Type Theory and Strong Functional Programming:

Adventures at the Edge of Reason

Aaron Stump
Computer Science
The University of Iowa

Boston College, February 13, 2024

Type Theory and Functional Programming

Type theory (TT)

- a language for computer-checked proofs
- intense interest currently, for formalized Math
- longstanding interest in CS:
  - verified compilers (Compcert, in Coq [award 2021])
  - now standard for Programming Languages theory

Functional programming (FP)

- Haskell, OCaml, Scala, Clojure, influencing many more
- also tightly connected to TT...

The Curry-Howard Isomorphism

Connection between Constructive Logic and FP:

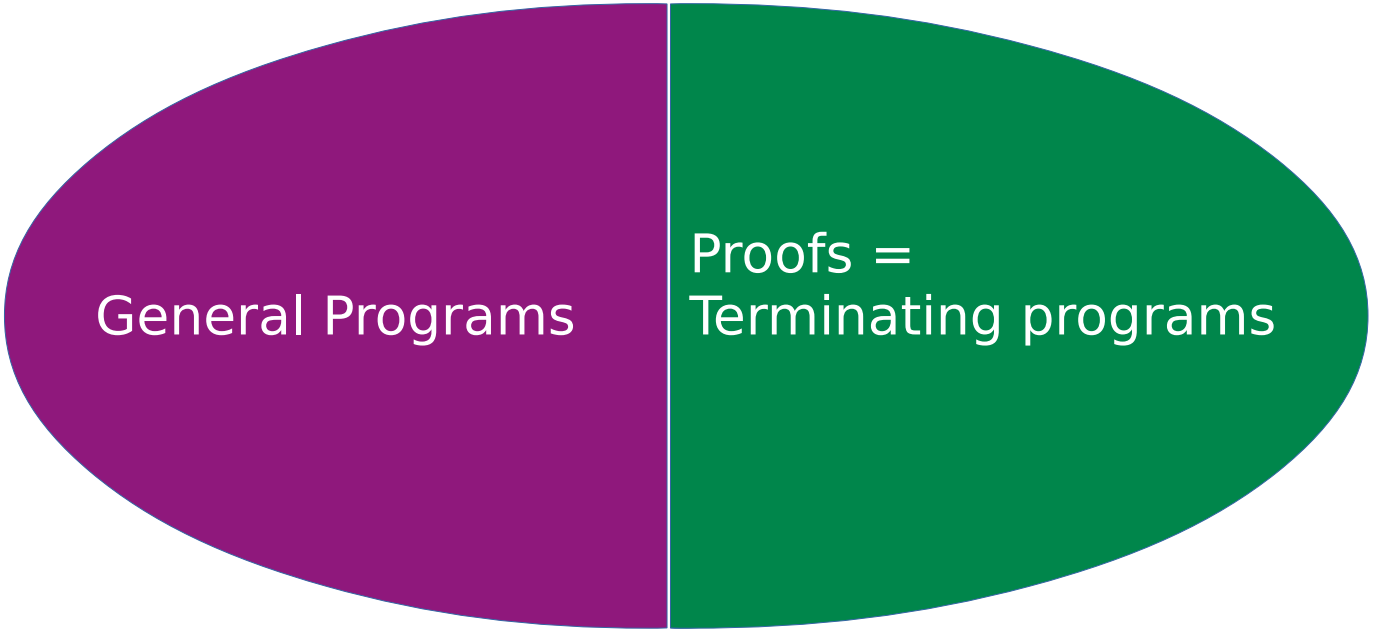
Formulas	≃	Types
Proofs	≃	Programs
Case splitting	≃	Pattern matching
Induction	≃	Terminating recursion

