

CS 565

Programming Languages (graduate) Spring 2024

Week 1

Introduction, Functional Programming, Datatypes

Administrivia

2



Who:

Instructor: Suresh Jagannathan

Office Hours: MW, 12pm - 1pm (LWSN 3154J)

TA₁: Pedro da Costa Abreu

Office Hours: Wed. 11 - 12pm HAAS G072

TA₂: Srinivasa Arun Yeragudipati

Office Hours: Tues. 2 - 3pm HAAS 143

Where: GRSM 103

When: January 8 - April 27, 2024

Discussion Board: Piazza

Homeworks and Quizzes: Brightspace and Gradescope

How

3

Lectures

- In-person lectures

Homeworks (35%)

- Approximately 7 over the course of the semester
- Typically 2 weeks to complete
- Involves programming and proving in Coq and Dafny

Quizzes (10%)

- Once a week
- Short answer, multiple-choice on Gradescope
- Covers material covered in lecture

Midterm (25%)

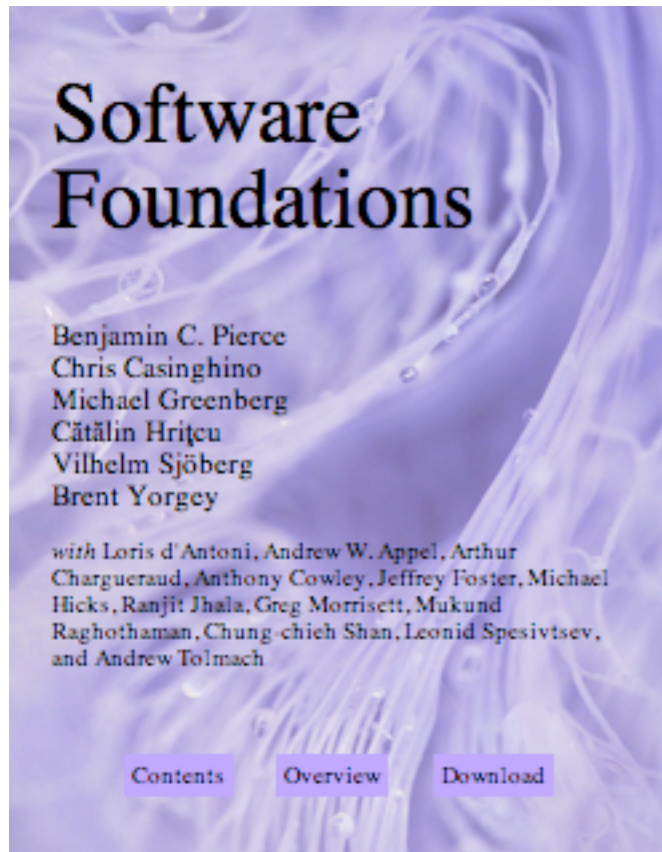
- Evening exam (paper)
- March 20, 8 - 9:30 PM, KRAN G016

Final (30%)

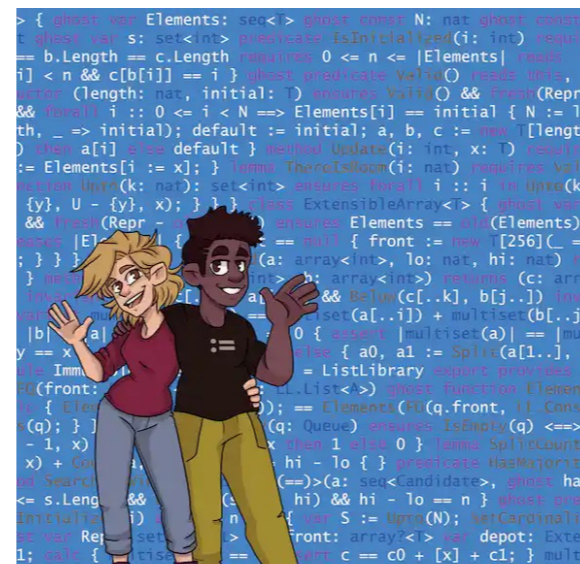
- Take home

Textbooks

4



Software Foundations



K. Rustan M. Leino
illustrated by Kaleb Leino

PROGRAM PROOFS

Program Proofs

Additional Resources

- Types and Programming Languages
(Pierce, 2002 MIT Press)
- Certified Programming with Dependent Types
(Chlipala, eBook)

