qquad (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E)))\\ \qquad\qquad\quad (\Pi\ ((\mathit{almost}\ (m_1\ \mathit{xs})))\\ \qquad\qquad\qquad (m_1\ (::\ x\ \mathit{xs})))))\end{array}}{\Gamma \vdash (\textsf{ind-List}\ t_1\ m_1\ b_1\ s_1) \equiv (\textsf{ind-List}\ t_2\ m_2\ b_2\ s_2) : (m_1\ t_1)}\ [\text{ListSame-}\textsf{ind-List}]$$

$$\frac{\begin{array}{l}\Gamma \vdash m \equiv m : (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E)))\ \mathcal{U})\\ \Gamma \vdash b_1 \equiv b_2 : (m\ \mathsf{nil})\\ \Gamma \vdash s \equiv s : (\Pi\ ((x\ E))\\ \qquad\qquad (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E)))\\ \qquad\qquad\quad (\Pi\ ((\mathit{almost}\ (m\ \mathit{xs})))\\ \qquad\qquad\qquad (m\ (::\ x\ \mathit{xs})))))\end{array}}{\Gamma \vdash (\textsf{ind-List}\ \mathsf{nil}\ m\ b_1\ s) \equiv b_2 : (m\ \mathsf{nil})}\ [\text{ListSame-}\textsf{i-L}\iota 1]$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \equiv e_2 : E\\ \Gamma \vdash es_1 \equiv es_2 : (\mathsf{List}\ E)\\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E)))\ \mathcal{U})\\ \Gamma \vdash b_1 \equiv b_2 : (m_1\ \mathsf{nil})\\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((x\ E))\\ \qquad\qquad (\Pi\ ((\mathit{xs}\ (\mathsf{List}\ E)))\\ \qquad\qquad\quad (\Pi\ ((\mathit{almost}\ (m_1\ \mathit{xs})))\\ \qquad\qquad\qquad (m_1\ (::\ x\ \mathit{xs})))))\end{array}}{\Gamma \vdash \begin{array}{c}(\textsf{ind-List}\ (::\ e_1\ es_1)\ m_1\ b_1\ s_1)\\ \equiv\\ (((s_2\ e_2)\ es_2)\ (\textsf{ind-List}\ es_2\ m_2\ b_2\ s_2))\end{array} : (m_1\ (::\ e_1\ es_1))}\ [\text{ListSame-}\textsf{i-L}\iota 2]$$

Vectors

$$\frac{\Gamma \vdash E\ \textbf{type} \rightsquigarrow E^o \qquad \Gamma \vdash \ell \in \mathsf{Nat} \rightsquigarrow \ell^o}{\Gamma \vdash (\mathsf{Vec}\ E\ \ell)\ \ \textbf{type} \rightsquigarrow (\mathsf{Vec}\ E^o\ \ell^o)}\ [\textsc{VecF}]$$

$$\frac{\Gamma \vdash E_1 \equiv E_2\ \ \textbf{type} \qquad \Gamma \vdash \ell_1 \equiv \ell_2 : \mathsf{Nat}}{\Gamma \vdash (\mathsf{Vec}\ E_1\ \ell_1) \equiv (\mathsf{Vec}\ E_2\ \ell_2)\ \ \textbf{type}}\ [\textsc{VecSame}\text{-}\mathsf{Vec}]$$

$$\frac{}{\Gamma \vdash \mathsf{vecnil} \in (\mathsf{Vec}\ E\ \mathsf{zero}) \rightsquigarrow \mathsf{vecnil}}\ [\textsc{VecI-1}]$$

$$\frac{}{\Gamma \vdash \mathsf{vecnil} \equiv \mathsf{vecnil} : (\mathsf{Vec}\ E\ \mathsf{zero})}\ [\textsc{VecSame}\text{-}\mathsf{vecnil}]$$

$$\frac{\Gamma \vdash e \in E \rightsquigarrow e^o \qquad \Gamma \vdash es \in (\mathsf{Vec}\ E\ \ell) \rightsquigarrow es^o}{\Gamma \vdash (\mathsf{vec{::}}\ e\ \ es) \in (\mathsf{Vec}\ E\ (\mathsf{add1}\ \ell)) \rightsquigarrow (\mathsf{vec{::}}\ e^o\ \ es^o)}\ [\textsc{VecI-2}]$$

$$\frac{\Gamma \vdash e_1 \equiv e_2 : E \qquad \Gamma \vdash es_1 \equiv es_2 : (\mathsf{Vec}\ E\ \ell)}{\Gamma \vdash (\mathsf{vec{::}}\ e_1\ \ es_1) \equiv (\mathsf{vec{::}}\ e_2\ \ es_2) : (\mathsf{Vec}\ E\ (\mathsf{add1}\ \ell))}\ [\textsc{VecSame}\text{-}\mathsf{vec{::}}]$$

$$\frac{\Gamma \vdash t\ \ \textbf{synth} \rightsquigarrow (\mathtt{the}\ (\mathsf{Vec}\ E\ (\mathsf{add1}\ \ell))\ t^o)}{\Gamma \vdash (\textsf{\textbf{head}}\ t)\ \ \textbf{synth} \rightsquigarrow (\mathtt{the}\ E\ (\textsf{\textbf{head}}\ t^o))}\ [\textsc{VecE-1}]$$

$$\frac{\Gamma \vdash es_1 \equiv es_2 : (\mathsf{Vec}\ E\ (\mathsf{add1}\ \ell))}{\Gamma \vdash (\textsf{\textbf{head}}\ es_1) \equiv (\textsf{\textbf{head}}\ es_2) : E}\ [\textsc{VecSame}\text{-}\textsf{\textbf{head}}]$$

