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!

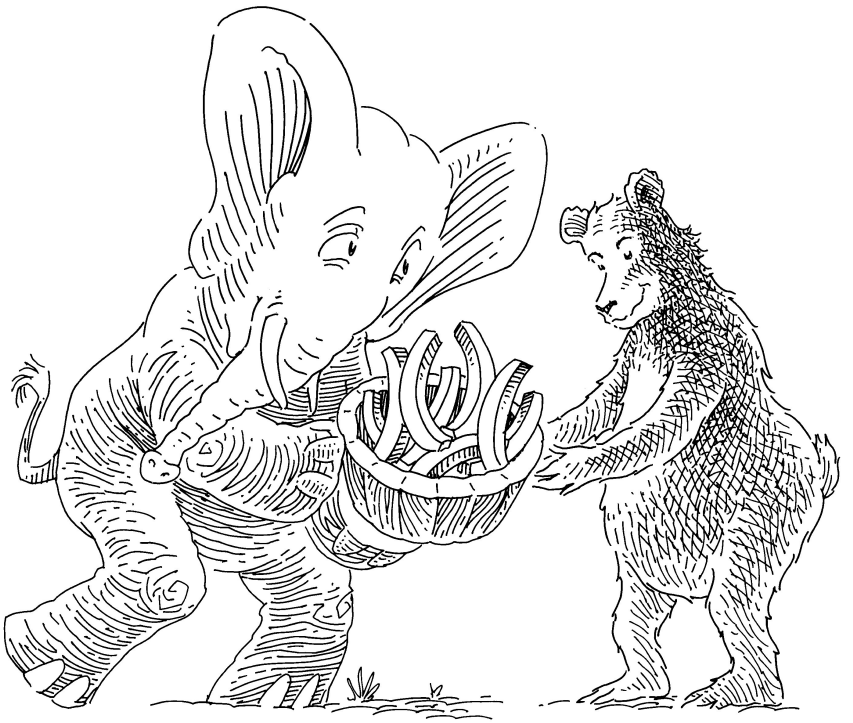
Daniel P. Friedman
Bloomington, Indiana

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*.†

11 What is the point of a judgment?

†Thanks, Per Martin-Löf (1942–).

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).

Is

(cons 'ratatouille 'baguette)

the same

(Pair Atom Atom)

as

(cons 'baguette 'baguette)?

[24] 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).

How can

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

be described?

[25] It is an

Atom.

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.

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

The third form of judgment is

____ is a type.

[27] 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.

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

[33] Aren't judgments sentences?

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.

[34] Ah, so coming to know that

(Pair Atom Atom)

and

(Pair Atom Atom)

are the same type was a judgment?

It was.

Is 'pêche the same 'fruit as 'pêche?

[35] Good question.

Is 'pêche a 'fruit?

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*.

[36] Which are these?

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.

[37] 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.

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.

42 '**baguette** certainly seems more direct than

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

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.

43 Is '**olive-oil** the normal form of

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

?

That question is incomplete.

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

44 Is '**olive-oil** the normal form of the Atom

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

?

Yes, it is.

Is

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

a normal

```
(Pair Atom Atom)?†
```

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

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

Does every expression have a normal form?

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.

[46] 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.

What is the normal form of

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

[47] What about the type?

If the type is

(Pair Atom Atom),

then the normal form is

(cons 'aubergine 'courgette).

Nice catch!

The previous description of what it means to be a

(Pair Atom Atom)

is incomplete. It must mean ...

[48] ... 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.

[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).
```

[†] 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.

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?

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

Is one the same Nat as 1?

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

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))
```

[77] Why is the box around the definition dashed?

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.

[78] What is the matter with that definition?

It looks okay.

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))
```

[79] 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*.

Here's another expression that is not normal.

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

Is this the most direct way of writing 3?

[88] No.

What exactly is a constructor?

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.

[89] What are some examples of constructors?

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

[90] What is the relationship between values and normal forms?

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?

[91] 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
 [          ])
```

[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 **car**s are the same Atom and their **cdr**s 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.

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

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

Is Nat a constructor, then?

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?

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

What is Pair called, then?

