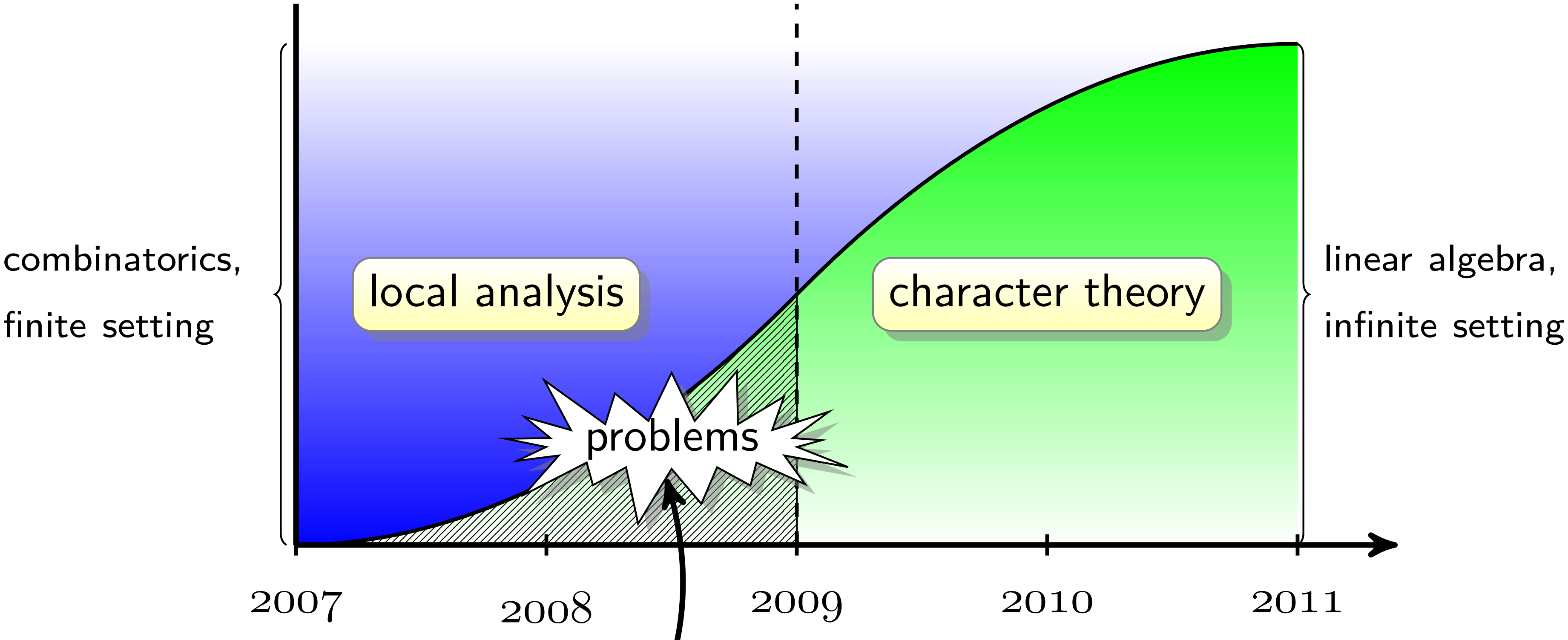
Packaging Mathematical Structures

F. Garillot[1], G. Gonthier[2],
A. Mahboubi[3], L. Rideau[4]

[1]: Microsoft Research - INRIA Joint Centre
[2]: Microsoft Research
[3]: INRIA Saclay-Île de France
[4]: INRIA Sophia-Antipolis

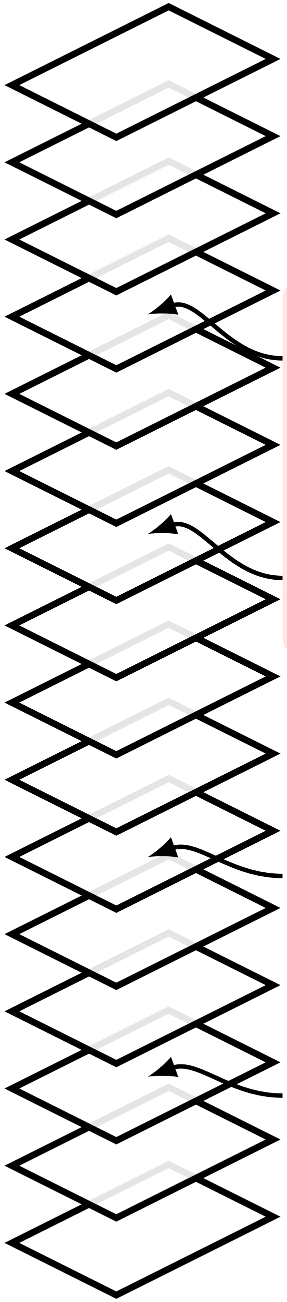
Formalizing Finite Group Algebra

the Feit-Thompson proof



This paper presents our solutions using Coq's dependent records, coercions, type inference.