$$\frac{\Gamma \vdash e_1 \equiv e_2 : E \qquad \Gamma \vdash es \equiv es : (\mathsf{Vec}\ E\ \ell)}{\Gamma \vdash (\textsf{\textbf{head}}\ (\mathsf{vec{::}}\ e_1\ \ es)) \equiv e_2 : E}\ [\textsc{VecSame}\text{-}\textsf{\textbf{h}}\iota]$$

$$\frac{\Gamma \vdash t\ \ \textbf{synth} \rightsquigarrow (\mathtt{the}\ (\mathsf{Vec}\ E\ (\mathsf{add1}\ \ell))\ t^o)}{\Gamma \vdash (\textsf{\textbf{tail}}\ t)\ \ \textbf{synth} \rightsquigarrow (\mathtt{the}\ (\mathsf{Vec}\ E\ \ell)\ (\textsf{\textbf{tail}}\ t^o))}\ [\textsc{VecE-2}]$$

$$\frac{\Gamma \vdash es_1 \equiv es_2 : (\mathsf{Vec}\ E\ (\mathsf{add1}\ \ell))}{\Gamma \vdash (\textsf{\textbf{tail}}\ es_1) \equiv (\textsf{\textbf{tail}}\ es_2) : (\mathsf{Vec}\ E\ \ell)}\ [\textsc{VecSame}\text{-}\textsf{\textbf{tail}}]$$

$$\frac{\Gamma \vdash e \equiv e : E \qquad \Gamma \vdash es_1 \equiv es_2 : (\mathsf{Vec}\ E\ \ell)}{\Gamma \vdash (\textsf{\textbf{tail}}\ (\mathsf{vec{::}}\ e\ \ es_1)) \equiv es_2 : (\mathsf{Vec}\ E\ \ell)}\ [\textsc{VecSame}\text{-}\textsf{\textbf{t}}\iota]$$

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.

$$
\frac{\begin{array}{l}
\Gamma \vdash \ell \in \textsf{Nat} \rightsquigarrow \ell^o \\
\Gamma \vdash t\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textsf{Vec}\ E\ n)\ t^o) \\
\Gamma \vdash \ell^o \equiv n : \textsf{Nat} \\
\Gamma \vdash m \in (\Pi\ ((\textsf{\textit{k}}\ \textsf{Nat}))\ (\Pi\ ((\textsf{\textit{es}}\ (\textsf{Vec}\ E\ \textsf{\textit{k}})))\ \mathcal{U})) \rightsquigarrow m^o \\
\Gamma \vdash b \in ((m^o\ \textsf{zero})\ \textsf{vecnil}) \rightsquigarrow b^o \\
\Gamma \vdash s \in (\Pi\ ((\textsf{\textit{k}}\ \textsf{Nat})) \\
\qquad\qquad (\Pi\ ((\textsf{\textit{e}}\ E)) \\
\qquad\qquad\quad (\Pi\ ((\textsf{\textit{es}}\ (\textsf{Vec}\ E\ \textsf{\textit{k}}))) \\
\qquad\qquad\qquad (\Pi\ ((\textsf{\textit{almost}}\ ((m^o\ \textsf{\textit{k}})\ \textsf{\textit{es}}))) \\
\qquad\qquad\qquad\quad ((m^o\ (\textsf{add1}\ \textsf{\textit{k}}))\ (\textsf{vec::}\ \textsf{\textit{e}}\ \textsf{\textit{es}})))))) \rightsquigarrow s^o
\end{array}}{\Gamma \vdash (\textbf{\textsf{ind-Vec}}\ \ell\ t\ m\ b\ s)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ ((m^o\ \ell^o)\ t^o)\ (\textbf{\textsf{ind-Vec}}\ \ell^o\ t^o\ m^o\ b^o\ s^o))}\ [\textsc{VecE-3}]
$$

$$
\frac{\begin{array}{l}
\Gamma \vdash \ell_1 \equiv \ell_2 : \textsf{Nat} \\
\Gamma \vdash t_1 \equiv t_2 : (\textsf{Vec}\ E\ \ell_1) \\
\Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((\textsf{\textit{k}}\ \textsf{Nat}))\ (\Pi\ ((\textsf{\textit{x}}\ (\textsf{Vec}\ E\ \textsf{\textit{k}})))\ \mathcal{U})) \\
\Gamma \vdash b_1 \equiv b_2 : ((m_1\ \textsf{zero})\ \textsf{vecnil}) \\
\Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((\textsf{\textit{k}}\ \textsf{Nat})) \\
\qquad\qquad (\Pi\ ((\textsf{\textit{e}}\ E)) \\
\qquad\qquad\quad (\Pi\ ((\textsf{\textit{es}}\ (\textsf{Vec}\ E\ \textsf{\textit{k}}))) \\
\qquad\qquad\qquad (\Pi\ ((\textsf{\textit{almost}}\ ((m_1\ \textsf{\textit{k}})\ \textsf{\textit{es}}))) \\
\qquad\qquad\qquad\quad ((m_1\ (\textsf{add1}\ \textsf{\textit{k}}))\ (\textsf{vec::}\ \textsf{\textit{e}}\ \textsf{\textit{es}}))))))
\end{array}}{\Gamma \vdash \begin{array}{c} (\textbf{\textsf{ind-Vec}}\ \ell_1\ t_1\ m_1\ b_1\ s_1) \\ \equiv \\ (\textbf{\textsf{ind-Vec}}\ \ell_2\ t_2\ m_2\ b_2\ s_2) \end{array} : ((m_1\ \ell_1)\ t_1)}\ [\textsc{VecSame}\textbf{\textsf{-ind-Vec}}]
$$