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)?
```

[119] No.

Isn't it a

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

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).

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

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

Indeed it is.

[121] 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.

Oops! It is also known as **lambda**.†

†λ can be optionally written `lambda`.

[7] Oh, right, λ builds functions.

Does this mean that λ is a constructor?

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.

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

How can functions have an eliminator?

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

[9] Okay.

Eliminating Functions

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

What is the value of

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

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

Right.

What is the value of

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

[11] 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.

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: ->.

Yes, these are also

```
(→ Atom
  (Pair Atom Atom))
```

because they too become a

```
(Pair Atom Atom)
```

when given an Atom as an argument.

Yes, because

```
(car
  (cons Atom 'pepper))
```

is Atom and

```
(cdr
  (cons 'salt Atom))
```

is also Atom.

---

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

---

[16] Are they also

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

---

[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, 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

$(\lambda\ (y)$
$\quad (f\ y)),$

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

```
(car x)
```

a value?

[26] No, because **car** is an eliminator, and eliminators take apart values.

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

[33] Okay, then,

```
(λ (a d)
  (cons a d))
```

is an

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

These expressions are certainly getting long.

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

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 defined.

[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**

($\rightarrow$ 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).

What is the normal form of

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

[50] Is it 11 because

```
((λ (n)
   (+ 6 n))
 5)
```

is 11?

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.

[51] Ah, so the normal form is 10 because

```
((λ (n)
   (+ 6 n))
 4)
```

is 10.

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.

[52] The sum of Nats is a Nat.

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

Right.

Now define it.

[53] How?

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.

[54] 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)?

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

(**+** 5 [(***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*)?

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

(**+** (add1 *n*) [(***gauss*** *n*)])

What about zero?

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

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?

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).

[70] Are 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, $\rightarrow$, and $\mathcal{U}$.

[71] Are all types values?

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.

No.

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

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

[72] Which expressions are described by

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

?

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: **+**.

What about an expression that exchanges the Nats?

88 How about

(λ (*a d*)
  (cons *d a*))?

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

89 That must be

(λ (*a d*)
  a).

Very close. Actually, it would be

(λ (*a d*)
  a).

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

Indeed. To get a value of type $X^{\dagger}$ from a ***Pear***, one must have an expression of type

(→ Nat Nat
  X).

What type does **+** have?

$^{\dagger}X$ can be any type at all.

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

(→ Nat Nat
  Nat).

That's right.

What would be the type of

(λ (*a d*)
  (cons *d a*)),

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

92 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.

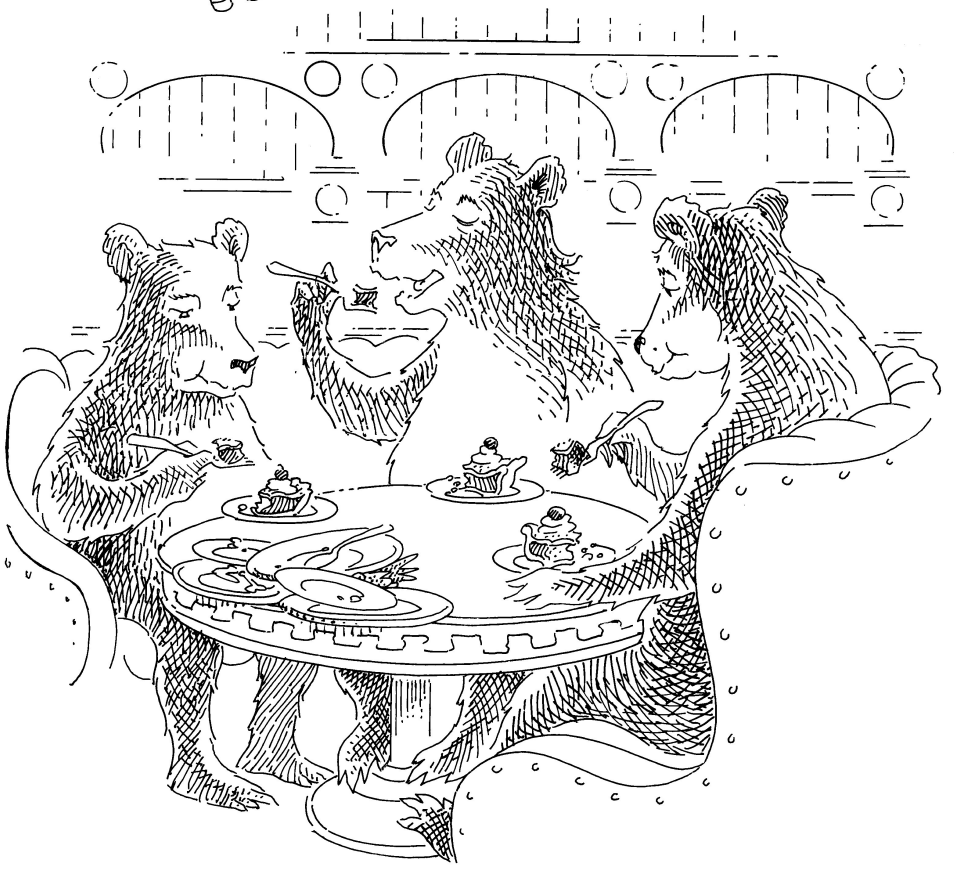
[102] It's a good thing recursion is not an option.

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

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

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**

($\rightarrow$ 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.†

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

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

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

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)))))).
```

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

What is the value?

