Concept representation in a type system

Lucas E. Morales

MIT
Computational Cognitive Science

Learning as Program Induction, CogSci 2018

Concept representation in a type system

Purpose:

- Learning programs ("child coder") is **more** than writing procedural code.

Spoken: First, there is more to learning programs than writing procedural code. We will discuss a more abstract, purely-relational aspect to programming.

Concept representation in a type system

Purpose:

- Learning programs ("child coder") is **more** than writing procedural code.
- Use type systems to **express meaning**, à la conceptual role semantics.

Spoken: We will discuss *types*, which give *meaning* to procedures at a more abstract level than concrete code.

Concept representation in a type system

Purpose:

- Learning programs ("child coder") is **more** than writing procedural code.
- Use type systems to **express meaning**,
  à la conceptual role semantics.
- Type systems provide a good representation for a computational study of **concept learning**.

Spoken: We will see the ways concept learning manifests in a type system.

Concept representation in a type system

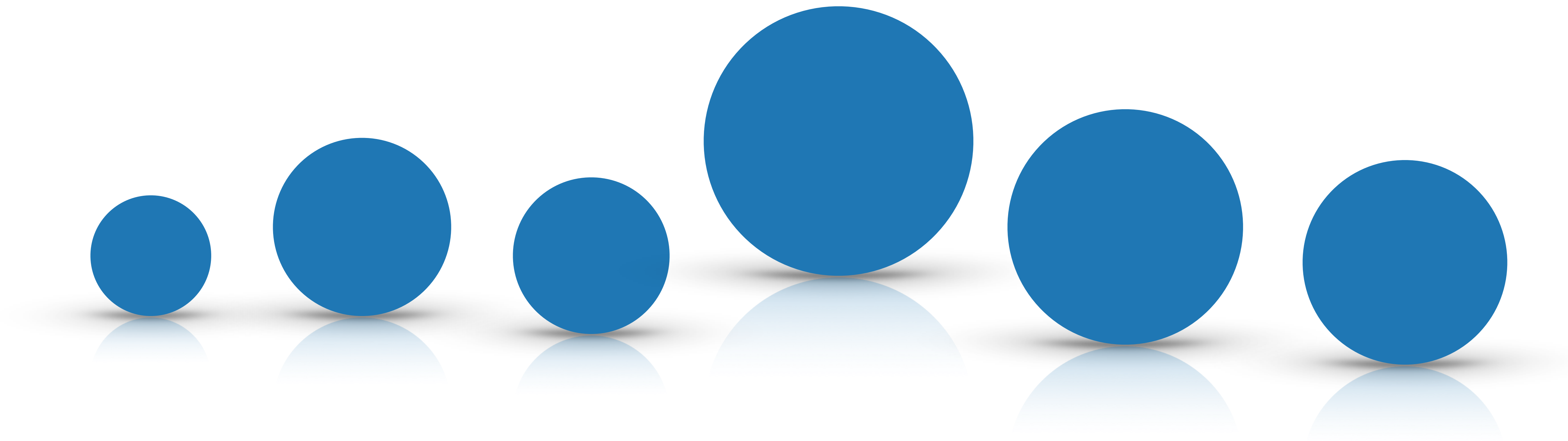
Purpose:

- Learning programs ("child coder") is **more** than writing procedural code.
- Use type systems to **express meaning**, à la conceptual role semantics.
- Type systems provide a good representation for a computational study of **concept learning**.

Note:

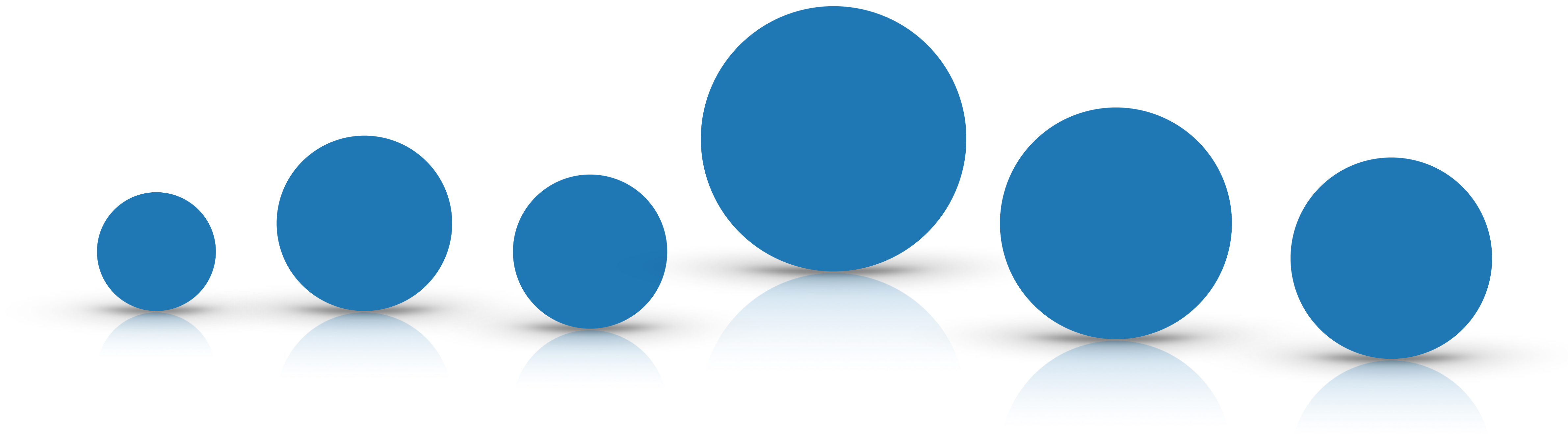
- Technical details ≪ key ideas.

Spoken: There will be technical details that should not discourage you. We are presenting a *formal framework* for concept representation, so there is mathematical content that is *not essential* for high-level understanding. We will look at code, but I will accompany code with natural description of the idea being demonstrated.



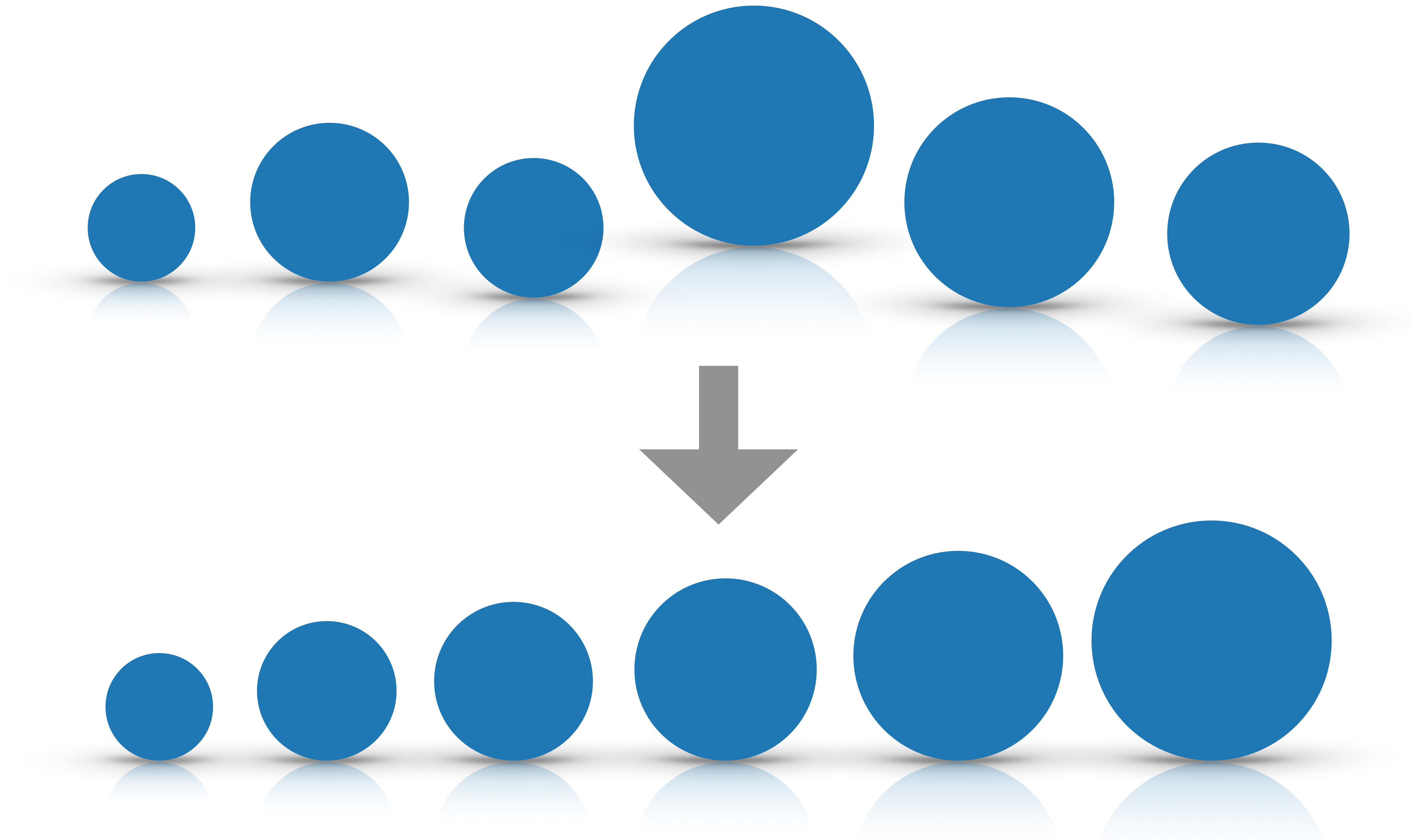
Spoken: Suppose we have balls of various sizes. You can..