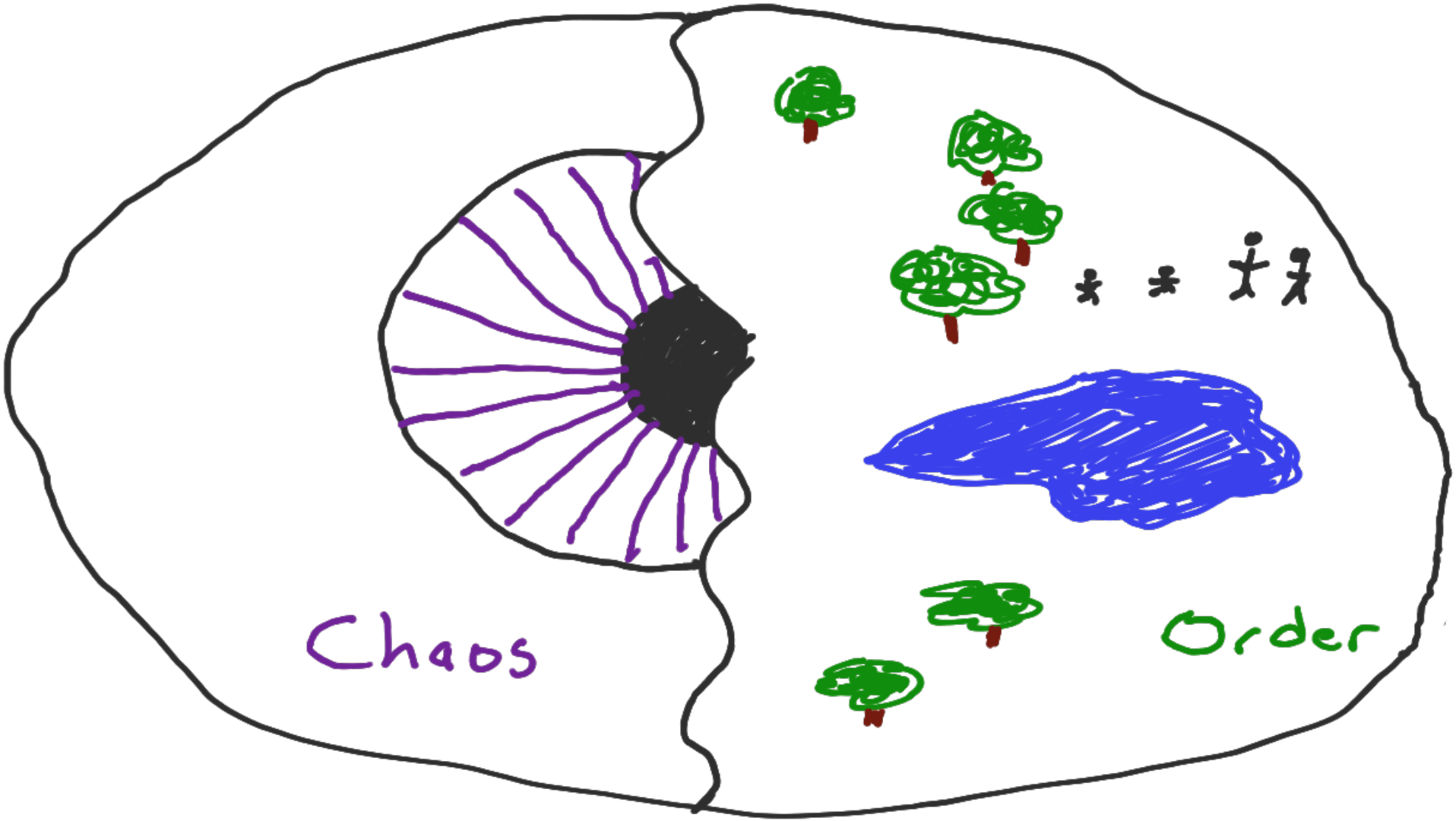
Type theory based on this connection

The Edge of Reason

Programs can diverge…

…but then these are not sound as proofs!




Chaos
Order

Outline

Past: Cedille and inductive lambda-encodings

Present: Strong functional programming with DCS

Future: More expressive type-based termination

Program termination for type theory and FP

Cedille and Inductive Lambda Encodings

Monotone recursive types and recursive data representations in Cedille.
Christopher Jenkins and Aaron Stump.
Mathematical Structures in Computer Science (MSCS), 31(6), pages 682-745, 2021.

Generic Derivation of Induction for Impredicative Encodings in Cedille.
Denis Firsov and Aaron Stump. 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP), pages 215-227, 2018.

From Realizability to Induction via Dependent Intersection.
Aaron Stump. Annals of Pure and Applied Logic (APAL), 169(7), pages 637-655, 2018.

Efficiency of Lambda-Encodings in Total Type Theory.
Aaron Stump and Peng Fu. Journal of Functional Programming (JFP), 26(e3), 2016.

and 8 more...

Pragmatics of foundations

Theorem provers based on set of axioms

How small?

How trustworthy?

How much code?

Inductive types

Types for tree-like data

```
data List a = Nil | Cons a (List a)
data Nat = Zero | Succ Nat
```

In TT, need induction principles, like

```
∀ P : Nat → ⋆.
  (∀ x : Nat. P x → P (Succ x)) →
  P Zero →
  ∀ x : Nat. P x
```

Complicated to state in general...

Inductive madhouse

$$\frac{\Gamma \vdash c : (I\ \vec{a}) \quad \Gamma \vdash Q : (\vec{x} : \vec{A})(I\ \vec{x}) \to s' \quad \Gamma \vdash f_i : C_i\{I, Q, \mathsf{Const}^{(\forall i = 1 \ldots n)}}{\Gamma \vdash \mathsf{Elim}(c, Q)\{f_1 | \ldots | f_n\} : (Q\ \vec{a}\ c)}$$

$$(\text{W-Elim}) \quad \frac{A \equiv (\vec{x} : \vec{A})\mathrm{Set} \qquad I = \text{'}}{\Gamma \vdash Q : (\vec{x} \cdot \text{'}}$$

Elimination (definitio

$$\frac{\begin{array}{l} r = n + m \\ x_1 \ldots x_n \notin \mathsf{FV}(|P|) \\ \mathsf{getArgs}(t') = [w_1, \ldots, w_m] \\ \mathsf{buildCtx}(t) = [y_1 : t''_1, \ldots, y_n : t''_m] \\ \mathsf{cut}([y_1 : t''_1, \ldots, y_n : t''_m], \mathsf{buildCtx}(\mathsf{getCType}(t', C, \Delta))) = [x_1 : t'_1, \ldots, x_n : t'_n] \\ \Delta, \Gamma \vdash^{PB} R\ t_1\ t'\ y\ (l - \{\mathsf{C} : getCType(t', C, \Delta)\}) : P \\ \Delta, \Gamma, x_1 : [w_1/y_1]t'_1, \ldots, x_n : [w_m/y_m]t'_n, y : t_1 = (\mathsf{C}\ w'_{1\varepsilon_1} \ldots w'_{r\varepsilon_n}) \vdash p'' : P \\ \Delta, \Gamma \vdash^{PB} (\mathsf{C}\ x_{1\varepsilon'_1} \ldots x_{n\varepsilon'_n} \Rightarrow p'' \mid R)\ t_1\ t'\ y\ l : P \end{array}}{\Delta, \Gamma \vdash^{PB} (\mathsf{C}\ x_{1\varepsilon'_1} \ldots x_{n\varepsilon'_n} \Rightarrow p'' \mid R)\ t_1\ t'\ y\ l\ d\ t)} \text{ PB\_Branch}$$