$$
\frac{\begin{array}{l}
\Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((\textsf{\textit{k}}\ \textsf{Nat}))\ (\Pi\ ((\textsf{\textit{x}}\ (\textsf{Vec}\ E\ \textsf{\textit{k}})))\ \mathcal{U})) \\
\Gamma \vdash b_1 \equiv b_2 : ((m_1\ \textsf{zero})\ \textsf{vecnil}) \\
\Gamma \vdash s \equiv s : (\Pi\ ((\textsf{\textit{k}}\ \textsf{Nat})) \\
\qquad\qquad (\Pi\ ((\textsf{\textit{e}}\ E)) \\
\qquad\qquad\quad (\Pi\ ((\textsf{\textit{es}}\ (\textsf{Vec}\ E\ \textsf{\textit{k}}))) \\
\qquad\qquad\qquad (\Pi\ ((\textsf{\textit{almost}}\ ((m_1\ \textsf{\textit{k}})\ \textsf{\textit{es}}))) \\
\qquad\qquad\qquad\quad ((m_1\ (\textsf{add1}\ \textsf{\textit{k}}))\ (\textsf{vec::}\ \textsf{\textit{e}}\ \textsf{\textit{es}}))))))
\end{array}}{\Gamma \vdash (\textbf{\textsf{ind-Vec}}\ \textsf{zero}\ \textsf{vecnil}\ m_1\ b_1\ s) \equiv b_2 : ((m_2\ \textsf{zero})\ \textsf{vecnil})}\ [\textsc{VecSame}\textbf{\textsf{-i-V}}\iota 1]
$$

$$\frac{\begin{array}{l}\Gamma \vdash \ell_1 \equiv \ell_2 : \textsf{Nat} \\ \Gamma \vdash e_1 \equiv e_2 : E \\ \Gamma \vdash es_1 \equiv es_2 : (\textsf{Vec}\ E\ \ell_1) \\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((k\ \textsf{Nat}))\ (\Pi\ ((x\ (\textsf{Vec}\ E\ k)))\ \mathcal{U})) \\ \Gamma \vdash b_1 \equiv b_2 : ((m_1\ \textsf{zero})\ \textsf{vecnil}) \\ \Gamma \vdash s_1 \equiv s_2 : (\Pi\ ((k\ \textsf{Nat})) \\ \qquad\qquad (\Pi\ ((e\ E)) \\ \qquad\qquad\quad (\Pi\ ((es\ (\textsf{Vec}\ E\ k))) \\ \qquad\qquad\qquad (\Pi\ ((almost\ ((m_1\ k)\ es))) \\ \qquad\qquad\qquad\quad ((m_1\ (\textsf{add1}\ k))\ (\textsf{vec::}\ e\ es)))))) \end{array}}{\Gamma \vdash \begin{array}{c}(\textbf{ind-Vec}\ (\textsf{add1}\ \ell_1)\ (\textsf{vec::}\ e_1\ es_1)\ m_1\ b_1\ s_1) \\ \equiv \\ ((((s_2\ \ell_2)\ e_2)\ es_2) \\ (\textbf{ind-Vec}\ \ell_2\ es_2\ m_2\ b_2\ s_2))\end{array} : \begin{array}{c}((m_2\ (\textsf{add1}\ \ell_1)) \\ (\textsf{vec::}\ e_1\ es_1))\end{array}}\ [\text{VecSame-i-V}\iota 2]$$

Equality

$$\frac{\Gamma \vdash X\ \textbf{type} \rightsquigarrow X^o \qquad \Gamma \vdash \mathit{from} \in X^o \rightsquigarrow \mathit{from}^o \qquad \Gamma \vdash \mathit{to} \in X^o \rightsquigarrow \mathit{to}^o}{\Gamma \vdash (\textbf{=}\ X\ \mathit{from}\ \mathit{to})\ \textbf{type} \rightsquigarrow (\textbf{=}\ X^o\ \mathit{from}^o\ \mathit{to}^o)}\ [\text{EqF}]$$

$$\frac{\Gamma \vdash X_1 \equiv X_2\ \textbf{type} \qquad \Gamma \vdash \mathit{from}_1 \equiv \mathit{from}_2 : X_1 \qquad \Gamma \vdash \mathit{to}_1 \equiv \mathit{to}_2 : X_1}{\Gamma \vdash (\textbf{=}\ X_1\ \mathit{from}_1\ \mathit{to}_1) \equiv (\textbf{=}\ X_2\ \mathit{from}_2\ \mathit{to}_2)\ \textbf{type}}\ [\text{EqSame-=}]$$

$$\frac{\Gamma \vdash \mathit{mid} \in X \rightsquigarrow \mathit{mid}^o \qquad \Gamma \vdash \mathit{from} \equiv \mathit{mid}^o : X \qquad \Gamma \vdash \mathit{mid}^o \equiv \mathit{to} : X}{\Gamma \vdash (\textsf{same}\ \mathit{mid}) \in (\textbf{=}\ X\ \mathit{from}\ \mathit{to}) \rightsquigarrow (\textsf{same}\ \mathit{mid}^o)}\ [\text{EqI}]$$