How

5

to succeed
in CS 565

Should be familiar with:

- ▶ Programming in a high-level language
(Python, Java, Rust, Haskell, OCaml, ...)
- ▶ Basic logic and proofs techniques
sets, relations, functions, ...
- ▶ Basic data structures and algorithms

Participate!

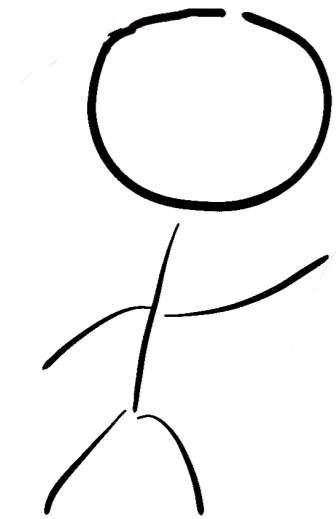
Think before you prove!

What

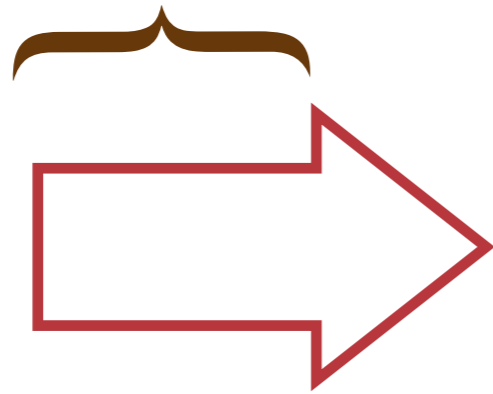
6

The focus in this class

Describe

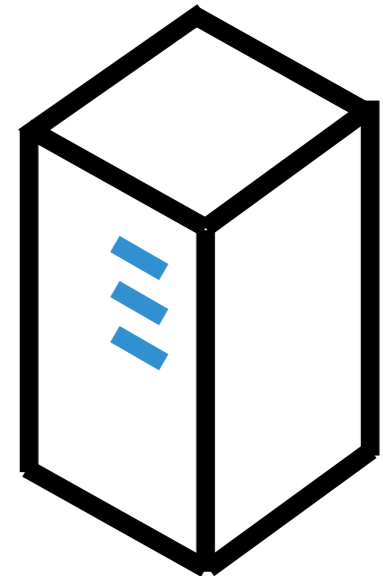
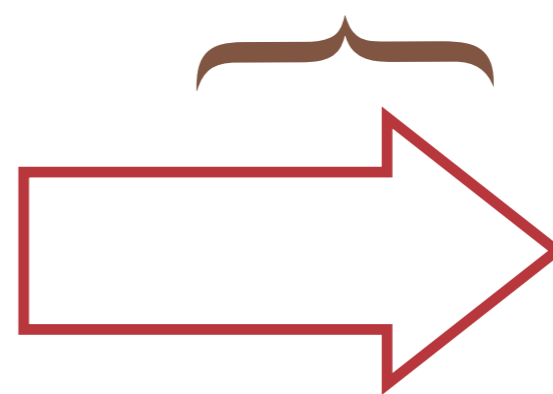


You



Programming Language

Implement



The Machine

What

7



Proof Assistant:

- ★ Generate and Check Proofs
- ★ Web Page: coq.inria.fr



Verifier-Aware Programming Language

- ★ Write programs along with specifications that are automatically verified
- ★ Web Page: dafny.org

Why?