$$\frac{\begin{array}{l} \mathcal{B} : \mathsf{Type}_{\ell_2} \\ \ldots\mathsf{e}_{\ell_1} \text{ where } \{\ \overline{d_i \text{ of } \Delta_i}^{\,i \in 1 \ldots k}\ \} \in \Gamma \\ \ldots\kappa, \quad \Gamma, [\overline{a_i}^{\,i}/\Delta]\Delta_i\,\theta, y :^{\mathsf{L}} a = d_i\,\Delta_i \vdash^{\theta} b_i : B \end{array}}{\Gamma \vdash^{\theta} \mathsf{case}\ a\ \mathsf{of}\ \{\ \overline{d_i\,\Delta_i \Rightarrow b_i}^{\,i \in 1 \ldots k}\ \} : B}\ \text{TCase}$$

Disadvantages

Inductive types complicated to specify

- increases trusted computing base
- more work to prove theory sound

A particular *class* of inductive types chosen

- new kinds of inductive type still being devised
- so old soundness proofs obsolete

Couldn’t you derive these somehow instead?

Lambda encodings

Represent data as functions

Original encoding due to Alonzo Church

- numbers encoded as *iterators*
- 2 encoded as function
  - Inputs: f and x
  - Output: (f (f x))
  - 2 := λ f . λ x . f (f x)
- can define usual operations by iteration

CNat := ∀ X. (X → X) → X → X

Inductive lambda-encodings?

H. Geuvers proved induction not derivable in pure type theory [Geuvers 2001]

No matter how Nat defined with constructors Zero and Succ, there is no term t with

- t : ∀ P : Nat → ⋆.

  (∀ x : Nat. P x → P (Succ x)) →

  P Zero →

  ∀ x : Nat. P x

The path they trod

This was already believed in the 1980s

Researchers added inductive types as primitives to the pure theory

Coq, Lean, Agda all use this approach

Cedille

Constructive type theory with tiny core

- core checker around 1kloc Haskell
- inductive types not in the core
  - they are in the core for Coq, Agda, Lean

Translates from rich surface language to core

- inductive types are lambda-encoded
- induction principles derived!

A different way

Geuvers's Theorem requires adding something to pure type theory to get induction

Cedille is based on the discovery that adding just three simple primitives is enough

- a primitive equality type on untyped terms
- "implicit products"
- "dependent intersections"

An insight of Leivant

Daniel Leivant: proofs of induction for n are isomorphic to lambda-encoded n

This is the key to Cedille's approach

Let's dig in...

Induction for n : CNat

Let’s define Ind n as

∀ P : CNat → ⋆.

(∀ x : CNat. P x → P (Succ x)) →

P Zero →

P n

What is a proof of Ind 2?

∀ P : CNat → ⋆.

(∀ x : CNat. P x → P (Succ x)) →

P Zero →

P 2

Assume

- P : CNat → ⋆
- step : ∀ x : CNat. P x → P (Succ x)
- base : P Zero

Apply step twice…

What is a proof of Ind 2?

Assuming:

- P : CNat → ⋆ — λ P .
- step : ∀ x : CNat. P x → P (Succ x) — λ step .
- base : P Zero — λ base .

Have: step 1 (step 0 base)

- base : P 0
- step 0 base : P 1
- step 1 (step 0 base) : P 2

What is a proof of Ind 2?

λ P . λ step . λ base. step 1 (step 0 base)

renaming

λ P . λ f . λ x. f 1 (f 0 x)

eliding

λ f . λ x. f (f x)

Tada! It's 2!

So what is an inductive Nat?

A Nat is

- a Church-encoded n
- that also proves Ind n

Can we say this in type theory?

Dependent intersections

There's a type for that! [Kopylov 2003]

x : A ∩ B

Like an intersection A ∩ B, but with x bound in B

Type for values v which

- have type A and also
- have type B v

Nat with dependent intersection

Nat := n : CNat ∩ Ind

2 : Nat means

- 2 : CNat, and also — *2 is a computational nat*
- 2 : Ind 2 — *2 proves induction for 2*

This requires erasure

- Need to erase λ P . λ f . λ x. f 1 (f 0 x)
- to get λ f . λ x. f (f x)

Use in Cedille

Ind: induction for CNat

From this, can derive induction for Nat

Datatypes supported in usual syntax

- Translated to inductive lambda-encodings

Checker in under 1kloc Haskell

Summary

New way to define inductive datatypes in type theory

- very small core theory
- hence, small proof checker
  - Just inductive types takes more code for Lean, Coq, Agda!