Packaging Mathematical Structures



Composable mathematical structures

Our algebraic hierarchy

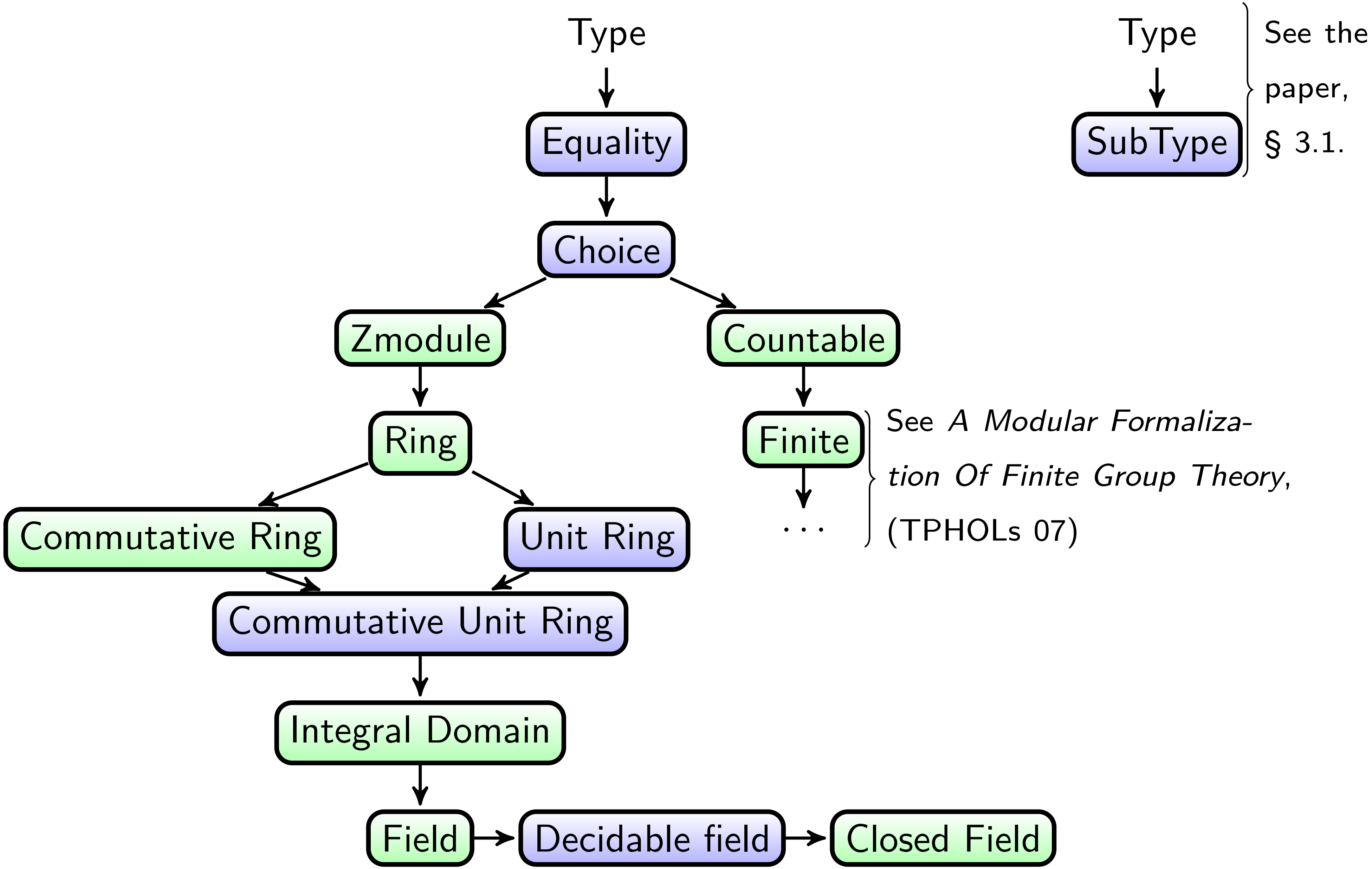
This talk

Hierarchy details,
a nice alternative to Σ-types

Two proof examples, in depth

Our paper

Our Algebraic Hierarchy



Type

Equality

Choice

Zmodule

Countable

Ring

Finite

…

Commutative Ring

Unit Ring

Commutative Unit Ring

Integral Domain

Field

Decidable field

Closed Field

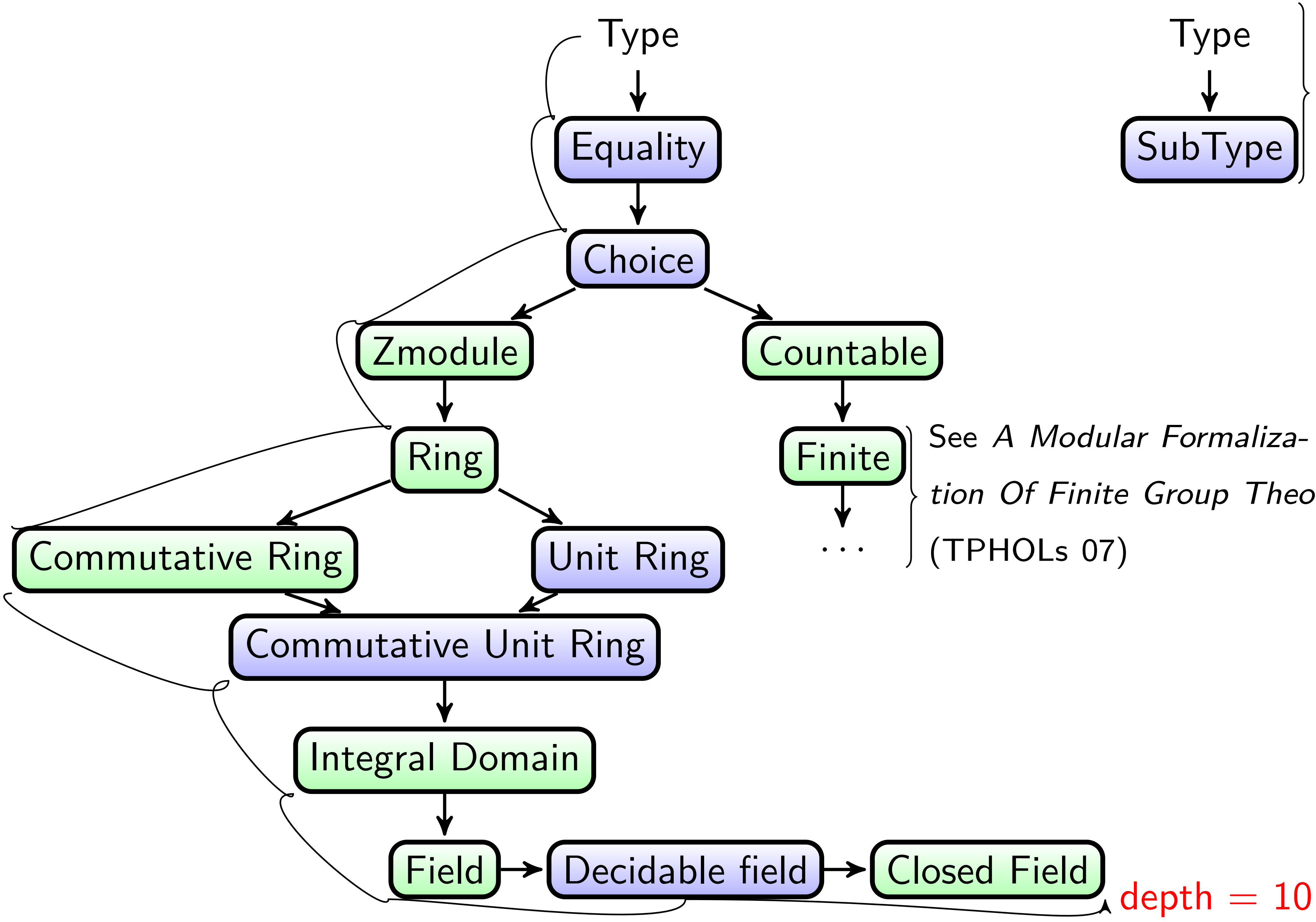
Type

SubType

See the paper, § 3.1.

See *A Modular Formalization Of Finite Group Theory*, (TPHOLs 07)

Our Algebraic Hierarchy

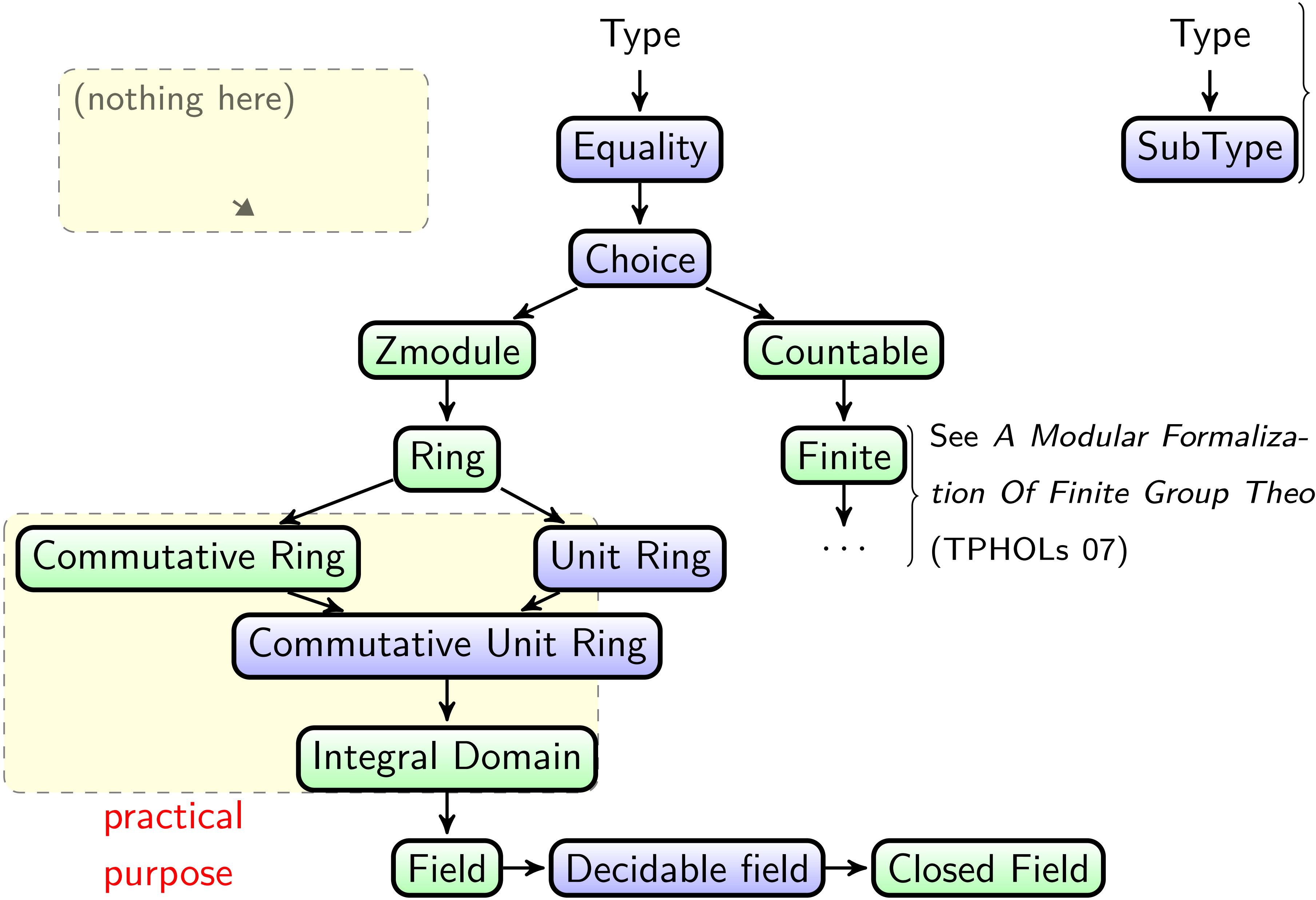


See the paper, § 3.1.