[37] 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.

What is its normal form?

[38] It is

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

The target is zero and the base is 0.

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

[39] What type should the base and step have?

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.

[40] Is that all?

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?

[41] 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 $zerop_{n-1}$ and $gauss_{n-1}$ 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?

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.

---

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

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."

---

[62] ∗ takes two Nats and their product is a Nat. So here is ∗'s type.

```
(claim ∗
  (→ Nat Nat
    Nat))
```

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

---

[63] ∗ adds *j*, its second argument, to the almost-answer.

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))))
```

---

[64] 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.

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))))).
```

[67] Does that mean that

```
(→ Nat Nat Nat
  Nat)
```

is also a shorter way of writing something?

It is a shorter way of writing

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

[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?

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

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

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

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**

($\rightarrow$ 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

$$(\rightarrow\ Y\ X).$$

[18] An

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

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,

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

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 $\Pi^{\dagger}$ can.

[23] What does Π mean?

$^{\dagger}\Pi$ is pronounced "pie," and it can optionally be written `Pi`.

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 → 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

$(\Pi\ ((Y\ \mathcal{U}))\ X)$

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

$$(\Pi\ ((Y\ \mathcal{U}))\ X)$$

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

$$(\Pi\ ((Y\ \mathcal{U})\ (Z\ \mathcal{U}))\ X)?$$

[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))
```

Is

```
(twin-Atom 'cherry-pie)
```

the same

```
(Pair Atom Atom)
```

as

```
((twin Atom) 'cherry-pie)?
```

[55] Yes, and its value, but also its normal form is