$$\frac{\Gamma \vdash \mathit{from} \equiv \mathit{to} : X}{\Gamma \vdash (\textsf{same}\ \mathit{from}) \equiv (\textsf{same}\ \mathit{to}) : (\textbf{=}\ X\ \mathit{from}\ \mathit{from})}\ [\text{EqSame-same}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t\ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textbf{=}\ X\ \mathit{from}\ \mathit{to})\ t^o) \\ \Gamma \vdash m \in (\Pi\ ((x\ X))\ \mathcal{U}) \rightsquigarrow m^o \\ \Gamma \vdash b \in (m^o\ \mathit{from}) \rightsquigarrow b^o\end{array}}{\Gamma \vdash (\textbf{replace}\ t\ m\ b)\ \textbf{synth} \rightsquigarrow (\texttt{the}\ (m^o\ \mathit{to})\ (\textbf{replace}\ t^o\ m^o\ b^o))}\ [\text{EqE-1}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : (\textsf{=}\ X\ \mathit{from}\ \mathit{to}) \\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((\times\ X))\ \mathcal{U}) \\ \Gamma \vdash b_1 \equiv b_2 : (m_1\ \mathit{from})\end{array}}{\Gamma \vdash (\textsf{replace}\ t_1\ m_1\ b_1) \equiv (\textsf{replace}\ t_2\ m_2\ b_2) : (m_1\ \mathit{to})}\ [\text{EqSame-}\textsf{replace}]$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{expr} \equiv \mathit{expr} : X \\ \Gamma \vdash m \equiv m : (\Pi\ ((\times\ X))\ \mathcal{U}) \\ \Gamma \vdash b_1 \equiv b_2 : (m\ \mathit{expr})\end{array}}{\Gamma \vdash (\textsf{replace}\ (\textsf{same}\ \mathit{expr})\ m\ b_1) \equiv b_2 : (m\ \mathit{expr})}\ [\text{EqSame-}\textsf{r}\iota]$$

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.

$$\frac{\begin{array}{l}\Gamma \vdash t\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textsf{=}\ X_1\ \mathit{from}\ \mathit{to})\ t^o) \\ \Gamma \vdash f\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\Pi\ ((\times\ X_2))\ Y)\ f^o) \\ \Gamma \vdash X_1 \equiv X_2\ \ \textbf{type}\end{array}}{\Gamma \vdash (\textsf{cong}\ t\ f)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textsf{=}\ Y\ (f^o\ \mathit{from})\ (f^o\ \mathit{to}))\ (\textsf{cong}\ X_1\ t^o\ f^o))}\ [\text{EqE-2}]$$

$$\frac{\begin{array}{l}\Gamma \vdash X_1 \equiv X_2\ \ \textbf{type} \\ \Gamma \vdash f_1 \equiv f_2 : (\Pi\ ((\times\ X_1))\ Y) \\ \Gamma \vdash t_1 \equiv t_2 : (\textsf{=}\ X_1\ \mathit{from}\ \mathit{to})\end{array}}{\Gamma \vdash (\textsf{cong}\ X_1\ t_1\ f_1) \equiv (\textsf{cong}\ X_2\ t_2\ f_2) : (\textsf{=}\ Y\ (f_1\ \mathit{from})\ (f_1\ \mathit{to}))}\ [\text{EqSame-}\textsf{cong}]$$

$$\frac{\Gamma \vdash \mathit{expr}_1 \equiv \mathit{expr}_2 : X \qquad \Gamma \vdash f_1 \equiv f_2 : (\Pi\ ((\times\ X))\ Y)}{\Gamma \vdash \begin{array}{c}(\textsf{cong}\ X\ (\textsf{same}\ \mathit{expr}_1)\ f_1) \\ \equiv \\ (\textsf{same}\ (f_2\ \mathit{expr}_2))\end{array} : (\textsf{=}\ X\ (f_1\ \mathit{expr}_1)\ (f_1\ \mathit{expr}_1))}\ [\text{EqSame-}\textsf{c}\iota]$$

$$\frac{\Gamma \vdash t\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textsf{=}\ X\ \mathit{from}\ \mathit{to})\ t^o)}{\Gamma \vdash (\textsf{symm}\ t)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textsf{=}\ X\ \mathit{to}\ \mathit{from})\ (\textsf{symm}\ t^o))}\ [\text{EqE-3}]$$

$$\frac{\Gamma \vdash t_1 \equiv t_2 : (\textsf{=}\ X\ \mathit{from}\ \mathit{to})}{\Gamma \vdash (\textsf{symm}\ t_1) \equiv (\textsf{symm}\ t_2) : (\textsf{=}\ X\ \mathit{to}\ \mathit{from})}\ [\text{EqSame-}\textsf{symm}]$$

$$\frac{\Gamma \vdash \mathit{expr}_1 \equiv \mathit{expr}_2 : X}{\Gamma \vdash (\textsf{\textbf{symm}}\ (\textsf{same}\ \mathit{expr}_1)) \equiv (\textsf{same}\ \mathit{expr}_2) : (\textsf{\textbf{=}}\ X\ \mathit{expr}_1\ \mathit{expr}_1)}\ [\text{EqSame-s}\iota]$$

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.