Sort



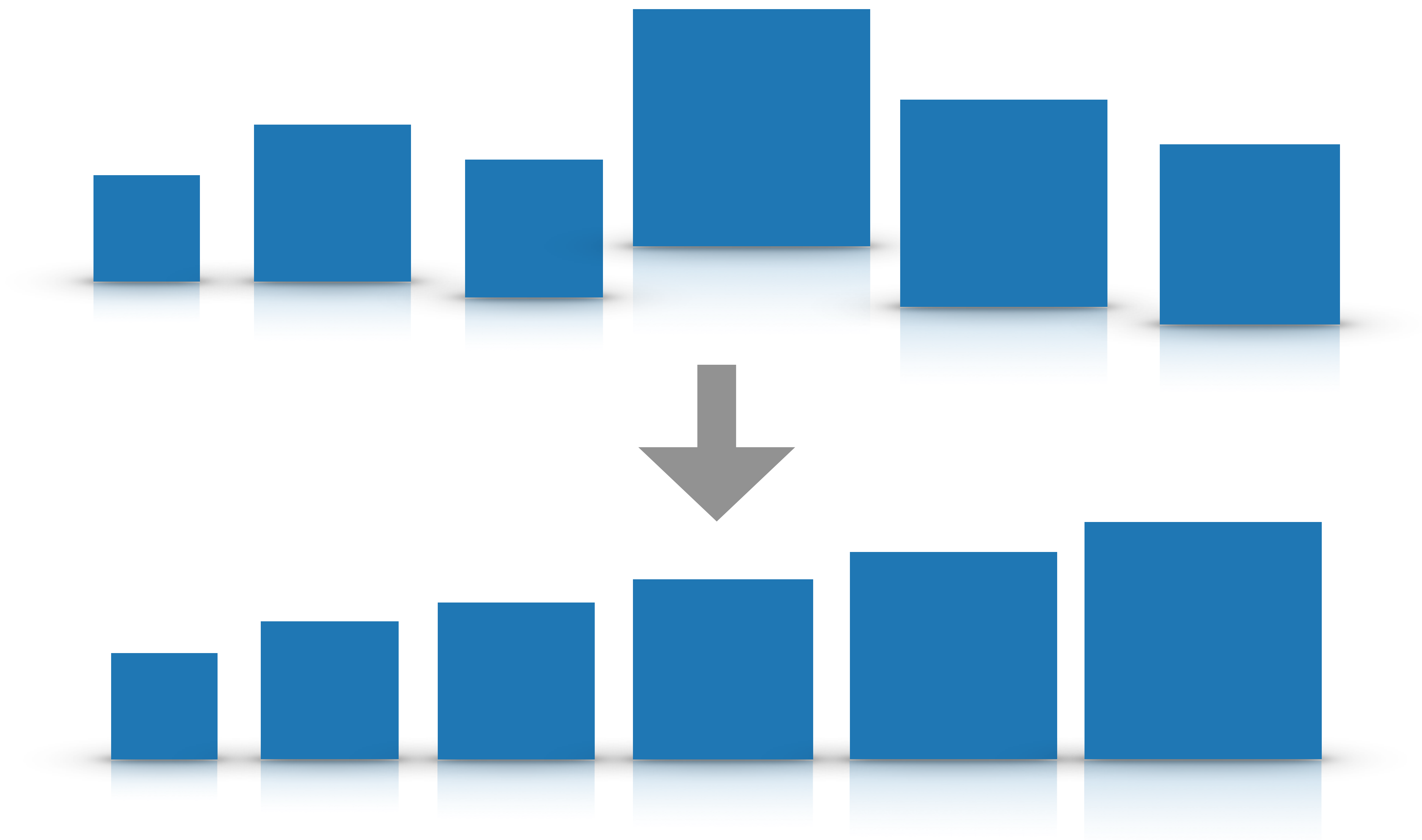
Spoken: ..sort them, like this:

Sort



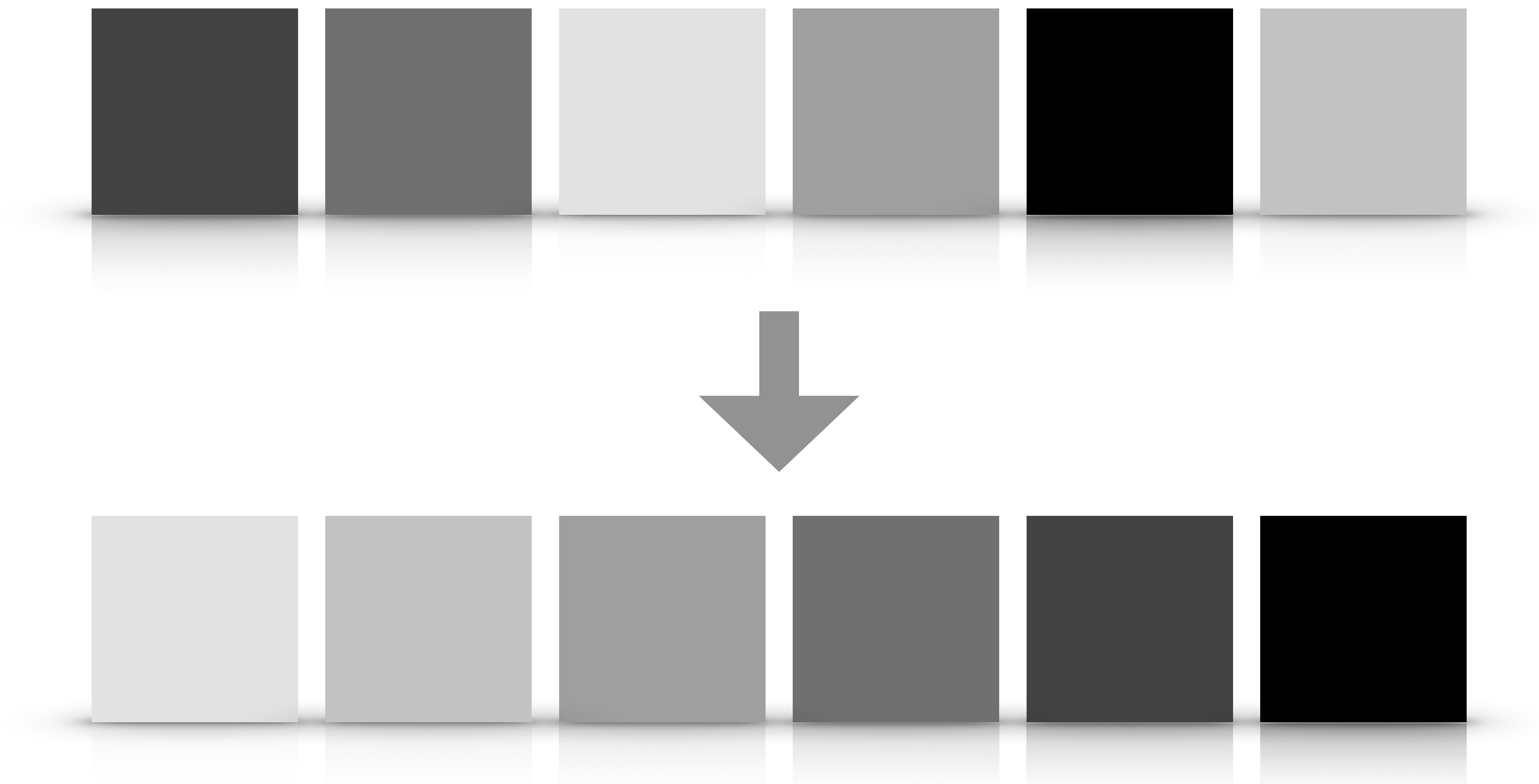
Spoken: You could also sort:

Sort



Spoken: sized boxes, or

Sort



Spoken: shaded boxes. What does it mean to learn sorting, as a program? It could be learning concrete code, *or* it could be learning the abstract definition: a program spec.

On *sort*

"It takes a set of things which are orderable,
and gives a sequence of those things in order."

Spoken: In words, we might say that it {takes a set of things} {which are orderable} and {gives a sequence of those things} {in order}. We don't communicate "sort" by giving an algorithm, but by the defining the *type* of procedure.

On *sort*

"It takes a set of things which are orderable, and gives a sequence of those things in order."

$$\textbf{sort} :: \underbrace{\textbf{Ord}\ T \Rightarrow}_{\text{type class constraint}} \overbrace{(i : [T])}^{\text{input type}} \rightarrow \underbrace{\big(v : [T] \mid \textbf{elems}(i) = \textbf{elems}(v) \wedge \textbf{nondecreasing}(v)\big)}_{\text{output type}}$$

Sort as a type

Spoken: This can be expressed by type declaration, which we'll try to make more sense of it later. But for now believe me that:

On *sort*

"It takes a set of things which are orderable, and gives a sequence of those things in order."

Sort as a type

$$\textbf{sort} :: \underbrace{\textbf{Ord}\ T}_{\text{type class constraint}} \Rightarrow \overbrace{(i : [T])}^{\text{input type}} \to \underbrace{\big(v : [T] \mid \textbf{elems}(i) = \textbf{elems}(v) \wedge \textbf{nondecreasing}(v)\big)}_{\text{output type}}$$

- This completely defines sorting.

On *sort*

"It takes a set of things which are orderable, and gives a sequence of those things in order."

Sort as a type

$$\textbf{sort} :: \underbrace{\textbf{Ord}\ T}_{\text{type class constraint}} \Rightarrow \overbrace{(i : [T])}^{\text{input type}} \rightarrow \underbrace{\big(v : [T] \mid \textbf{elems}(i) = \textbf{elems}(v) \wedge \textbf{nondecreasing}(v)\big)}_{\text{output type}}$$

- This completely defines sorting.
- It does not matter what we are sorting, as long as the items have an ordering.

On *sort*

"It takes a set of things which are orderable, and gives a sequence of those things in order."

Sort as a type

$$\mathbf{sort} :: \underbrace{\mathbf{Ord}\ T}_{\text{type class constraint}} \Rightarrow \overbrace{(i : [T])}^{\text{input type}} \to \underbrace{\big(v : [T] \mid \mathbf{elems}(i) = \mathbf{elems}(v) \wedge \mathbf{nondecreasing}(v)\big)}_{\text{output type}}$$

- This completely defines sorting.
- It does not matter what we are sorting, as long as the items have an ordering.
- Concrete implementation is irrelevant.

Spoken: This declarative style of *definition in a type system* puts us in the realm of *conceptual role.*

On *sort*

"It takes a set of things which are orderable,
and gives a sequence of those things in order."

$$\textbf{sort} :: \underbrace{\textbf{Ord}\ T \Rightarrow}_{\text{type class constraint}} \overbrace{(i : [T])}^{\text{input type}} \rightarrow \underbrace{\big(v : [T] \mid \textbf{elems}(i) = \textbf{elems}(v) \land \textbf{nondecreasing}(v)\big)}_{\text{output type}}$$