```
(cons 'cherry-pie 'cherry-pie).
```

There's twice as much for dessert!

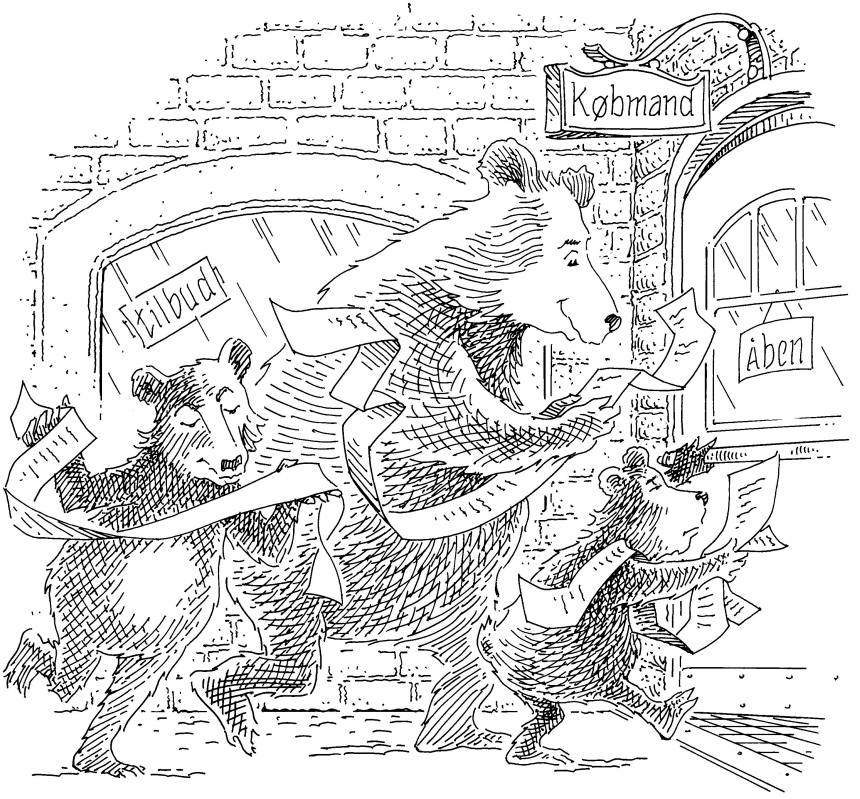
Now go to your favorite confectionary shop and share a delicious cherry Π.

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 ['ʁuˌbʁœðˀ]. 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 $(\rightarrow \text{Nat } X\ X)$.

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 $(\rightarrow E\ (\text{List } E)\ X\ X)$.

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**
  ($\rightarrow$ 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.

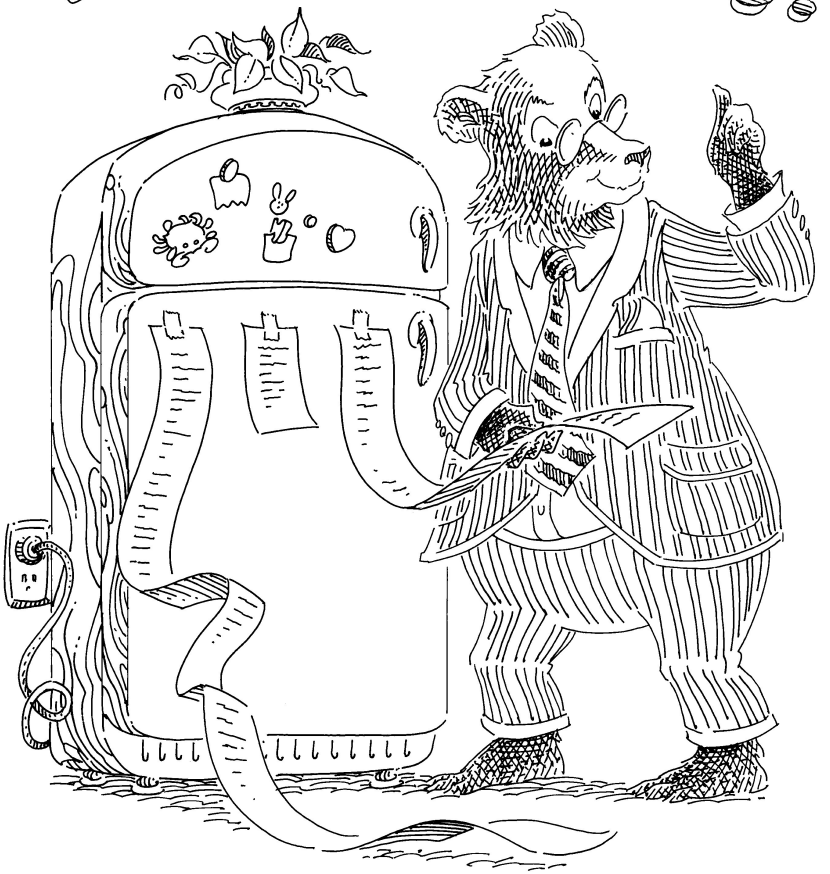
Make your bread your own by adding sunflower seeds, flax seeds, dark malt, pumpkin seeds, or whatever else strikes your fancy.

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 ⊓ 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?	7	Can types contain expressions that aren't types?

†Pronounced "list of E with length k," or simply "list of E length k."

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 $\mathcal{U}$ after argument names in Π-expressions.

The E in

```
(→ (Vec E (add1 ℓ))
  E)
```

refers to whatever $\mathcal{U}$ 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

(Π ((*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**

$(\lambda\ (y)$
$\quad x)$

is a

$(\Pi\ ((y\ Y))$
$\quad X).$

The Final Law of Application

If f **is a**

$(\Pi\ ((y\ Y))$
$\quad X)$

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

$(\Pi\ ((y\ Y))$
$\quad X),$

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

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?

[1] The type is

```
(→ Nat
   (List Atom))
```

because ***peas*** should be able to produce any number of peas.

How many peas should ***peas*** produce?

[2] It depends.

What does it depend on?

[3] It depends on how many peas are required—that is, the argument.

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.

[4] 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)))
```

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**?

[5] 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.

---

rec-Nat can do anything that **iter-Nat** can.

It is called **ind-Nat**, short for "induction on 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.

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.

---

[6] What about **iter-Nat**?

[7] Is there something more powerful to use?

[8] What is **ind-Nat**?

[9] There is a Nat called *how-many-peas* included in
(Vec Atom *how-many-peas*).

Is this a dependent type?

[10] What does this extra argument look like?

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–).

[11] So the motive is a function whose body is a $\mathcal{U}$.

It is. The motive explains *why* the target is to be eliminated.

What is the motive for ***peas***?

[12] 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.

Here is ***mot-peas***.

```
(define mot-peas
  (λ (k)
    (Vec Atom k)))
```

What is the value of (***mot-peas*** zero)?