$$\frac{\begin{array}{l}\Gamma \vdash t_1\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textsf{\textbf{=}}\ X\ \mathit{from}\ \mathit{mid}_1)\ t_1^o) \\ \Gamma \vdash t_2\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textsf{\textbf{=}}\ Y\ \mathit{mid}_2\ \mathit{to})\ t_2^o) \\ \Gamma \vdash X \equiv Y\ \ \textbf{type} \\ \Gamma \vdash \mathit{mid}_1 \equiv \mathit{mid}_2 : X\end{array}}{\Gamma \vdash (\textsf{\textbf{trans}}\ t_1\ t_2)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textsf{\textbf{=}}\ X\ \mathit{from}\ \mathit{to})\ (\textsf{\textbf{trans}}\ t_1^o\ t_2^o))}\ [\text{EqE-4}]$$

$$\frac{\Gamma \vdash t_1 \equiv t_2 : (\textsf{\textbf{=}}\ X\ \mathit{from}\ \mathit{mid}) \qquad \Gamma \vdash t_3 \equiv t_4 : (\textsf{\textbf{=}}\ X\ \mathit{mid}\ \mathit{to})}{\Gamma \vdash (\textsf{\textbf{trans}}\ t_1\ t_3) \equiv (\textsf{\textbf{trans}}\ t_2\ t_4) : (\textsf{\textbf{=}}\ X\ \mathit{from}\ \mathit{to})}\ [\text{EqSame-}\textsf{\textbf{trans}}]$$

$$\frac{\Gamma \vdash \mathit{expr}_1 \equiv \mathit{expr}_2 : X \qquad \Gamma \vdash \mathit{expr}_2 \equiv \mathit{expr}_3 : X}{\Gamma \vdash \begin{array}{c}(\textsf{\textbf{trans}}\ (\textsf{same}\ \mathit{expr}_1)\ (\textsf{same}\ \mathit{expr}_2)) \\ \equiv \\ (\textsf{same}\ \mathit{expr}_3)\end{array} : (\textsf{\textbf{=}}\ X\ \mathit{expr}_3\ \mathit{expr}_3)}\ [\text{EqSame-t}\iota]$$

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*.

$$\frac{\begin{array}{l}\Gamma \vdash t\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textsf{\textbf{=}}\ X\ \mathit{from}\ \mathit{to})\ t^o) \\ \Gamma \vdash m \in (\Pi\ ((x\ X))\ (\Pi\ ((t\ (\textsf{\textbf{=}}\ X\ \mathit{from}\ x)))\ \mathcal{U})) \rightsquigarrow m^o \\ \Gamma \vdash b \in ((m^o\ \mathit{from})\ (\textsf{same}\ \mathit{from})) \rightsquigarrow b^o\end{array}}{\Gamma \vdash (\textsf{\textbf{ind-=}}\ t\ m\ b)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ ((m^o\ \mathit{to})\ t^o)\ (\textsf{\textbf{ind-=}}\ t^o\ m^o\ b^o))}\ [\text{EqE-5}]$$

[5] Thanks again, Per Martin-Löf.

[6] Thanks, Christine Paulin-Mohring (1962–)

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : (\textsf{=}\ X\ \mathit{from}\ \mathit{to}) \\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((\mathsf{x}\ X))\ (\Pi\ ((t\ (\textsf{=}\ X\ \mathit{from}\ \mathsf{x})))\ \mathcal{U})) \\ \Gamma \vdash b_1 \equiv b_2 : ((m_1\ \mathit{from})\ (\textsf{same}\ \mathit{from}))\end{array}}{\Gamma \vdash (\textsf{ind-=}\ t_1\ m_1\ b_1) \equiv (\textsf{ind-=}\ t_2\ m_2\ b_2) : ((m_1\ \mathit{to})\ t_1)}\ [\textsc{EqSame}\text{-}\textsf{ind-=}]$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{expr} \equiv \mathit{expr} : X \\ \Gamma \vdash m \equiv m : (\Pi\ ((\mathsf{x}\ X))\ (\Pi\ ((t\ (\textsf{=}\ X\ \mathit{expr}\ \mathsf{x})))\ \mathcal{U})) \\ \Gamma \vdash b_1 \equiv b_2 : ((m\ \mathit{expr})\ (\textsf{same}\ \mathit{expr}))\end{array}}{\Gamma \vdash (\textsf{ind-=}\ (\textsf{same}\ \mathit{expr})\ m\ b_1) \equiv b_2 : ((m\ \mathit{expr})\ (\textsf{same}\ \mathit{expr}))}\ [\textsc{EqSame}\text{-}\textsf{i-=}\iota]$$

Either

$$\frac{\Gamma \vdash P\ \ \textbf{type} \rightsquigarrow P^o \qquad \Gamma \vdash S\ \ \textbf{type} \rightsquigarrow S^o}{\Gamma \vdash (\textsf{Either}\ P\ S)\ \ \textbf{type} \rightsquigarrow (\textsf{Either}\ P^o\ S^o)}\ [\textsc{EitherF}]$$

$$\frac{\Gamma \vdash P_1 \equiv P_2\ \ \textbf{type} \qquad \Gamma \vdash S_1 \equiv S_2\ \ \textbf{type}}{\Gamma \vdash (\textsf{Either}\ P_1\ S_1) \equiv (\textsf{Either}\ P_2\ S_2)\ \ \textbf{type}}\ [\textsc{EitherSame}\text{-}\textsf{Either}]$$

$$\frac{\Gamma \vdash \mathit{lt} \in P \rightsquigarrow \mathit{lt}^o}{\Gamma \vdash (\textsf{left}\ \mathit{lt}) \in (\textsf{Either}\ P\ S) \rightsquigarrow (\textsf{left}\ \mathit{lt}^o)}\ [\textsc{EitherI-1}]$$

$$\frac{\Gamma \vdash \mathit{lt}_1 \equiv \mathit{lt}_2 : P}{\Gamma \vdash (\textsf{left}\ \mathit{lt}_1) \equiv (\textsf{left}\ \mathit{lt}_2) : (\textsf{Either}\ P\ S)}\ [\textsc{EitherSame}\text{-}\textsf{left}]$$