See *A Modular Formalization Of Finite Group Theory*, (TPHOLs 07)

depth = 10

Our Algebraic Hierarchy



(nothing here)

Type

Equality

Choice

Zmodule

Countable

Ring

Finite

. . .

Commutative Ring

Unit Ring

Commutative Unit Ring

Integral Domain

Field

Decidable field

Closed Field

practical purpose

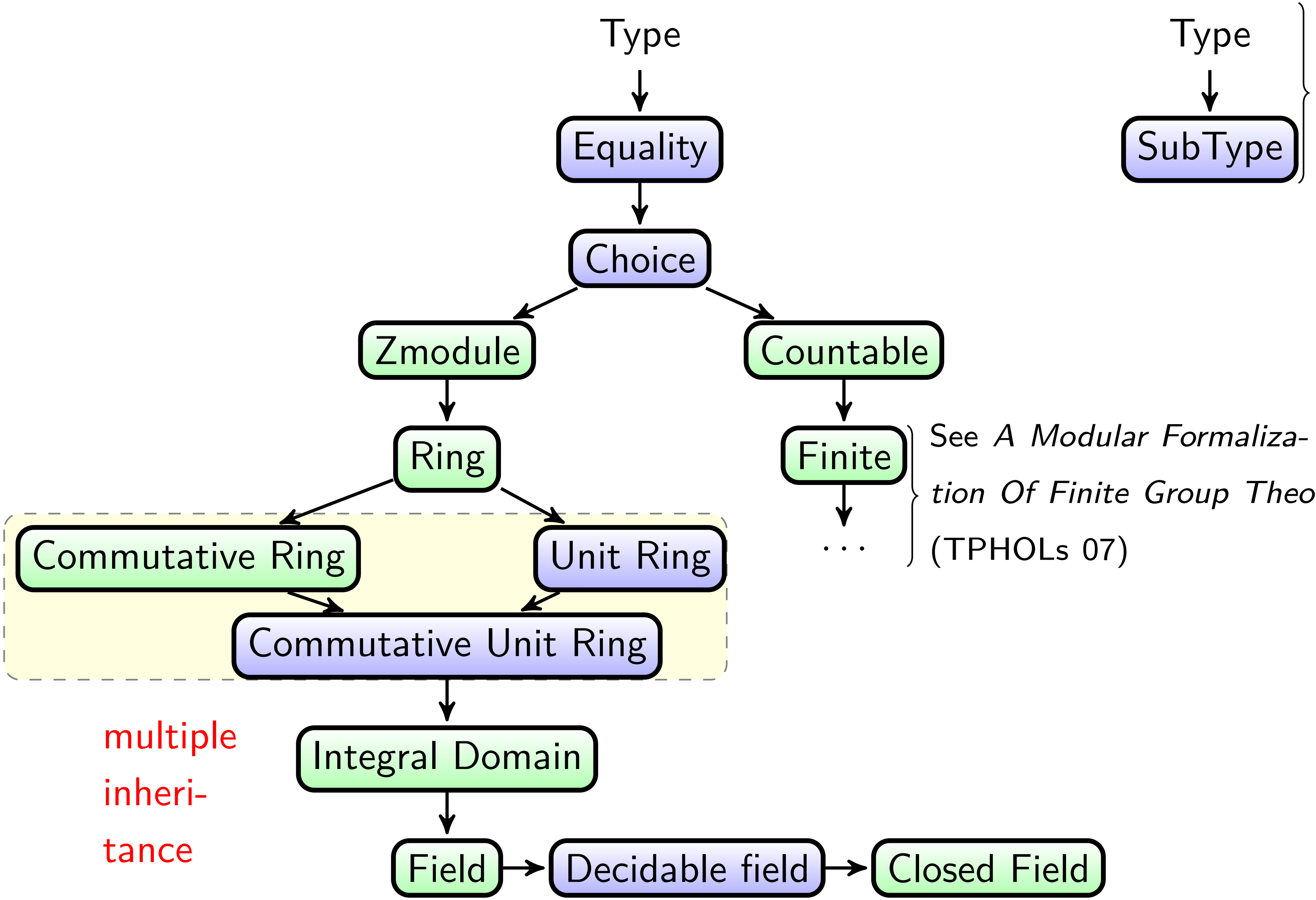
Type

SubType

See the paper, § 3.1.

See *A Modular Formalization Of Finite Group Theory*, (TPHOLs 07)

Our Algebraic Hierarchy



Type

Equality

Choice

Zmodule

Countable

Ring

Finite

. . .