Lambda encodings

Typed using dependent intersections

Strong functional programming with DCS

A Type-Based Approach to Divide-and-Conquer Recursion in Coq.
Pedro Abreu, Benjamin Delaware, Alex Hubers, Christa Jenkins, J. Garrett Morris, Aaron Stump. Proceedings of the ACM on Programming Languages (PACMPL), volume 7, number POPL, January 2023, pages 61-90, 2023.

Strong functional pearl: Harper's regular-expression matcher in Cedille. Aaron Stump, Christopher Jenkins, Stephan Spahn, and Colin McDonald. Proceedings of the ACM on Programming Languages (PACMPL), volume 2, number ICFP (International Conference on Functional Programming), pages 122:1 - 122:25, 2020.

https://gitlab.com/astump97/dcs

St. Philip Neri

Born 1515 in Florence
Arrives in Rome age 18
Befriends St. Ignatius around 1544
Becomes a priest in 1551
Founds the Oratory
congregation of priests under obedience
Dies 1595

Known for emphasizing humility through *mortification*

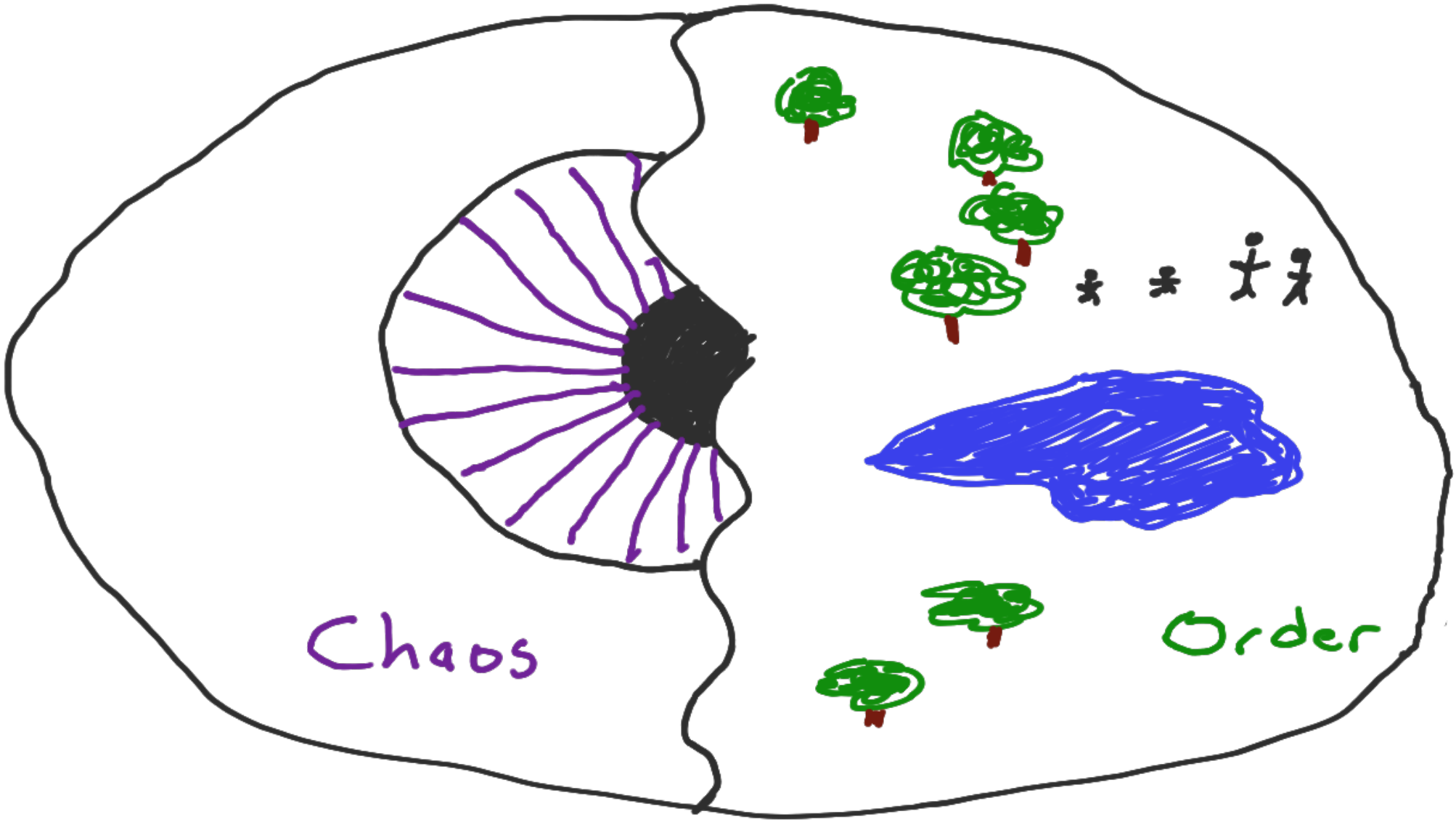
Eccentric behavior...
Hiding true sanctity.