$$\frac{\Gamma \vdash \mathit{rt} \in S \rightsquigarrow \mathit{rt}^o}{\Gamma \vdash (\textsf{right}\ \mathit{rt}) \in (\textsf{Either}\ P\ S) \rightsquigarrow (\textsf{right}\ \mathit{rt}^o)}\ [\textsc{EitherI-2}]$$

$$\frac{\Gamma \vdash \mathit{rt}_1 \equiv \mathit{rt}_2 : S}{\Gamma \vdash (\textsf{right}\ \mathit{rt}_1) \equiv (\textsf{right}\ \mathit{rt}_2) : (\textsf{Either}\ P\ S)}\ [\textsc{EitherSame}\text{-}\textsf{right}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (\textsf{Either}\ P\ S)\ t^o) \\ \Gamma \vdash m \in (\Pi\ ((\mathsf{x}\ (\textsf{Either}\ P\ S)))\ \mathcal{U}) \rightsquigarrow m^o \\ \Gamma \vdash b_l \in (\Pi\ ((\mathsf{x}\ P))\ (m^o\ (\textsf{left}\ \mathsf{x}))) \rightsquigarrow b_l^o \\ \Gamma \vdash b_r \in (\Pi\ ((\mathsf{x}\ S))\ (m^o\ (\textsf{right}\ \mathsf{x}))) \rightsquigarrow b_r^o\end{array}}{\Gamma \vdash (\textsf{ind-Either}\ t\ m\ b_l\ b_r)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ (m^o\ t^o)\ (\textsf{ind-Either}\ t^o\ m^o\ b_l^o\ b_r^o))}\ [\textsc{EitherE}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 \equiv t_2 : (\textsf{Either}\ P\ S) \\ \Gamma \vdash m_1 \equiv m_2 : (\Pi\ ((x\ (\textsf{Either}\ P\ S)))\ \mathcal{U}) \\ \Gamma \vdash b_{l1} \equiv b_{l2} : (\Pi\ ((x\ P))\ (m_1\ (\textsf{left}\ x))) \\ \Gamma \vdash b_{r1} \equiv b_{r2} : (\Pi\ ((x\ S))\ (m_1\ (\textsf{right}\ x)))\end{array}}{\Gamma \vdash \begin{array}{c}(\textbf{ind-Either}\ t_1\ m_1\ b_{l1}\ b_{r1}) \\ \equiv \\ (\textbf{ind-Either}\ t_2\ m_2\ b_{l2}\ b_{r2})\end{array} : (m_1\ t_1)}\ [\textsc{EitherSame}\text{-}\textbf{ind-Either}]$$

$$\frac{\begin{array}{l}\Gamma \vdash lt_1 \equiv lt_2 : P \\ \Gamma \vdash m \equiv m : (\Pi\ ((x\ (\textsf{Either}\ P\ S)))\ \mathcal{U}) \\ \Gamma \vdash b_{l1} \equiv b_{l2} : (\Pi\ ((x\ P))\ (m\ (\textsf{left}\ x))) \\ \Gamma \vdash b_r \equiv b_r : (\Pi\ ((x\ S))\ (m\ (\textsf{right}\ x)))\end{array}}{\Gamma \vdash \begin{array}{c}(\textbf{ind-Either}\ (\textsf{left}\ lt_1)\ m\ b_{l1}\ b_r) \\ \equiv \\ (b_{l2}\ lt_2)\end{array} : (m\ (\textsf{left}\ lt_1))}\ [\textsc{EitherSame}\text{-}\textbf{i-E}\iota 1]$$

$$\frac{\begin{array}{l}\Gamma \vdash rt_1 \equiv rt_2 : S \\ \Gamma \vdash m \equiv m : (\Pi\ ((x\ (\textsf{Either}\ P\ S)))\ \mathcal{U}) \\ \Gamma \vdash b_l \equiv b_l : (\Pi\ ((x\ P))\ (m\ (\textsf{left}\ x))) \\ \Gamma \vdash b_{r1} \equiv b_{r2} : (\Pi\ ((x\ S))\ (m\ (\textsf{right}\ x)))\end{array}}{\Gamma \vdash \begin{array}{c}(\textbf{ind-Either}\ (\textsf{right}\ rt_1)\ m\ b_l\ b_{r1}) \\ \equiv \\ (b_{r2}\ rt_2)\end{array} : (m\ (\textsf{right}\ rt_1))}\ [\textsc{EitherSame}\text{-}\textbf{i-E}\iota 2]$$

Unit

$$\frac{}{\Gamma \vdash \textsf{Trivial}\ \ \textbf{type} \rightsquigarrow \textsf{Trivial}}\ [\textsc{TrivF}]$$

$$\frac{}{\Gamma \vdash \textsf{Trivial} \equiv \textsf{Trivial}\ \ \textbf{type}}\ [\textsc{TrivSame}\text{-}\textsf{Trivial}]$$

$$\frac{}{\Gamma \vdash \textsf{sole}\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \textsf{Trivial}\ \textsf{sole})}\ [\textsc{TrivI}]$$

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.

$$\frac{\Gamma \vdash c \equiv c : \textsf{Trivial}}{\Gamma \vdash c \equiv \textsf{sole} : \textsf{Trivial}}\ [\textsc{TrivSame}\text{-}\eta]$$