Commutative Ring

Unit Ring

Commutative Unit Ring

Integral Domain

Field

Decidable field

Closed Field

Type

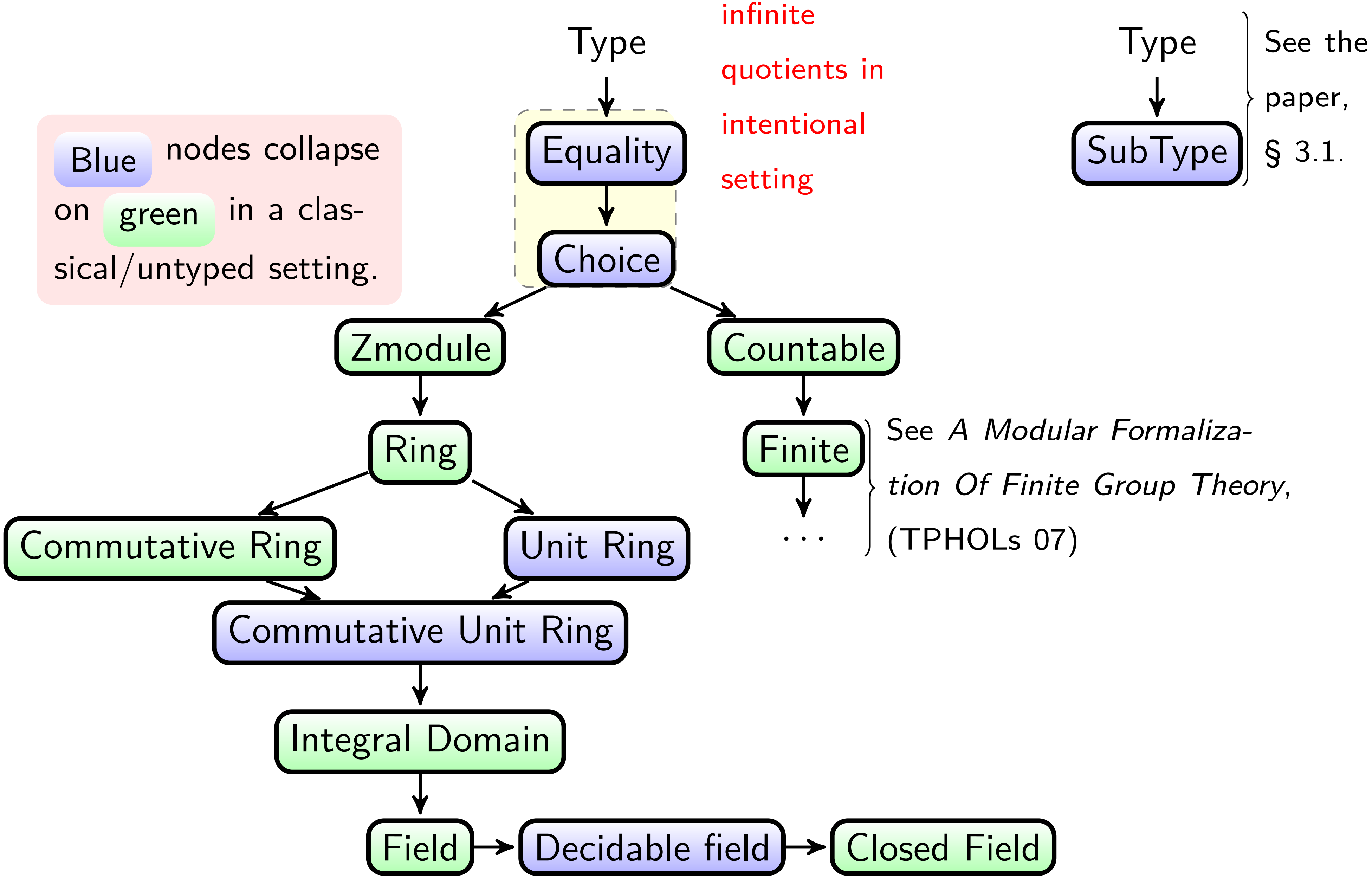
SubType

See the paper, § 3.1.

See *A Modular Formalization Of Finite Group Theory*, (TPHOLs 07)

multiple inheritance

Our Algebraic Hierarchy



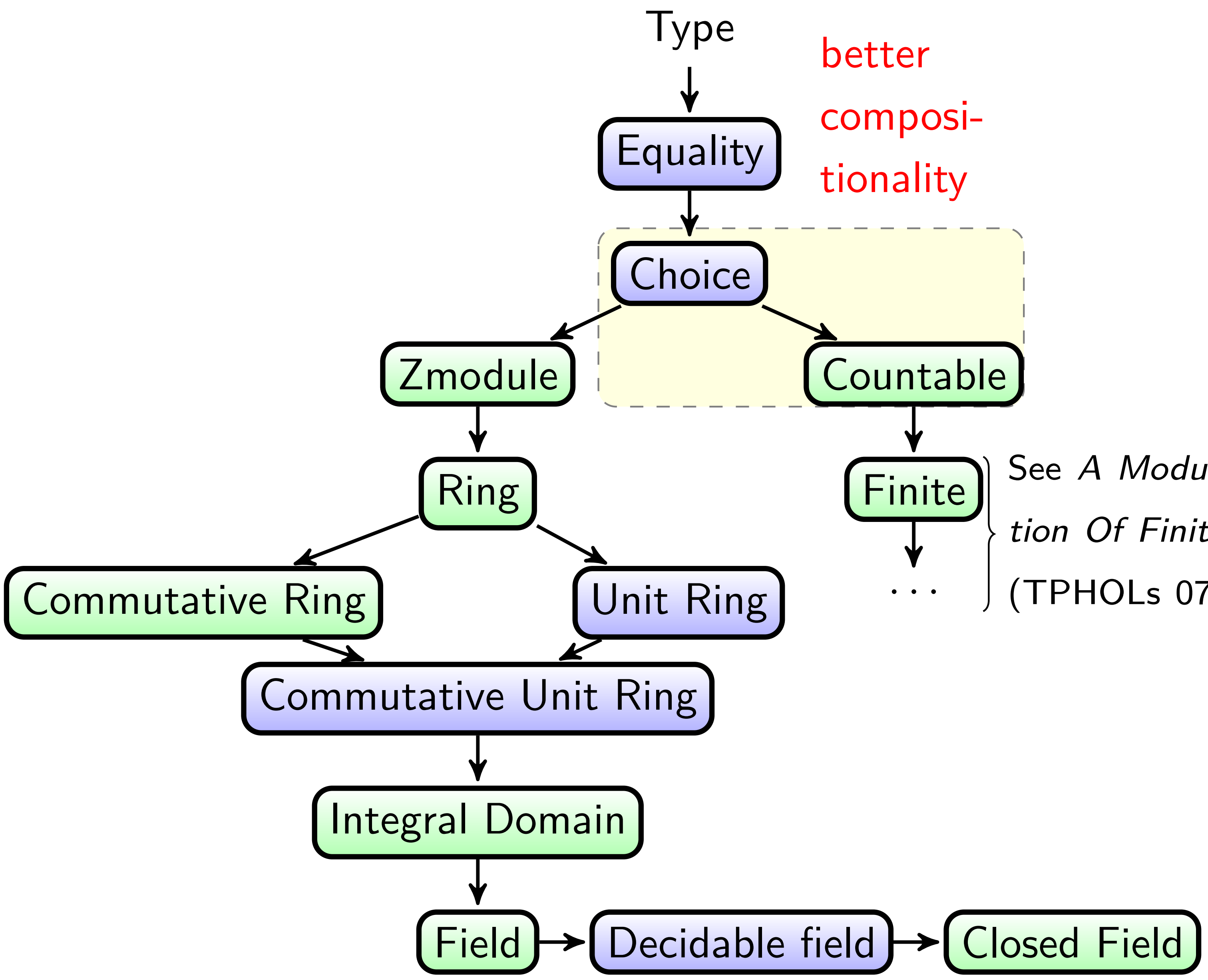
Blue nodes collapse on green in a classical/untyped setting.

infinite quotients in intentional setting

See the paper, § 3.1.

See *A Modular Formalization Of Finite Group Theory*, (TPHOLs 07)

Our Algebraic Hierarchy



better compositionality

See the paper, § 3.1.

See *A Modular Formalization Of Finite Group Theory*, (TPHOLs 07)

Our Algebraic Hierarchy

Our Algebraic Hierarchy

(1) This hierarchy is populated by ground types, e.g.: $\mathbb{Z}/n\mathbb{Z}, \mathbb{F}_p$

Our Algebraic Hierarchy

(1) This hierarchy is populated by ground types, e.g.: $\mathbb{Z}/n\mathbb{Z}, \mathbb{F}_p$

(2) ... and parametric types, e.g.: matrix, polynomials.

Our Algebraic Hierarchy

(1) This hierarchy is populated by ground types, e.g.: $\mathbb{Z}/n\mathbb{Z}, \mathbb{F}_p$

(2) ... and parametric types, e.g.: matrix, polynomials.

(3) Very little computational content or theory in a `Ring`

Our Algebraic Hierarchy

(1) This hierarchy is populated by ground types, e.g.: $\mathbb{Z}/n\mathbb{Z}, \mathbb{F}_p$

(2) . . . and parametric types, e.g.: matrix, polynomials.

(3) Very little computational content or theory in a `Ring`

(4) but it is a nice package that occurs several times in lemma statements.

Our Algebraic Hierarchy

(1) This hierarchy is populated by ground types, e.g.: $\mathbb{Z}/n\mathbb{Z}, \mathbb{F}_p$

(2) … and parametric types, e.g.: matrix, polynomials.

(3) Very little computational content or theory in a `Ring`

(4) but it is a nice package that occurs several times in lemma statements.

(5)

$$\left[\ \text{Components, objects}\qquad\qquad\right] \neq \left[\ \text{APIs, interfaces}\qquad\qquad\qquad\right]$$

Our Algebraic Hierarchy

(1) This hierarchy is populated by ground types, e.g.: $\mathbb{Z}/n\mathbb{Z}, \mathbb{F}_p$

(2) . . . and parametric types, e.g.: matrix, polynomials.

(3) Very little computational content or theory in a `Ring`

(4) but it is a nice package that occurs several times in lemma statements.