[13] 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 (peas_ℓ-1)
      (vec:: 'pea peas_ℓ-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
  $\mathcal{U}$),
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 $\mathcal{U}$ 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

$$(\rightarrow\ (\rightarrow\ (\mathsf{Vec}\ E\ (\mathsf{add1}\ \ell\text{-}1))\ E)\ (\rightarrow\ (\mathsf{Vec}\ E\ (\mathsf{add1}\ (\mathsf{add1}\ \ell\text{-}1)))\ E))$$

because the step must construct a

$$(\textbf{\textit{mot-last}}\ E\ (\mathsf{add1}\ \ell\text{-}1))$$

from a

$$(\textbf{\textit{mot-last}}\ E\ \ell\text{-}1).$$

How can that type be explained in prose?

[48] The step transforms a

last

function for

ℓ

into a

last

function for

$(\mathsf{add1}\ \ell)$.

---

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:

$$(\Pi\ ((n\ \mathsf{Nat}))\ (\rightarrow\ (mot\ n)\ (mot\ (\mathsf{add1}\ n))))$$

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.

What is $last_{\ell\text{-}1}$'s type in the outer λ-expression in frame 49?

[53] $last_{\ell\text{-}1}$ is an

(→ (Vec *E* (add1 *ℓ-1*)) *E*),

which is the value of (***mot-last*** *ℓ-1*).

Now it is time to define ***last***. The **claim** is in frame 28 on page 151.

[54] Here goes.

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

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)))

[55] 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 $last_{\ell\text{-}1}$, to find the ***last*** of its own (**tail** *es*),

step-drop-last

uses its almost-answer, $drop\text{-}last_{\ell\text{-}1}$, 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 ($drop\text{-}last_{\ell\text{-}1}$ (**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 `TODO`s.

[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

How was that mushroom pot pie?

[1] Delicious, though very filling. How about something less filling here?

How about a
(***sandwich*** **'**hoagie)?

[2] That should be manageable.

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

[3] Let's find out.

Let's start.

1. (**+** (add1 zero))
2. (λ (*j*)
   (**iter-Nat** (add1 zero)
   j
   step-+))

This is the value.

[4] 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*))

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)?

[5] 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.

Is

```
(= Atom 'kale 'blackberries)
```

a type?

[14] Yes, because Atom is a type and both 'kale and 'blackberries are Atoms.

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 $1 + 1 = 2$."

[15] Yes, because Nat is a type and both (+ 1 1) and 2 are Nats.

Is

```
(= (car (cons Nat 'kale))
   17
   (+ 14 3))
```

a type?

[16] Yes, it is, because

```
(car (cons Nat 'kale))
```

and Nat are the same type, and the FROM and the TO are both Nats.

Is

```
(= (car (cons Nat 'kale))
   15
   (+ 14 3))
```

a type?

[17] 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 =?

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*.

[18] 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))))
```

That's a solid start. What goes in the box?

Right, the box should contain an
(= Nat (+ 1 n) (add1 n)).

What is the normal form of the box's type?

That's right.

Now finish the definition.

---

[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)
    [          ]))
```

---

[39] Following the Law of λ,
(= Nat (+ 1 n) (add1 n))
is the type of the body of the λ-expression.

---

[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)).

---

[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.

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).

[48] Okay.

Also, if the target of an eliminator expression is neutral, then the entire expression is neutral.

[49] 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.

No.

The body of the λ-expression

```
(λ (x)
  (add1 x))
```

contains the variable *x*, but λ-expressions are values, not neutral expressions.

[50] 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

$$(\lambda\ (x)\ (f\ x)),$$

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 $\lambda^{\dagger}$ on page 140, f is the same as

$$(\lambda\ (x)\ (f\ x)),$$

but they are *not* written identically.

[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?

[†] 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.

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 $\rightarrow$, 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=add1n-1)
      [ incr=add1n-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*.

If *f* is an

(→ *X*
Y)

and *target* is an

(**=** *X from to*),

then

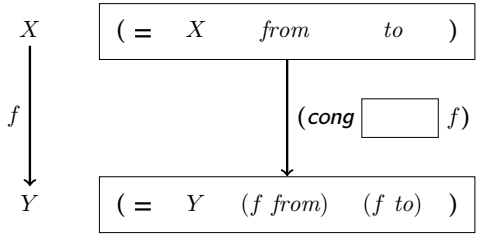
(**cong** *target f*)

is an

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

[74] Is there another way to look at **cong**?

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**
(→ *X*
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 $\mathit{incr{=}add1}_{n\text{-}1}$.

What are *from* and *to*?

[76] $\mathit{incr{=}add1}_{n\text{-}1}$'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

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

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.