8

Other Compilers: 325
CompCert: <10 (in unverified front-end)

The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith **cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task.**

The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations **within a proof framework**, where safety checks are explicit and machine-checked, has **tangible benefits** for compiler users.

Finding and Understanding Bugs in C Compilers [Yang et al. PLDI 2011]

IronFleet: Proving Practical Distributed Systems Correct

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch,
Bryan Parno, Michael L. Roberts, Srinath Setty, Brian Zill

Microsoft Research

Distributed systems are notorious for harboring subtle bugs. Verification can, in principle, eliminate these bugs a priori, but verification has historically been difficult to apply at full-program scale, much less distributed-system scale.

We describe a methodology for building practical and provably correct distributed systems based on a unique blend of TLA-style state-machine refinement and Hoare-logic verification. We demonstrate the methodology on a complex implementation of a Paxos-based replicated state machine library and a lease-based sharded key-value store. We prove that each obeys a concise safety specification, as well as desirable liveness requirements. Each implementation achieves performance competitive with a reference system. With our methodology and lessons learned, we aim to raise the standard for distributed systems from “tested” to “correct.”

In many cases, Dafny’s automated reasoning allows the developer to write little or no proof annotation. For instance, Dafny excels at automatically proving statements about linear arithmetic. Also, its heuristics for dealing with quantifiers, while imperfect, often produce proofs automatically.

Dafny can also prove more complex statements automatically. For instance, the lemma proving that IronRSL’s `ImplNext` always meets the reduction-enabling obligation consists of only two lines: one for the precondition and one for the postcondition. Dafny automatically enumerates all ten possible actions and all of their subcases, and observes that all of them produce I/O sequences satisfying the property.

SOSP’15

What

9

Foundations:

- ★ Functional Programming
- ★ Polymorphism and Higher-Order Programming
- ★ Dependent Types
- ★ Propositions, Evidence, and Relations

Program Semantics:

- ★ Operational Semantics
- ★ Denotational Semantics
- ★ Hoare Logic and Axiomatic Semantics

Types:

- ★ Type Soundness
- ★ Type Inference
- ★ Simply-Typed Lambda Calculus
- ★ System F

Automated Program Verification

- ★ Hoare Logic and Axiomatic Semantics
- ★ Verification-Aware Languages

Functional Programming

10

- We'll start our investigation by considering a small functional language
- These languages tend to have a small core set of features
 - Datatypes, functions, and their application
 - Written in Gallina, the specification and programming language for Coq

```
Definition double (n : nat) : nat := n + n.
```

Functions

11

- Functional languages tend to have a small core
- Standard libraries tend to have the usual suspects
- Functions are **applied** to arguments
- Functions are **pure**: consume values, produce values

```
Definition double (n : nat) : nat := n + n.
```

```
Eval compute in (double 1). (* = 2 *)
```

Functions

12

- Functional languages tend to have a small core
- Standard libraries tend to have the usual suspects
- Functions are **applied** to arguments
- Functions are **pure**: consume values, produce values

```
Definition double (n : nat) : nat :=  
  plus n n.
```

```
Eval compute in (double 1). (* = 2 *)
```

Functions

13

- Functional languages tend to have a small core
- Standard libraries tend to have the usual suspects
- Functions are **applied** to arguments
- Functions are **pure**: consume values, produce values

```
Definition concat (s1 : string) (s2 : string)
                  (s3 : string) :=
  append s1 (append s2 s3).
```

```
Eval compute in (concat "Hello" " " "World").
(* = "Hello World" *)
```

Functions

14

- Functional languages tend to have a small core
- Standard libraries tend to have the usual suspects
- Functions are **applied** to arguments
- Functions are **pure**: consume value, produce value

```
Definition concat (s1 s2 s3 : string) : string :=  
  append s1 (append s2 s3).
```

```
Eval compute in (concat "Hello" " " "World").  
  (* = "Hello World" *)
```