(5)

$$\left[\ \text{Components, objects}\qquad\qquad\qquad \right] \neq \left[\ \text{APIs, interfaces}\qquad\qquad\qquad\qquad \right]$$

(6) algebraic properties $\triangleq$ interface programming

Representing objects and structures

We want to manipulate objects:

Representing objects and structures

We want to manipulate objects:

(Cayley-Hamilton)

$$\left[\begin{array}{c} P(A) = 0 \\ \text{where } P(X) = det(X \bullet I_n - A) \end{array}\right]$$

Representing objects and structures

We want to manipulate objects:

(Cayley-Hamilton)

$$\left[\begin{array}{c} P(A) = 0 \\ \text{where } P(X) = det(X \bullet I_n - A) \end{array}\right]$$

But we do not want to specify the corresponding structure:

$$\left[\begin{array}{l} \text{The } \mathtt{Ring} \text{ of polynomials over} \\ \text{the } \mathtt{Ring} \text{ of matrices} \\ \text{over a general } \mathtt{Commutative\ Ring}. \end{array}\right]$$

A Type class

`ZModule (M:Type)`

- a constant `zero : M`
- an operation : `add : M-> M -> M`
- axiom(s) verified by `add` on `M`

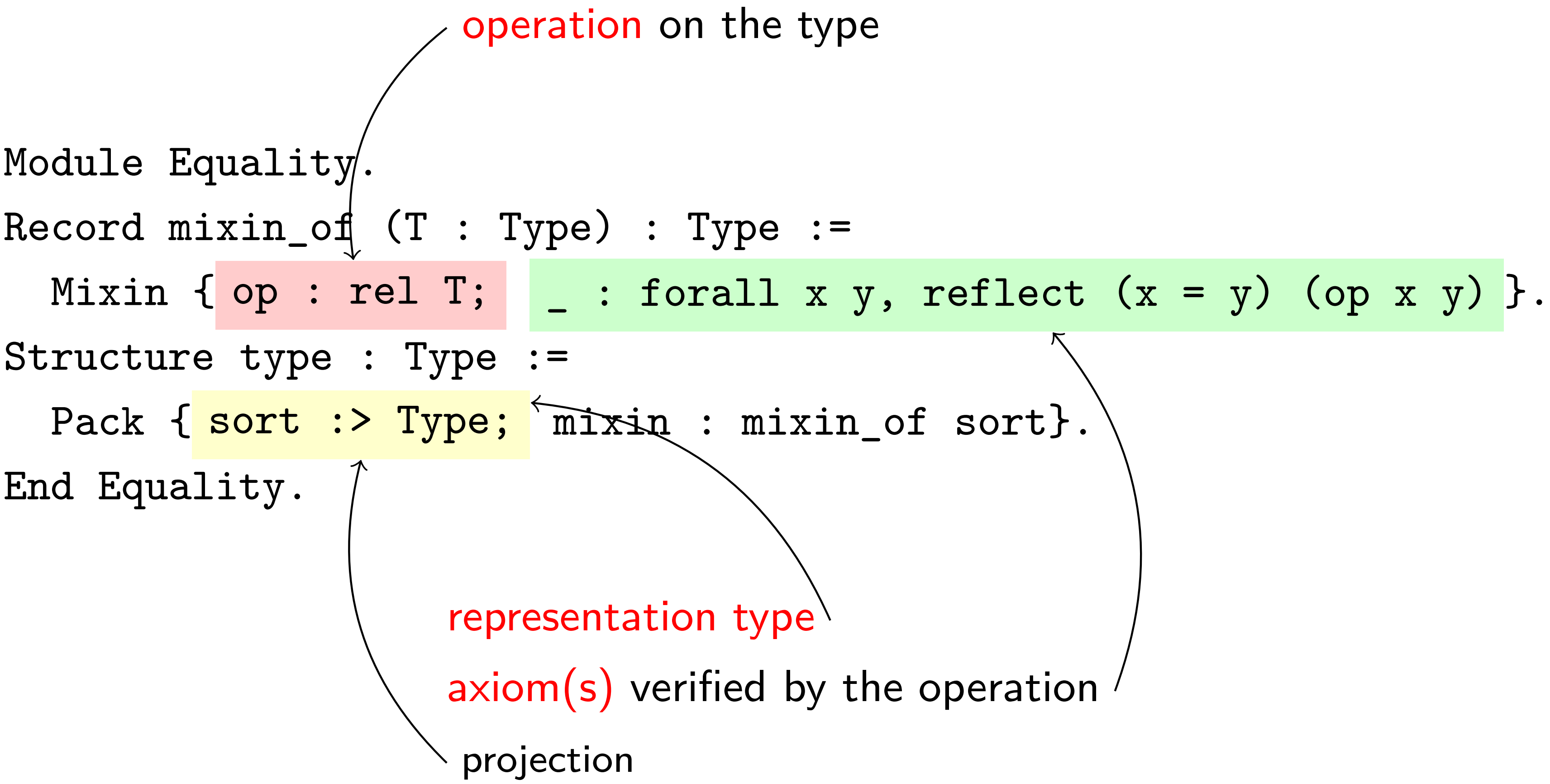
  associative add;
- . . .

C elements per structure, n nested (parametric) structures in which the parameter occurs at every element: term size in C^n

- How do we pass structures when enunciating lemmas ?
- Proofs introduce structures.

A Typical structure

We fill the blanks using `Canonical Structures`.



operation on the type

```
Module Equality.
Record mixin_of (T : Type) : Type :=
  Mixin { op : rel T;  _ : forall x y, reflect (x = y) (op x y) }.
Structure type : Type :=
  Pack { sort :> Type; mixin : mixin_of sort}.
End Equality.
```

representation type

axiom(s) verified by the operation

projection

A Typical structure

We fill the blanks using `Canonical Structures`.

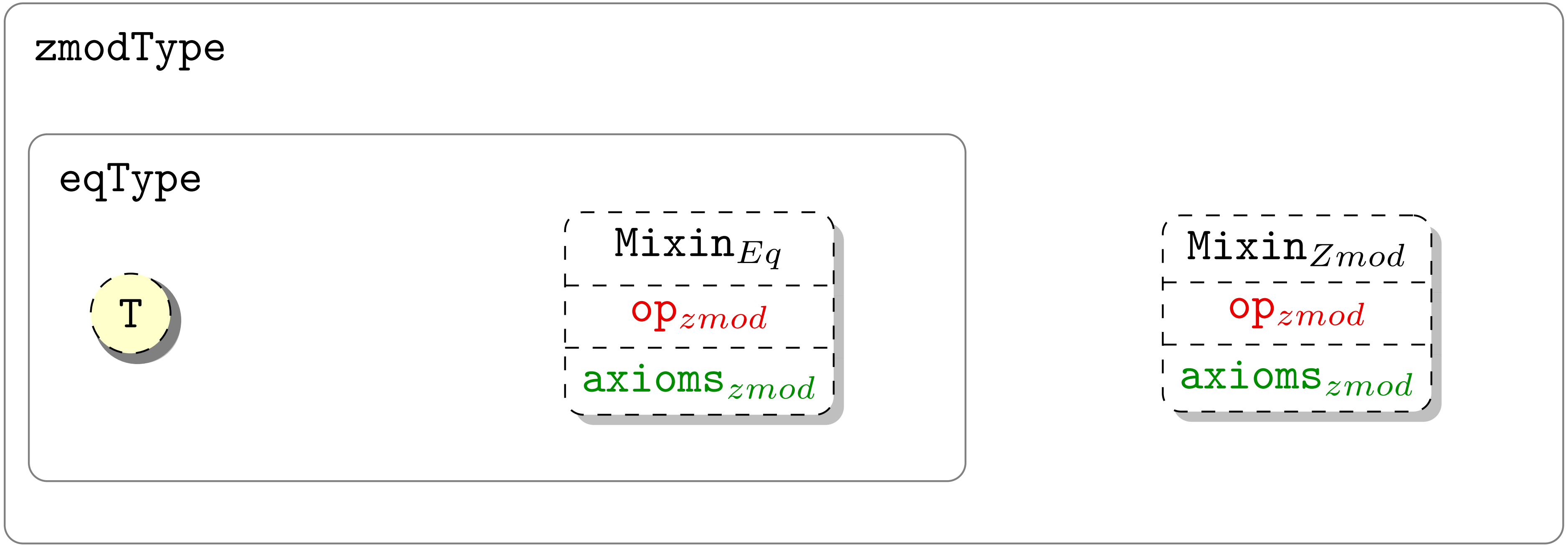
```
Module Equality.
Record mixin_of (T : Type) : Type :=
  Mixin {    : _    ; _ : _                   }.
Structure type : Type :=
  Pack {sort :> Type; mixin : mixin_of sort}.
End Equality.
```