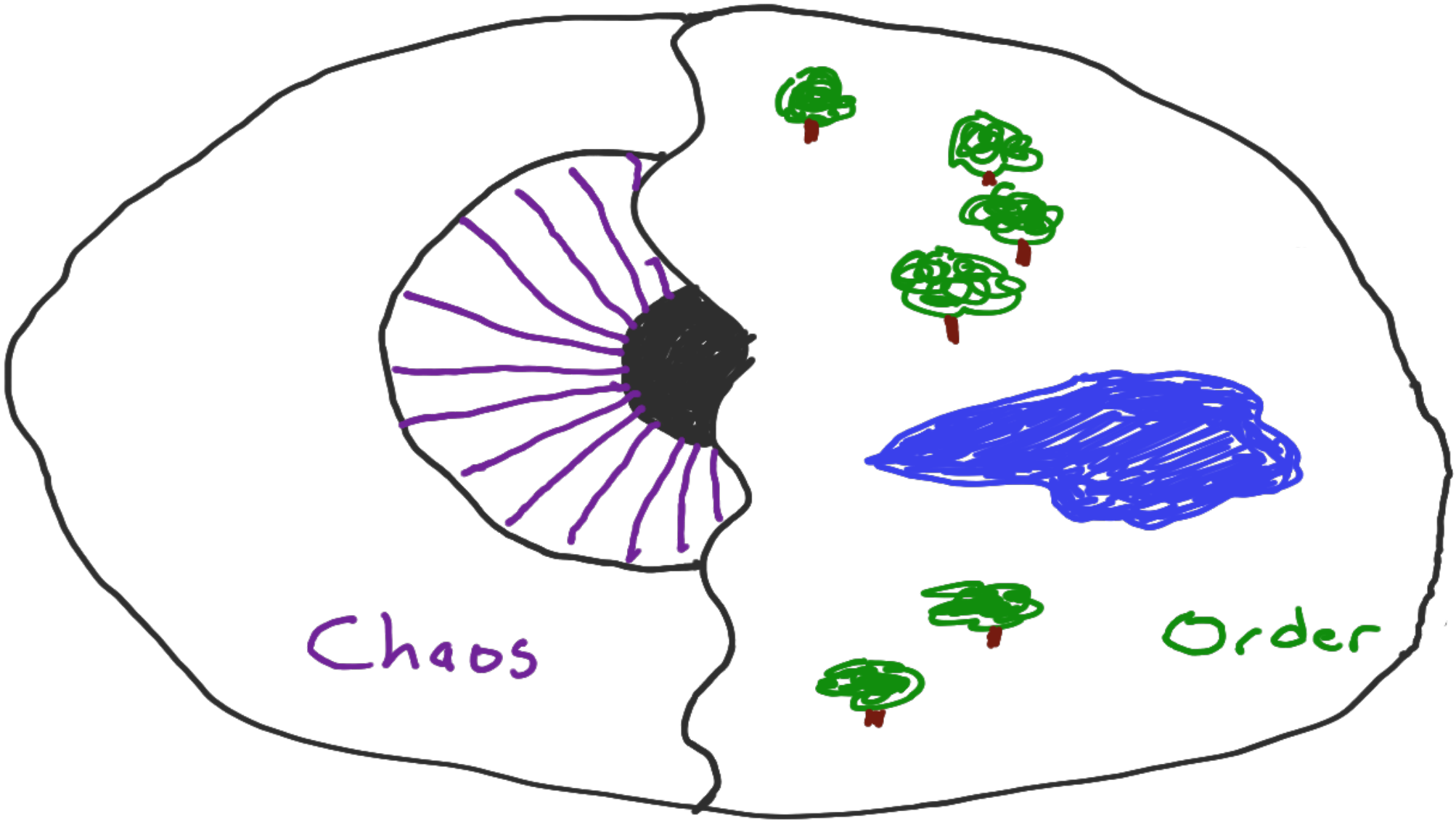
What is strong FP?

Type Theory minus the fancy types!

FP with static check for uniform termination for all functions

“Elementary Strong Functional Programming”, David Turner, LNCS 1022:1-13, 1995


Chaos
Order


Chaos
Order

Structural termination

Standard approach (Coq, Agda, Lean)

- check code directly (syntactic) for
- structural decrease at recursive call sites

In Agda:

```
length : List A → Nat
length Nil = 0
length (Cons x xs) = 1 + length xs
```

```
Cons x xs > xs
```

Issues with structural termination

Cannot recurse on output of function

- defining division by repeated subtraction, cannot recurse on x – y

So refactoring can break termination

- code that passed may fail after refactoring

Cannot call constructor and then recurse

Divide-and-conquer recursion

Example: mergesort

- split list (length > 1) into two sublists
- recursively sort
- merge results

Embarrassing!

With structural recursion:

- splitting builds a new list, so cannot recurse

Mergesort cannot be written in Coq/Lean/Agda!

- must resort to tricks like recursing on length of list

DCS

New programming language for strong FP

Typing enforces termination

- supports refactoring
- divide-and-conquer recursion
- soundness proven in Coq [POPL 2023]
- general recursion (to be) supported through *monads*

Central design ideas:

- Commitment to subtyping
- *Algebraic* approach to datatypes

Signature functors

Datatypes from *signature functors F*

- Show one layer of datatype structure
- Look like the datatype, but
- recursive occurrences abstracted away

List datatype:

```
data List A = Nil | Cons A (List A)
```

Its signature functor:

```
data ListF A X = NilF | ConsF A X
```

Datatypes in DCS

DCS datatype declaration:

δ List A = Nil | Cons A (List A)

Introduces both datatype, signature functor with same names:

δ List A = Nil | Cons A (List A)

δ List A X = Nil | Cons A X

Datatypes as fixed-points

If D is a datatype with sig functor F:

D ≅ F D

So in DCS:

D ≅ D D

With subtyping:

- D <: D D
- D D <: D

Computation via algebras

(List A)-algebra with carrier X has type

alg : (List A X) → X

Example:

lengthAlg : (List A Nat) → Nat

lengthAlg Nil = 0

lengthAlg (Cons x n) = 1 + n

From this, obtain ⦇lengthAlg⦈ : List A → Nat

Mendler algebras

Instead of

$$\mathsf{alg} : F\ X \to X$$

Mendler proposed [Mendler 1991]

$$\mathsf{alg} : \forall\ R\ .\ (R \to X) \to F\ R \to X$$

Now an algebra gets an F R, way to turn Rs into Xs.

$$\mathsf{Alg}_F\ X = \forall\ R\ .\ (R \to X) \to F\ R \to X$$

Mendler algebra for length

length : $\mathbf{Alg}_{\mathbf{ListF\ A}}$ Nat

length _ Nil = 0

length f (Cons x r) = 1 + f r

- r : R
- f : R → Nat

If we just wrote length instead of f,

it would look like a recursive call

1 + length r

DCS algebras

Syntax: ω f(xs) : C . t

- f for making recursive calls in body t
- xs is input of type F R (sig. functor F)
- R as in Mendler algebras
- C is the carrier of the algebra
- f : R → ...
- R ~ F means “R from an F-alg”

Type: F ⇒ C

length

```
length A : List A ⇒ K Nat =
  ω length(xs) : K Nat .
    γ xs {
      Nil → Zero
    | Cons x xs' → Succ (length xs')
    }
```

In body of length, we have:

- abstract type R
- xs : List A R
- xs' : R
- length : R → Nat

So `length xs'` **well typed**

Subsidiary recursions

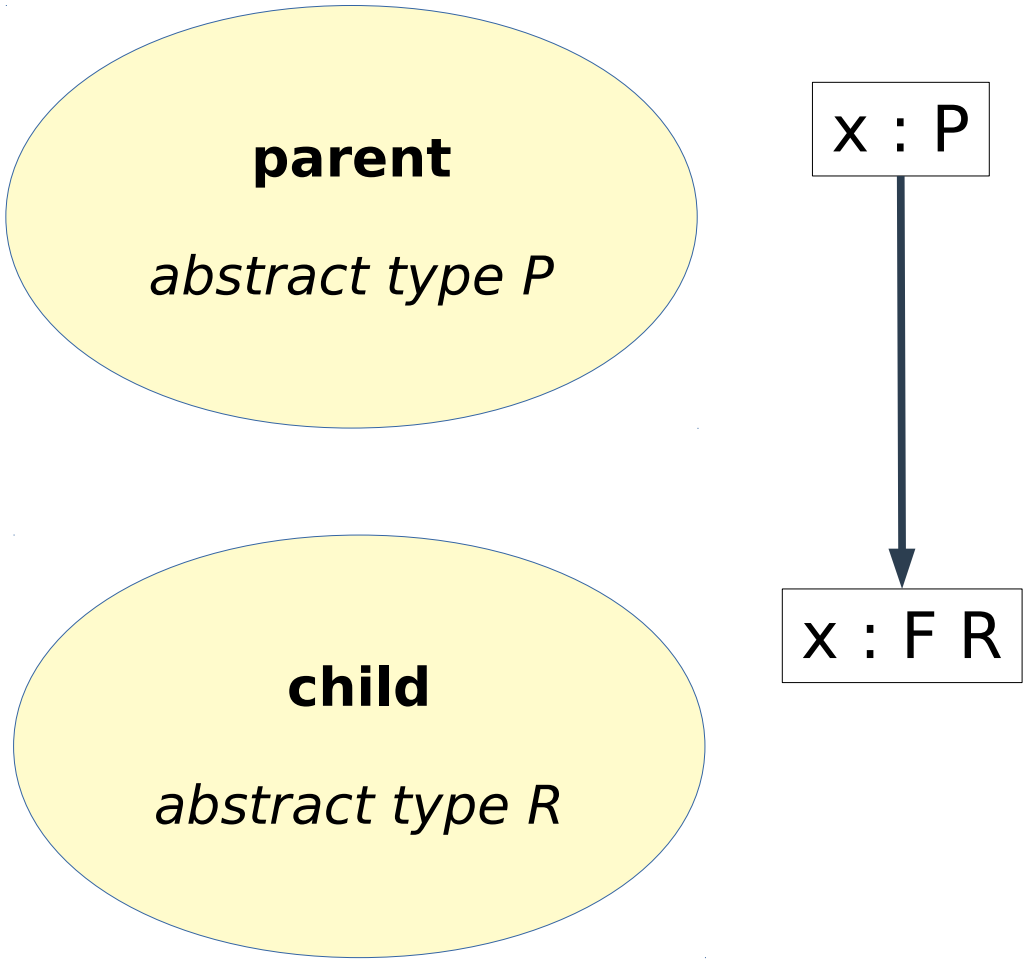
Suppose parent recursion invokes child

- Parent has abstract type P
- Child has abstract type R

Idea: child algebra can reference P

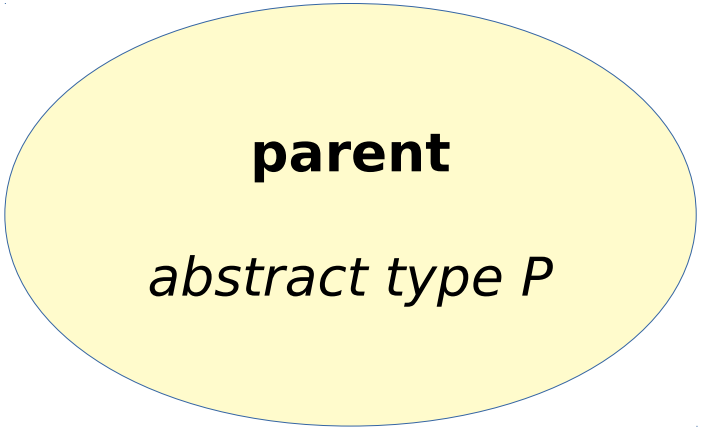
- to build data that the parent can recurse on