Functions

15

- Functional languages tend to have a small core
- Standard libraries tend to have the usual suspects
- Functions are **applied** to arguments
- Functions are **pure**: consume value, produce value
- Coq can automatically infer many type annotations

```
Definition concat s1 s2 s3 :=  
  append s1 (append s2 s3).
```

```
Eval compute in (concat "Hello" " " "World").  
(* = "Hello World" *)
```

Building Blocks

16

Given the following ingredients:

- bool: a datatype for booleans
- andb: logical and
- orb: logical or
- negb: logical negation

Define a boolean equality function

```
Definition eqb (b1 b2 : bool) : bool :=  
  orb (andb b1 b2) (andb (negb b1) (negb b2)).
```


Algebraic Data Types

17

Enumerated types are the simplest data types in Coq:

```
Inductive bool : Type :=  
| true : bool  
| false : bool.
```

Algebraic Data Types

18

- Enumerated types are the simplest data types in Coq
- Type annotations can be inferred here as well

```
Inductive bool :=
```

```
| true
```

```
| false.
```

Algebraic Data Types

19

- Enumerated types are the simplest data types in Coq
- Type annotations can be inferred here
- Constructors describe how to **introduce** a value of a type

```
Inductive bool :=
```

```
| true
```

```
| false.
```

```
Inductive weekdays :=
```

```
| monday | tuesday | wednesday | thursday
```

```
| friday : weekdays.
```

Pattern Matching

20

- Pattern matching lets a program use values of a type
- Coq only permits **total** functions
 - A total function is defined on all values in its domain

```
Definition negb (b : bool) : bool :=  
  match b with  
  | true => false  
  | false => true  
  end.
```

```
Eval compute in (negb true). (* = false *)
```

Pattern Matching

21

- Pattern matching lets a program use values of a type
- Coq only permits **total** functions
 - A total function is defined on all values in its domain

```
Definition eqb (b1 b2 : bool) : bool :=  
  match b1, b2 with  
  | true, true => true  
  | false, false => true  
  | false, true => false  
  | true, false => false  
  end.
```

Pattern Matching

22

- Pattern matching lets a program use values of a type
- Coq only permits **total** functions
 - A total function is defined on all values in its domain
- Underscores are the wildcard pattern (don't care)

```
Definition eqb (b1 b2 : bool) : bool :=  
  match b1, b2 with  
  | true, true => true  
  | false, false => true  
  | _, _ => false  
  end.
```

Compound ADTs

23

- Can build new ADTs from existing ones:
 - A color is either black, white, or a primary color
 - Need to apply primary to something of type rgb:
- ADTs are **algebraic** because they are built from a small set of operators (sums of product).

Inductive `rgb` : Type := | red | green | blue.

Inductive `color` := | black | white
| primary (p : rgb).

Eval `compute in` (primary red). (* = primary red *)

Pattern Matching²

24

- Patterns on compound types need to mention arguments
 - Can be a **variable**

```
Definition monochrome (c : color) : bool :=  
  match c with  
  | black => true  
  | white => true  
  | primary p => false  
  end.
```


Pattern Matching²

25

- Patterns on compound types need to mention arguments
 - Can be a **variable**
 - Can be a **pattern** for the type of the argument

```
Definition isred (c : color) : bool :=  
  match c with  
  | black => false  
  | white => false  
  | primary red => true  
  | primary _ => false  
  end.
```

Concept Check

26

- How many colors are there?
- In general, each ADT defines an algebra whose operations are the constructors

```
Inductive rgb : Type := | red | green | blue.
```

```
Inductive color := | black | white  
| primary (p : rgb).
```

```
Eval compute in (primary red). (* = primary red *)
```

Concept Check²

27

- Define a type for the ‘basic’ (h, a, and p) html tags:
 - A header should include a nat indicating its importance
 - The anchor tag should include a string for its destination
 - The paragraph doesn’t need anything extra

```
Inductive tag : Type :=  
| h (importance : nat)  
| a (href : string)  
| p.
```