This is generic: we use modules as namespaces only:

```
Notation eqType := Equality.type.
```

Canonical Structure Composition

- Most widely used: Telescopes



Telescopes : Canonical Structure inference

$2 + 2$

Notation

$\texttt{Zmodule.add}\ \alpha\ \texttt{2}\ \texttt{2}$

type inference

$\texttt{Equality.sort}(\texttt{Zmodule.sort}\alpha) \equiv_{\beta\iota\delta} \texttt{int}$

lookup $(\texttt{Equality.sort},\texttt{int}) \rightsquigarrow \texttt{int}_{eqType}$

$\texttt{Zmodule.sort}\alpha \equiv_{\beta\iota\delta} \texttt{int}_{eqType}$

lookup $(\texttt{Zmodule.sort},\texttt{int}_{eqType}) \rightsquigarrow \texttt{int}_{zmodType}$

$\texttt{Equality.sort (Zmodule.sort (}\texttt{int}_{zmodType}\texttt{))}$

Telescopes : Canonical Structure inference

$2 + 2$

Notation

$\texttt{Zmodule.add}\ \alpha\ 2\ 2$

type inference

$\texttt{Equality.sort}(\texttt{Zmodule.sort}\alpha) \equiv_{\beta\iota\delta} \texttt{int}$

lookup $(\texttt{Equality.sort},\texttt{int}) \leadsto \texttt{int}_{eqType}$

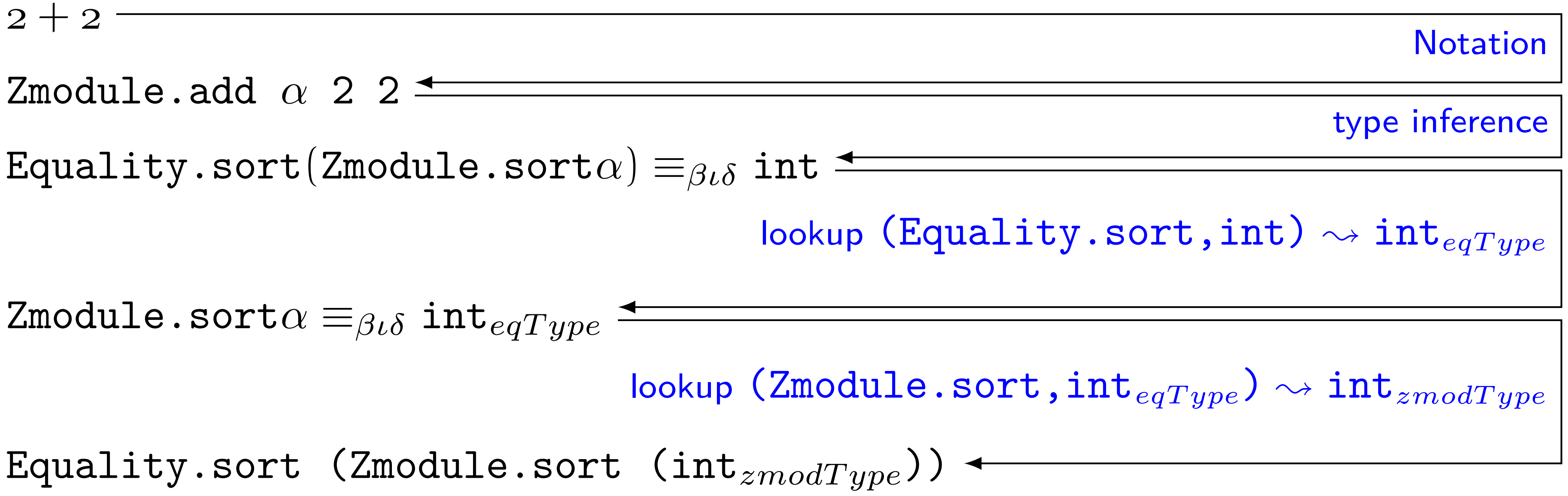
$\texttt{Zmodule.sort}\alpha \equiv_{\beta\iota\delta} \texttt{int}_{eqType}$

lookup $(\texttt{Zmodule.sort},\texttt{int}_{eqType}) \leadsto \texttt{int}_{zmodType}$

$\texttt{Equality.sort (Zmodule.sort (int}_{zmodType}\texttt{))}$

- Simple packaging, but

Telescopes : Canonical Structure inference



$2+2$

Notation

$\texttt{Zmodule.add}\ \alpha\ \texttt{2 2}$

type inference

$\texttt{Equality.sort}(\texttt{Zmodule.sort}\alpha) \equiv_{\beta\iota\delta} \texttt{int}$

lookup $(\texttt{Equality.sort},\texttt{int}) \rightsquigarrow \texttt{int}_{eqType}$

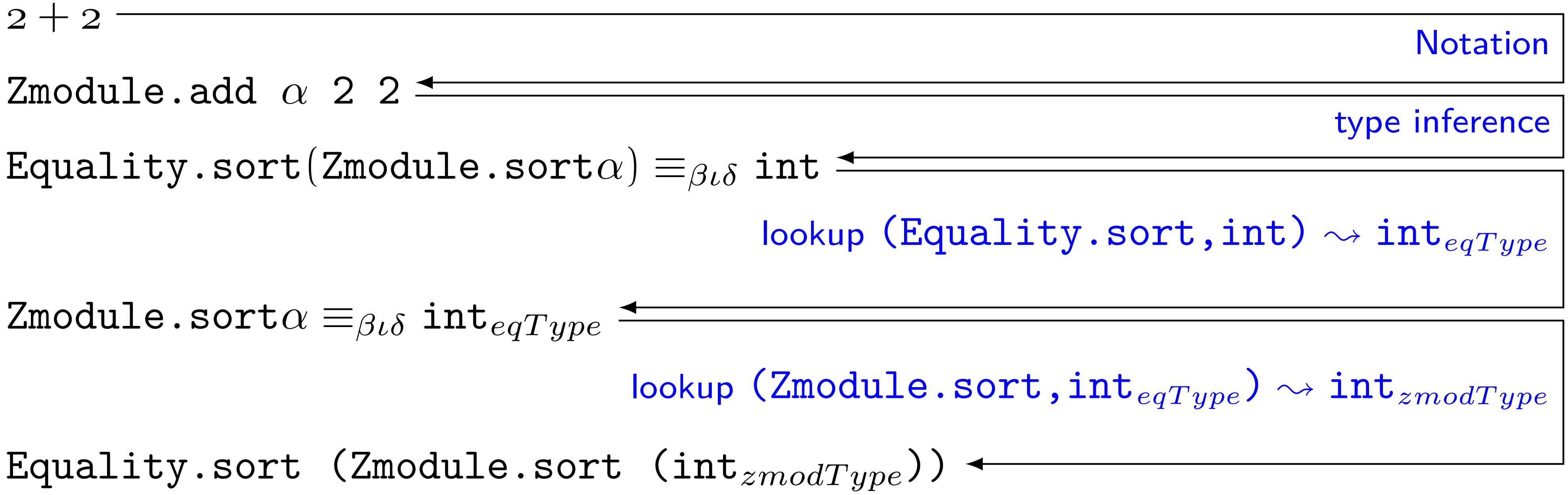
$\texttt{Zmodule.sort}\alpha \equiv_{\beta\iota\delta} \texttt{int}_{eqType}$

lookup $(\texttt{Zmodule.sort},\texttt{int}_{eqType}) \rightsquigarrow \texttt{int}_{zmodType}$

$\texttt{Equality.sort (Zmodule.sort (int}_{zmodType}\texttt{))}$

- Simple packaging, but
- head constant always the same `x:T` is interpreted as `x:Equality.sort(Zmodule.sort T)`

Telescopes : Canonical Structure inference



$2 + 2$

Notation

$\texttt{Zmodule.add}\ \alpha\ \texttt{2}\ \texttt{2}$

type inference

$\texttt{Equality.sort}(\texttt{Zmodule.sort}\alpha) \equiv_{\beta\iota\delta} \texttt{int}$

lookup $(\texttt{Equality.sort},\texttt{int}) \leadsto \texttt{int}_{eqType}$

$\texttt{Zmodule.sort}\alpha \equiv_{\beta\iota\delta} \texttt{int}_{eqType}$

lookup $(\texttt{Zmodule.sort},\texttt{int}_{eqType}) \leadsto \texttt{int}_{zmodType}$

$\texttt{Equality.sort (Zmodule.sort (}\texttt{int}_{zmodType}\texttt{))}$

- Simple packaging, but
- head constant always the same `x:T` is interpreted as `x:Equality.sort(Zmodule.sort T)`
- we're defining coercions/canonical projections on `Equality.sort` !