Absurdities

$$\frac{}{\Gamma \vdash \textsf{Absurd}\ \ \textbf{type} \leadsto \textsf{Absurd}}\ [\textsc{AbsF}]$$

$$\frac{}{\Gamma \vdash \textsf{Absurd} \equiv \textsf{Absurd}\ \ \textbf{type}}\ [\textsc{AbsSame}\text{-}\textsf{Absurd}]$$

$$\frac{\Gamma \vdash t \in \textsf{Absurd} \leadsto t^o \qquad \Gamma \vdash m\ \ \textbf{type} \leadsto m^o}{\Gamma \vdash (\textsf{\textbf{ind-Absurd}}\ t\ m)\ \ \textbf{synth} \leadsto (\texttt{the}\ m^o\ (\textsf{\textbf{ind-Absurd}}\ t^o\ m^o))}\ [\textsc{AbsE}]$$

$$\frac{\Gamma \vdash t_1 \equiv t_2 : \textsf{Absurd} \qquad \Gamma \vdash m_1 \equiv m_2 : \mathcal{U}}{\Gamma \vdash (\textsf{\textbf{ind-Absurd}}\ t_1\ m_1) \equiv (\textsf{\textbf{ind-Absurd}}\ t_2\ m_2) : m_1}\ [\textsc{AbsSame}\text{-}\textsf{\textbf{ind-Absurd}}]$$

$$\frac{\Gamma \vdash c_1 \equiv c_1 : \textsf{Absurd} \qquad \Gamma \vdash c_2 \equiv c_2 : \textsf{Absurd}}{\Gamma \vdash c_1 \equiv c_2 : \textsf{Absurd}}\ [\textsc{AbsSame}\text{-}\eta]$$

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.

$$\frac{}{\Gamma \vdash \mathcal{U} \;\; \textbf{type} \rightsquigarrow \mathcal{U}}\,[\text{UF}] \qquad \frac{}{\Gamma \vdash \mathcal{U} \equiv \mathcal{U} \;\; \textbf{type}}\,[\text{U}\textsc{Same}\text{-}\mathcal{U}]$$

$$\frac{}{\Gamma \vdash \mathsf{Atom} \;\; \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ \mathsf{Atom})}\,[\text{UI-1}] \qquad \frac{}{\Gamma \vdash \mathsf{Atom} \equiv \mathsf{Atom} : \mathcal{U}}\,[\text{U}\textsc{Same}\text{-}\mathsf{Atom}]$$

$$\frac{\Gamma \vdash A \in \mathcal{U} \rightsquigarrow A^o \qquad \Gamma, x : A^o \vdash D \in \mathcal{U} \rightsquigarrow D^o}{\Gamma \vdash (\Sigma\ ((x\ A))\ D) \;\; \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ (\Sigma\ ((x\ A^o))\ D^o))}\,[\text{UI-2}]$$

$$\frac{\Gamma \vdash A \in \mathcal{U} \rightsquigarrow A^o \qquad \Gamma, x : A^o \vdash (\Sigma\ ((x_1\ A_1)\ \ldots\ (x_n\ A_n))\ D) \in \mathcal{U} \rightsquigarrow Z}{\Gamma \vdash (\Sigma\ ((x\ A)\ (x_1\ A_1)\ \ldots\ (x_n\ A_n))\ D) \;\; \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ (\Sigma\ ((x\ A^o))\ Z))}\,[\text{UI-3}]$$

$$\frac{\Gamma \vdash A \in \mathcal{U} \rightsquigarrow A^o \qquad \Gamma \vdash \textbf{fresh} \rightsquigarrow x \qquad \Gamma, x : A^o \vdash D \in \mathcal{U} \rightsquigarrow D^o}{\Gamma \vdash (\mathsf{Pair}\ A\ D) \;\; \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ (\Sigma\ ((x\ A^o))\ D^o))}\,[\text{UI-4}]$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 : \mathcal{U} \qquad \Gamma, x : A_1 \vdash D_1 \equiv D_2 : \mathcal{U}}{\Gamma \vdash (\Sigma\ ((x\ A_1))\ D_1) \equiv (\Sigma\ ((x\ A_2))\ D_2) : \mathcal{U}}\,[\text{U}\textsc{Same}\text{-}\Sigma]$$

$$\frac{\Gamma \vdash X \in \mathcal{U} \rightsquigarrow X^o \qquad \Gamma, x : X^o \vdash R \in \mathcal{U} \rightsquigarrow R^o}{\Gamma \vdash (\Pi\ ((x\ X))\ R) \;\; \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ (\Pi\ ((x\ X^o))\ R^o))}\,[\text{UI-5}]$$

$$\frac{\Gamma \vdash X \in \mathcal{U} \rightsquigarrow X^o \qquad \Gamma, x : X^o \vdash (\Pi\ ((x_1\ X_1)\ \ldots\ (x_n\ X_n))\ R) \in \mathcal{U} \rightsquigarrow R^o}{\Gamma \vdash (\Pi\ ((x\ X)\ (x_1\ X_1)\ \ldots\ (x_n\ X_n))\ R)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ (\Pi\ ((x\ X^o))\ R^o))}\ [\text{UI-6}]$$

$$\frac{\Gamma \vdash X \in \mathcal{U} \rightsquigarrow X^o \qquad \Gamma \vdash \textbf{fresh} \rightsquigarrow x \qquad \Gamma, x : X^o \vdash R \in \mathcal{U} \rightsquigarrow R^o}{\Gamma \vdash (\rightarrow\ X\ R)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ (\Pi\ ((x\ X^o))\ R^o))}\ [\text{UI-7}]$$