Concept Check²

28

- Define a pretty printer for opening a tag

```
(* pp (h l) = "<h l>" *) *
```

- Assume we have a `natToString` function

```
Inductive tag : Type :=  
| h (importance : nat)  
| a (href : string)  
| p.
```

Concept Check²

29

- ★ Define a pretty printer for opening a tag
 - ★ (* pp (h 1) = "<h1>" *) *)
 - ★ Assume we have a natToString function

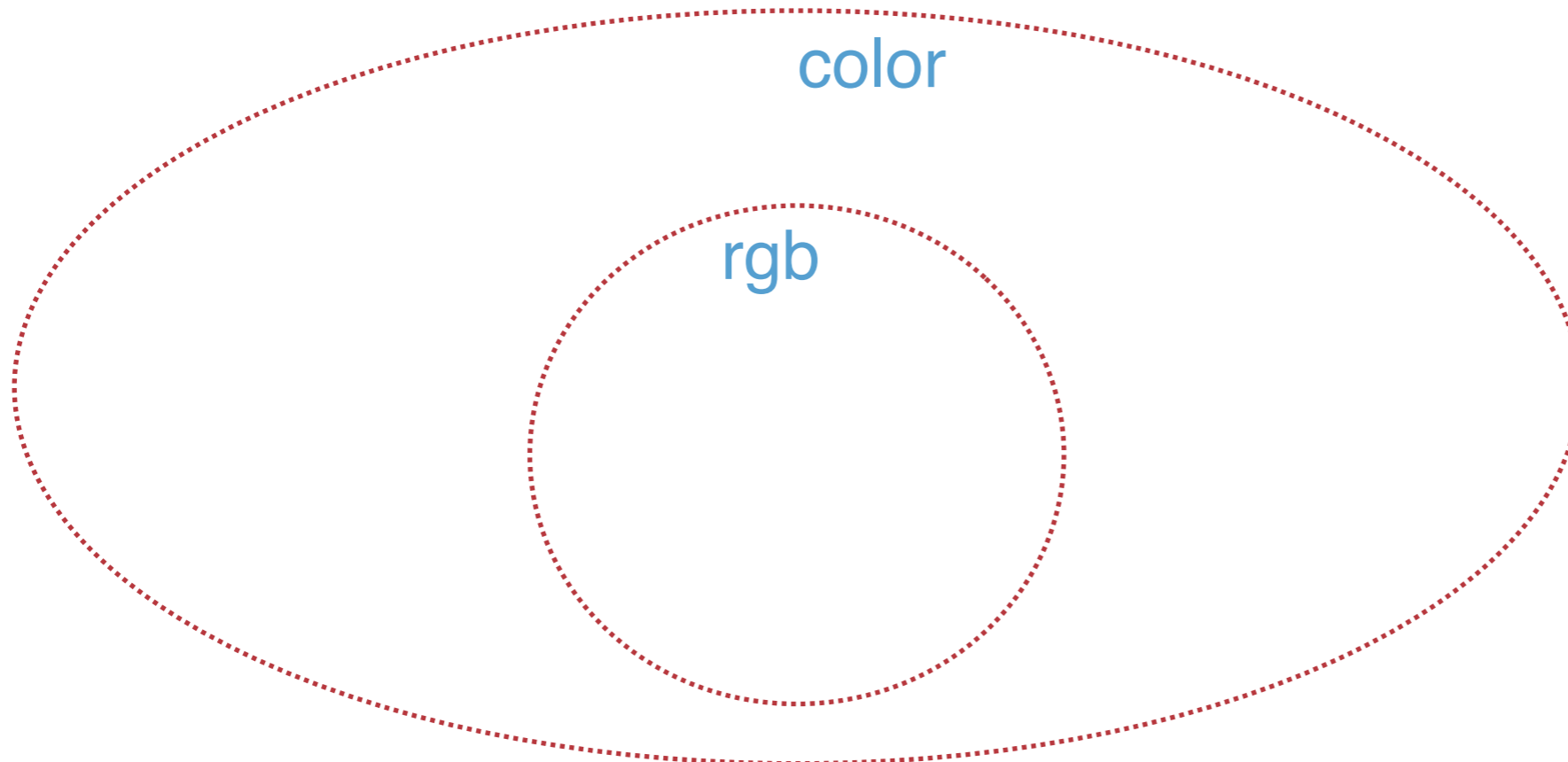
```
Definition pp (t : tag) : string :=  
  match t with  
  | h i => concat "<h" (natToString i) ">"  
  | a hr => concat "<a href=\"" hr "\">"  
  | _ => "<p>"  
  end.
```