Telescopes : Canonical Structure inference

$2+2$

Notation

$\texttt{Zmodule.add}\ \alpha\ \texttt{2}\ \texttt{2}$

type inference

$\texttt{Equality.sort}(\texttt{Zmodule.sort}\alpha) \equiv_{\beta\iota\delta} \texttt{int}$

lookup $(\texttt{Equality.sort},\texttt{int}) \rightsquigarrow \texttt{int}_{eqType}$

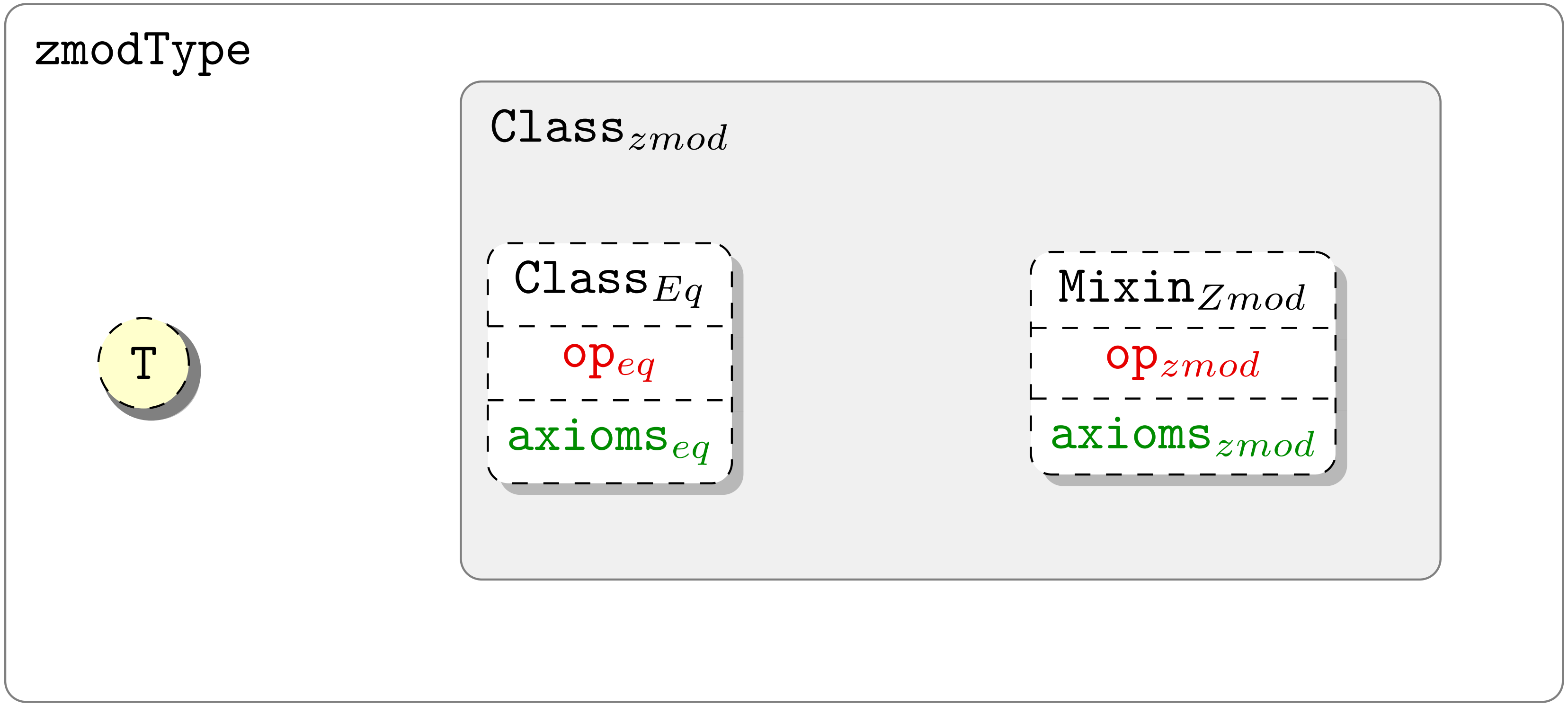
$\texttt{Zmodule.sort}\alpha \equiv_{\beta\iota\delta} \texttt{int}_{eqType}$

lookup $(\texttt{Zmodule.sort},\texttt{int}_{eqType}) \rightsquigarrow \texttt{int}_{zmodType}$

$\texttt{Equality.sort}\ (\texttt{Zmodule.sort}\ (\texttt{int}_{zmodType}))$

- Simple packaging, but
- head constant always the same `x:T` is interpreted as `x:Equality.sort(Zmodule.sort T)`
- we're defining coercions/canonical projections on `Equality.sort` !
- as is, no multiple inheritance