Subsidiary recursion and subtyping

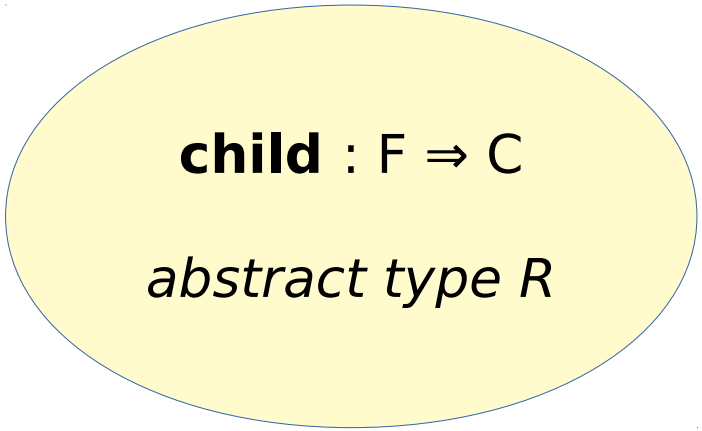


F R <: P
R <: P

Subsidiary invocation

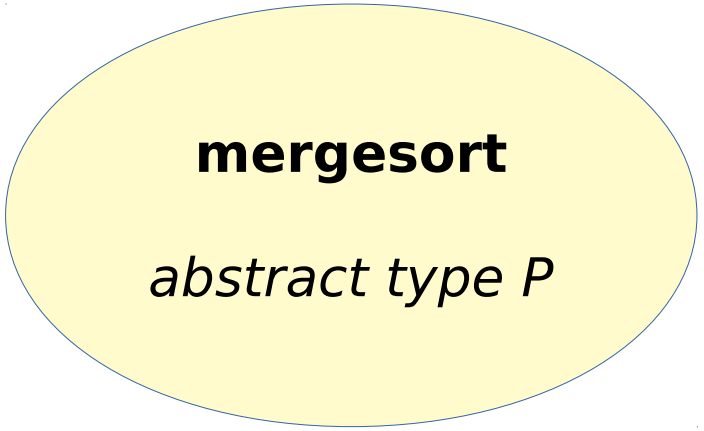


In parent:
child : P → C P, where P ~ F

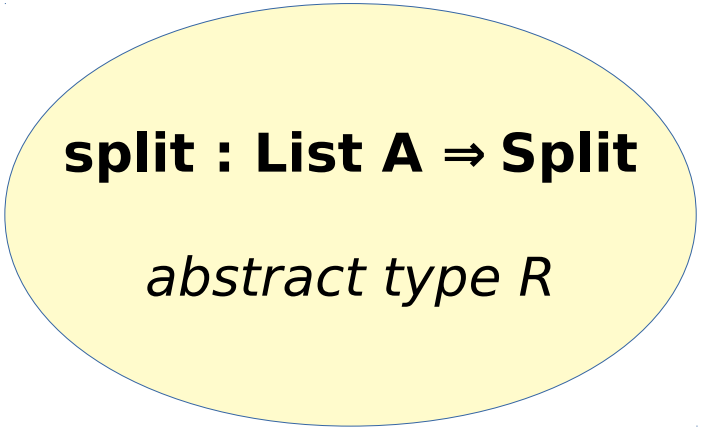


In child:
child : R → C R

Algebras for mergesort



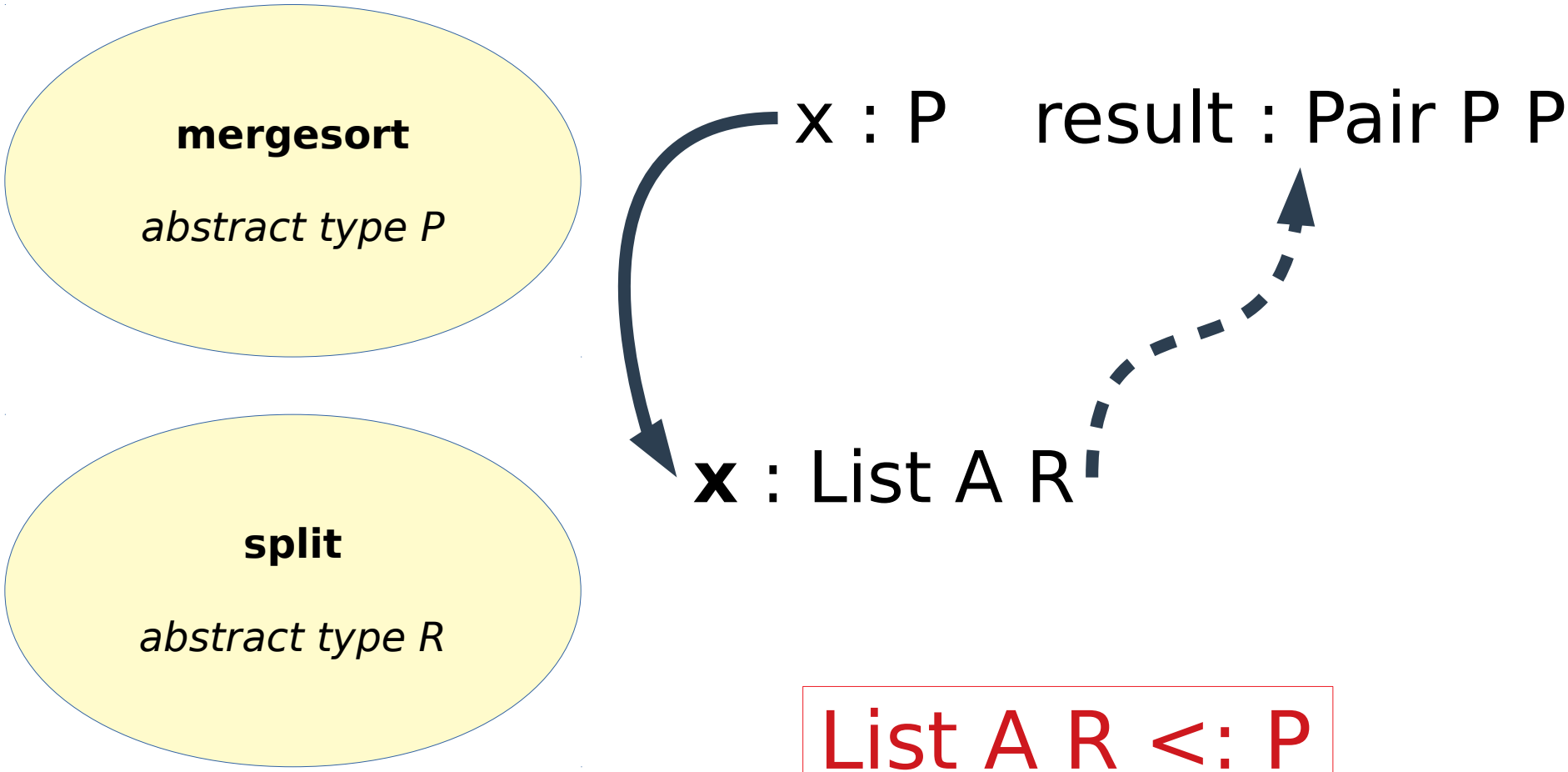
In mergesort:
split : P → Pair P P



In split:
split : R → Pair R R

where Split P = Pair P P

How does subtyping fit in?



List A R <: P

DCS Demo

More expressive type-based termination

John Ronald Reuel Tolkien

John Ronald Philip Reuel Tolkien

From lists to trees

DCS’s current interface works great for lists

- List A R <: P
- child recursion can build a list, and parent can recurse

Less well for trees

- Tree A R <: P
- If both subtrees of type R, parent can recurse
- But what if one subtree smaller, other the same?

Goal: recurse when one part smaller, others unchanged

Multi-argument signature functors

Instead of Tree A X = Leaf | Node A X X

Would like to have something more fine-grained:

Tree A X1 X2 = Leaf | Node A X1 X2

Then can have Tree A L1 R2 <: P Tree A R1 L2 <: P

Semantics: well-founded structural order

Express structural decrease explicitly,

as a well-founded ordering:

- ignore data stored in the structure
- Node x l r < Node x’ l’ r, if l < l’

Instead of comparing (ordinal) sizes of data, compare the data themselves in this ordering

Have a formalization in Agda for any *algebraic* datatype

Goal: improve DCS interface for types beyond list

General future directions

Interplay between logic and programming

- type theory
- strong functional programming

Advances in core CS

- new type theories
  - more expressive, better abstractions
- improve programming through typing
  - use subtyping to infer coercions, reduce boilerplate code
  - applying to abstractions from Haskell (Functor, Monad)

Advance tech for computer-checked proofs

Conclusion

Past: Cedille and inductive lambda-encodings

Present: Strong functional programming with DCS

Future: More expressive type-based termination

Program termination for type theory and FP

AMDG

Five Canonized in 1622

Type Theory and Strong Functional Programming:

Adventures at the Edge of Reason

Aaron Stump
Computer Science
The University of Iowa

Boston College, February 13, 2024

Some further reading

Elementary Strong Functional Programming. David Turner, Functional Programming Languages in Education, LNCS 1022, 1-13, 1995

A Type-Based Approach to Divide-and-Conquer Recursion in Coq. Proc. ACM Program. Lang. 7(POPL), 61-90, 2023

Data types à la carte. Wouter Swierstra, Journal of Functional Programming, 18(4), 423-436, 2008

Polymorphic subtyping in O'Haskell. Johan Nordlander, Science of Computer Programming, 43(2), 93-127, 2002

https://www.wordonfire.org/articles/fellows/whats-in-a-name-tolkiens-st-philip-neri-connection/