$$\frac{\begin{array}{l}\Gamma \vdash X \in \mathcal{U} \rightsquigarrow X^o \\ \Gamma \vdash \textbf{fresh} \rightsquigarrow x \\ \Gamma, x : X^o \vdash (\rightarrow\ X_1\ \ldots\ X_n\ R) \in \mathcal{U} \rightsquigarrow R^o\end{array}}{\Gamma \vdash (\rightarrow\ X\ X_1\ \ldots\ X_n\ R)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ (\Pi\ ((x\ X^o))\ R^o))}\ [\text{UI-8}]$$

$$\frac{\Gamma \vdash X_1 \equiv X_2 : \mathcal{U} \qquad \Gamma, x : X_1 \vdash Y_1 \equiv Y_2 : \mathcal{U}}{\Gamma \vdash (\Pi\ ((x\ X_1))\ Y_1) \equiv (\Pi\ ((x\ X_2))\ Y_2) : \mathcal{U}}\ [\text{USame-}\Pi]$$

$$\frac{}{\Gamma \vdash \mathsf{Nat}\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ \mathsf{Nat})}\ [\text{UI-9}] \qquad \frac{}{\Gamma \vdash \mathsf{Nat} \equiv \mathsf{Nat} : \mathcal{U}}\ [\text{USame-Nat}]$$

$$\frac{\Gamma \vdash E \in \mathcal{U} \rightsquigarrow E^o}{\Gamma \vdash (\mathsf{List}\ E)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ (\mathsf{List}\ E^o))}\ [\text{UI-10}]$$

$$\frac{\Gamma \vdash E_1 \equiv E_2 : \mathcal{U}}{\Gamma \vdash (\mathsf{List}\ E_1) \equiv (\mathsf{List}\ E_2) : \mathcal{U}}\ [\text{USame-List}]$$

$$\frac{\Gamma \vdash E \in \mathcal{U} \rightsquigarrow E^o \qquad \Gamma \vdash \ell \in \mathsf{Nat} \rightsquigarrow \ell^o}{\Gamma \vdash (\mathsf{Vec}\ E\ \ell)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ (\mathsf{Vec}\ E^o\ \ell^o))}\ [\text{UI-11}]$$

$$\frac{\Gamma \vdash E_1 \equiv E_2 : \mathcal{U} \qquad \Gamma \vdash \ell_1 \equiv \ell_2 : \mathsf{Nat}}{\Gamma \vdash (\mathsf{Vec}\ E_1\ \ell_1) \equiv (\mathsf{Vec}\ E_2\ \ell_2) : \mathcal{U}}\ [\text{USame-Vec}]$$

$$\frac{\Gamma \vdash X \in \mathcal{U} \rightsquigarrow X^o \qquad \Gamma \vdash \mathit{from} \in X^o \rightsquigarrow \mathit{from}^o \qquad \Gamma \vdash \mathit{to} \in X^o \rightsquigarrow \mathit{to}^o}{\Gamma \vdash (\mathbf{=}\ X\ \mathit{from}\ \mathit{to})\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ (\mathbf{=}\ X^o\ \mathit{from}^o\ \mathit{to}^o))}\ [\text{UI-12}]$$

$$\frac{\Gamma \vdash X_1 \equiv X_2 : \mathcal{U} \qquad \Gamma \vdash \mathit{from}_1 \equiv \mathit{from}_2 : X_1 \qquad \Gamma \vdash \mathit{to}_1 \equiv \mathit{to}_2 : X_1}{\Gamma \vdash (\mathsf{=}\ X_1\ \mathit{from}_1\ \mathit{to}_1) \equiv (\mathsf{=}\ X_2\ \mathit{from}_2\ \mathit{to}_2) : \mathcal{U}}\ [\text{U}\textsc{Same}\text{-}\mathsf{=}]$$

$$\frac{\Gamma \vdash P_1 \equiv P_2 : \mathcal{U} \qquad \Gamma \vdash S_1 \equiv S_2 : \mathcal{U}}{\Gamma \vdash (\mathsf{Either}\ P_1\ S_1) \equiv (\mathsf{Either}\ P_2\ S_2) : \mathcal{U}}\ [\text{U}\textsc{Same}\text{-}\mathsf{Either}]$$

$$\frac{\Gamma \vdash P \in \mathcal{U} \rightsquigarrow P^o \qquad \Gamma \vdash S \in \mathcal{U} \rightsquigarrow S^o}{\Gamma \vdash (\mathsf{Either}\ P\ S)\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ (\mathsf{Either}\ P^o\ S^o))}\ [\text{UI-13}]$$

$$\frac{}{\Gamma \vdash \mathsf{Trivial}\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ \mathsf{Trivial})}\ [\text{UI-14}]$$

$$\frac{}{\Gamma \vdash \mathsf{Trivial} \equiv \mathsf{Trivial} : \mathcal{U}}\ [\text{U}\textsc{Same}\text{-}\mathsf{Trivial}]$$

$$\frac{}{\Gamma \vdash \mathsf{Absurd}\ \ \textbf{synth} \rightsquigarrow (\texttt{the}\ \mathcal{U}\ \mathsf{Absurd})}\ [\text{UI-15}]$$

$$\frac{}{\Gamma \vdash \mathsf{Absurd} \equiv \mathsf{Absurd} : \mathcal{U}}\ [\text{U}\textsc{Same}\text{-}\mathsf{Absurd}]$$

The Grammar of Pie

e	::=	(the e e)	Type annotation
	|	x	Variable reference
	|	Atom	Atom type
	|	'⌈sym⌉	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
	|	⌈n⌉	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 $\to$ 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 →. 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