Packed Classes



zmodType

$\texttt{Class}_{zmod}$

T

$\texttt{Class}_{Eq}$

$\texttt{op}_{eq}$

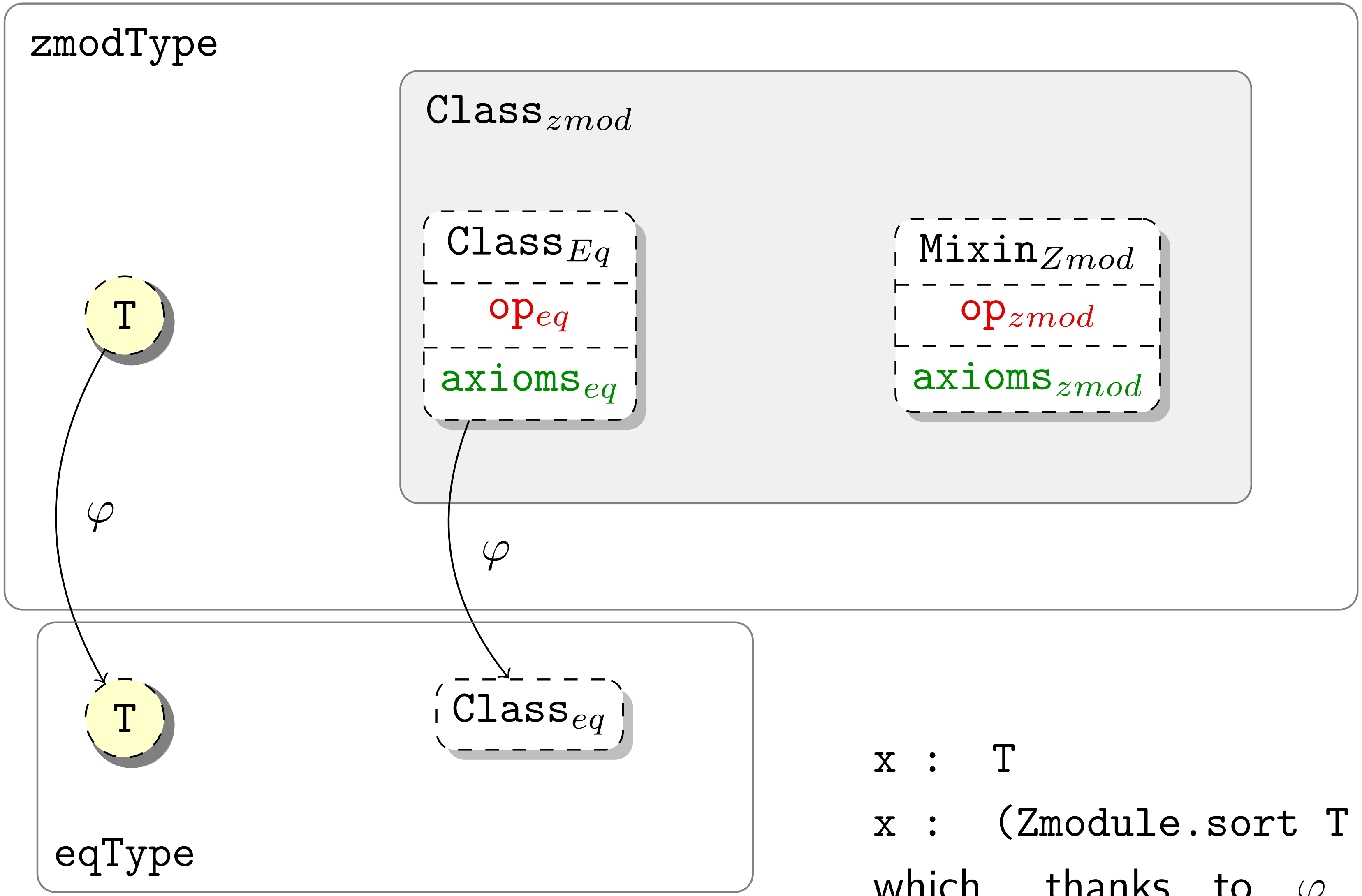
$\texttt{axioms}_{eq}$

$\texttt{Mixin}_{Zmod}$

$\texttt{op}_{zmod}$

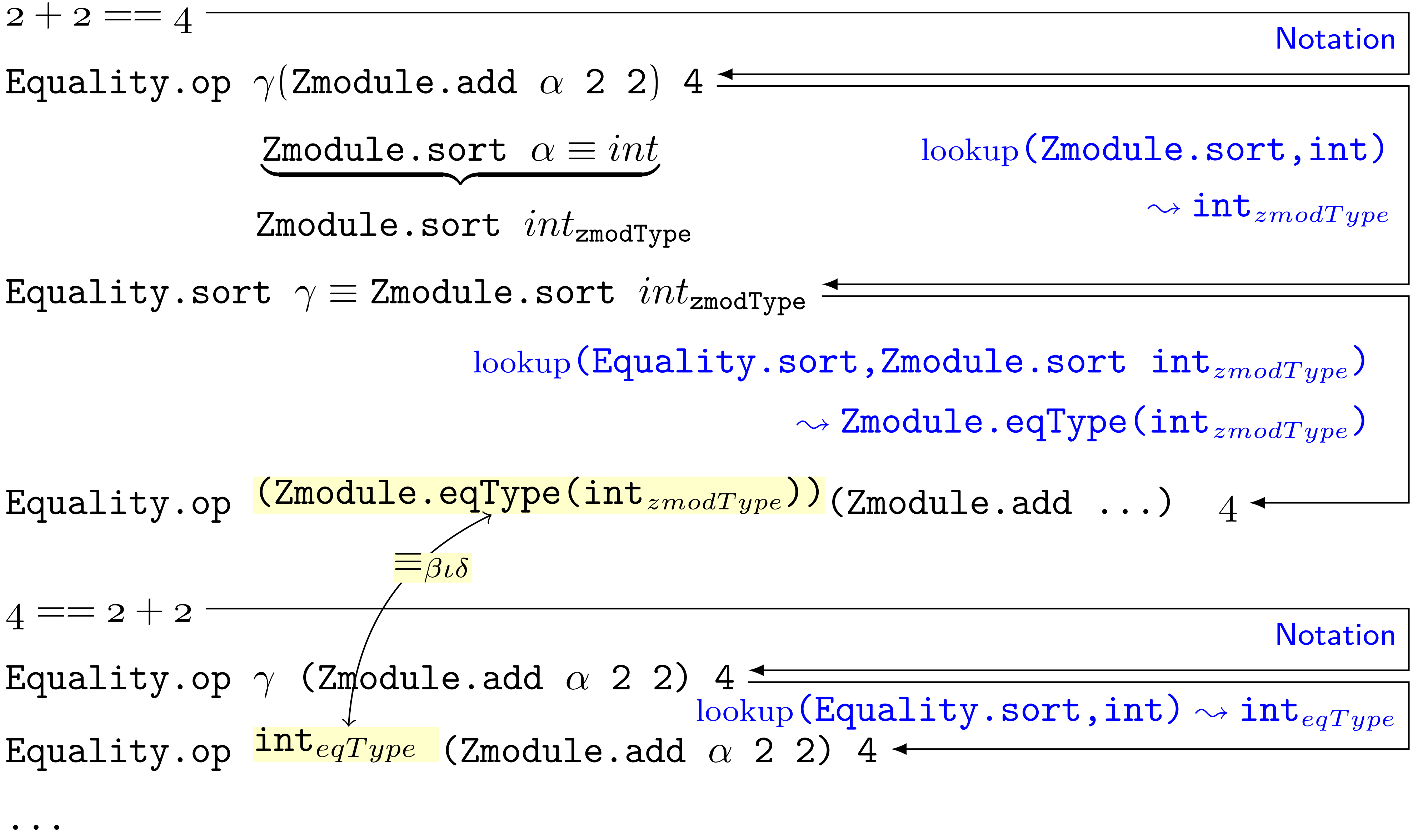
$\texttt{axioms}_{zmod}$

Packed Classes



`x : T` denotes `x : (Zmodule.sort T` α`)`, which, thanks to φ, has a canonical eqType structure.

Packed Classes



$2 + 2 == 4$

Notation

$\texttt{Equality.op}\ \gamma(\texttt{Zmodule.add}\ \alpha\ 2\ 2)\ 4$

$$\underbrace{\texttt{Zmodule.sort}\ \alpha \equiv int}_{\texttt{Zmodule.sort}\ int_{\texttt{zmodType}}}$$

lookup(Zmodule.sort,int) $\rightsquigarrow$ $\texttt{int}_{zmodType}$

$\texttt{Equality.sort}\ \gamma \equiv \texttt{Zmodule.sort}\ int_{\texttt{zmodType}}$

lookup(Equality.sort,Zmodule.sort $\texttt{int}_{zmodType}$)

$\rightsquigarrow$ Zmodule.eqType($\texttt{int}_{zmodType}$)

$\texttt{Equality.op}\ (\texttt{Zmodule.eqType}(\texttt{int}_{zmodType}))(\texttt{Zmodule.add}\ \ldots)\ 4$

$\equiv_{\beta\iota\delta}$

$4 == 2 + 2$

Notation

$\texttt{Equality.op}\ \gamma\ (\texttt{Zmodule.add}\ \alpha\ 2\ 2)\ 4$

lookup(Equality.sort,int) $\rightsquigarrow$ $\texttt{int}_{eqType}$

$\texttt{Equality.op}\ \texttt{int}_{eqType}\ (\texttt{Zmodule.add}\ \alpha\ 2\ 2)\ 4$

...

Packed Classes

- Coherent coercions,
- `x:T` interpreted as `Zmodule.sort T`.
- Multiple inheritance:

```
Module ComUnitRing.
Record class_of (R : Type) : Type := Class {
  base1 :> ComRing.class_of R;
  ext :> UnitRing.mixin_of (Ring.Pack base1 R)}.
Coercion base2 R m := UnitRing.Class (@ext R m).
...
End ComUnitRing.
```

Conclusion

- algebraic structure composition is interface programming
- representing a large hierarchy implies packaging
- know the work ahead,
- and take the time to build the tools you will need

- `A + x`

- `A + x`
- `add (matrix T) (zeromx T) (addmx T)`

- `A + x`
- `add (matrix T) (zeromx T) (addmx T)`
- `... (addmx (poly T) (zeropx T) (addpx T) )`

◆ `A + x`

◆ `add (matrix T) (zeromx T) (addmx T)`

◆ `... (addmx (poly T) (zeropx T) (addpx T) )`

◆ and `zeromx` is itself `(zeromx (poly T) (zeropx T) (addpx T) )` ...