So Far:

30

Inductive `rgb` : Type := | red | green | blue.

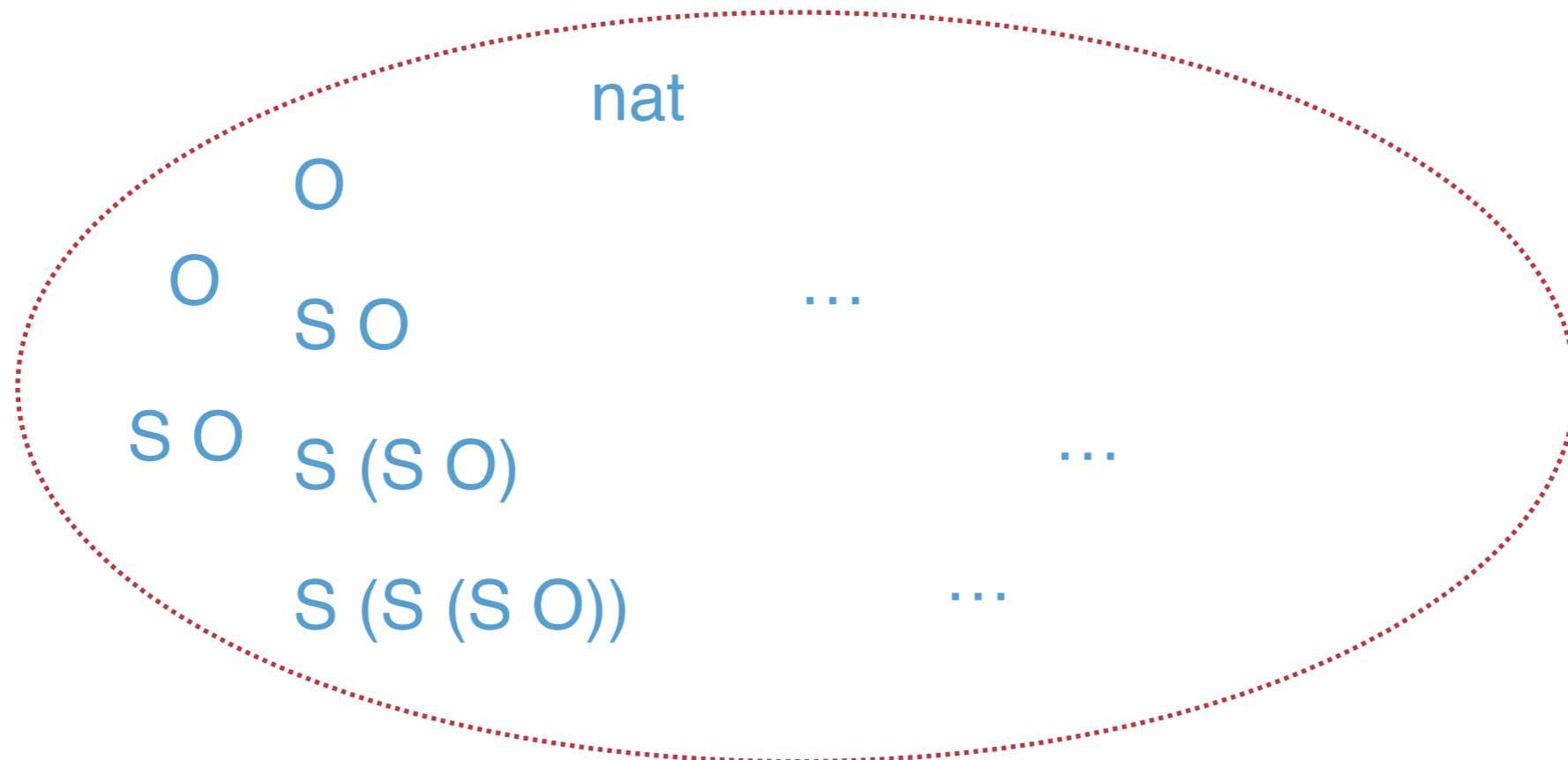
Inductive `color` := | black | white
| primary (p : rgb).



Natural Numbers

31

```
Inductive nat : Type :=  
  | O  
  | S (n : nat).
```



Functions

32

The *interpretation* of these constructors comes from how we use them to compute:

```
Definition pred (n : nat) : nat :=  
  match n with  
  | 0 => 0  
  | S m => m  
  end.
```


Recursion

33

Recursive functions use themselves in their definition

```
Fixpoint iseven (n : nat) : bool :=  
???
```

Recursion

34

Recursive functions use themselves in their definition

```
Fixpoint iseven (n : nat) : bool :=  
  match n with  
  | 0 => true  
  | S 0 => false  
  | S (S m) => iseven m  
  end.
```

Recursion

35

Recursive functions use themselves in their definition

```
Fixpoint plus (n m : nat) : nat :=  
  match n with  
  | 0 => m  
  | S n' => S (plus n' m)  
  end.  
Eval compute in (plus 2 3). (* = 5 *)
```

Recursion

36

Recursive functions use themselves in their definition

```
Fixpoint plus (n m : nat) : nat :=
```

```
  match n with
```

```
  | 0 => m
```

```
  | S n' => S (plus n' m)
```

```
  end.
```

```
Eval compute in (plus 2 3). (* = 5 *)
```

```
(* plus 2 5 = plus (S (S 0)) (S (S (S 0))) *)
```

Recursion

37

Recursive functions use themselves in their definition

```
Fixpoint plus (n m : nat) : nat :=  
  match n with  
  | 0 => m  
  | S n' => S (plus n' m)  
  end.  
Eval compute in (plus 2 3). (* = 5 *)  
(* plus (S (S O)) (S (S (S O))) =  
   S (plus (S O) (S (S (S O))))*)
```

Recursion

38

Recursive functions use themselves in their definition

```
Fixpoint plus (n m : nat) : nat :=  
  match n with  
  | 0 => m  
  | S n' => S (plus n' m)  
  end.  
Eval compute in (plus 2 3). (* = 5 *)  
(* S (plus (S 0) (S (S (S 0)))) =  
   S (S (plus 0 (S (S (S 0)))))*
```

Recursion

39

- ★ Recursive functions use themselves in their definition
- ★ Recall: functions need to be **total**
- ★ Coq requires functions be structurally recursive

```
Fixpoint plus (n m : nat) : nat :=
```

```
  match n with
```

```
  | 0 => m
```

```
  | S n' => S (plus n' m)
```

```
  end.
```

```
Eval compute in (plus 2 3). (* = 5 *)
```


```
(* S (S (plus 0 (S (S (S 0)))))) =  
   S (S (S (S (S 0)))) = 5 *)
```

Recursion

40

- ★ Recursive functions use themselves in their definition
- ★ Recall: functions need to be **total**
- ★ Coq requires functions be structurally recursive

```
Fixpoint plus (n m : nat) : nat :=  
  match n with  
  | 0 => m  
  | S n' => S (plus m n')  
  end.
```



Recursion

41

- ★ Recursive functions use themselves in their definition
- ★ Recall: functions need to be **total**
- ★ Coq requires functions be structurally recursive

```
Fixpoint mult (n m : nat) : nat :=  
  match n with  
  | 0 => 0  
  | S n' => plus m (mult n' m)  
  end.
```