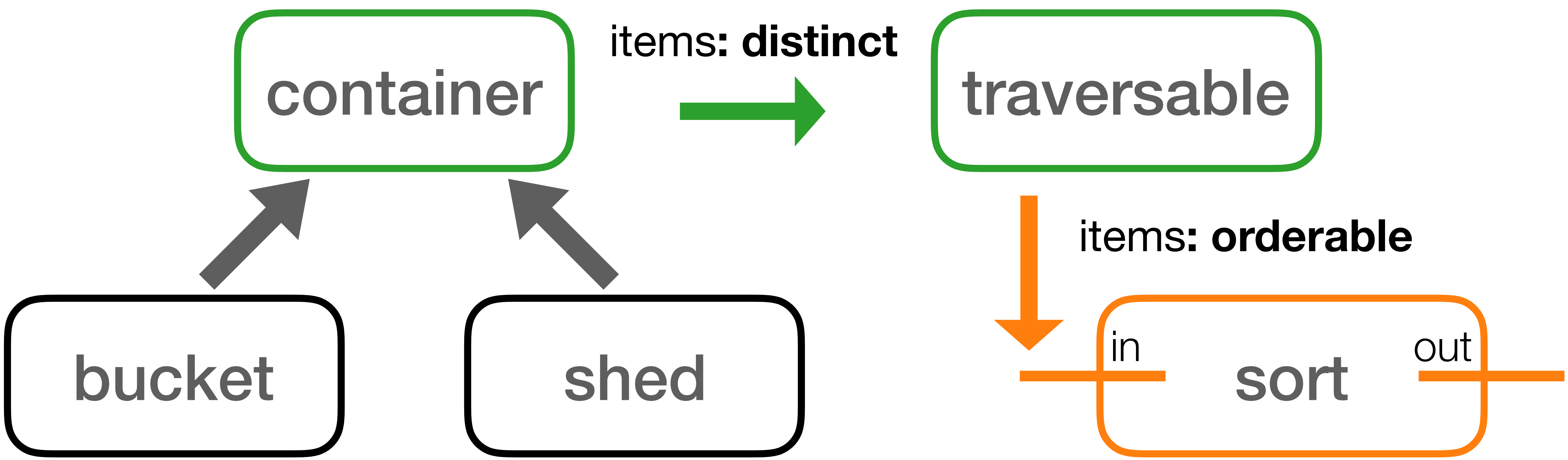
Sort as a type

- This completely defines sorting.
- It does not matter what we are sorting, as long as the items have an ordering.
- Concrete implementation is irrelevant.

Realm: conceptual role

Spoken: Keep that last point in mind: *concrete implementation is irrelevant*. Also, types are useful for more than just procedures — computer scientists developed ways of representing many kinds of relations in a type system:

Some representable concepts

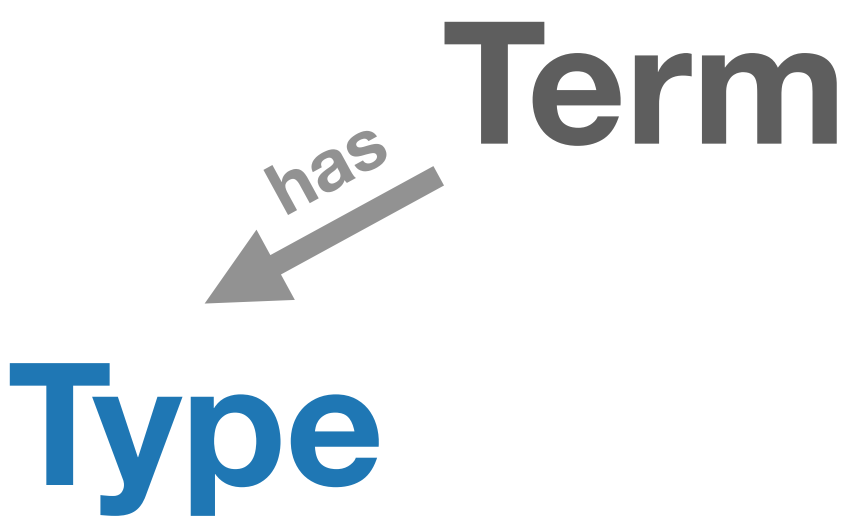


Spoken: For example, we can model classes of object like *container*, with instances like *box* or *shed*. If a container has distinct object and not something liquid, then the container is *traversable*. If something is traversable and the items are orderable, then we can sort those items.

Concept representation in a type system

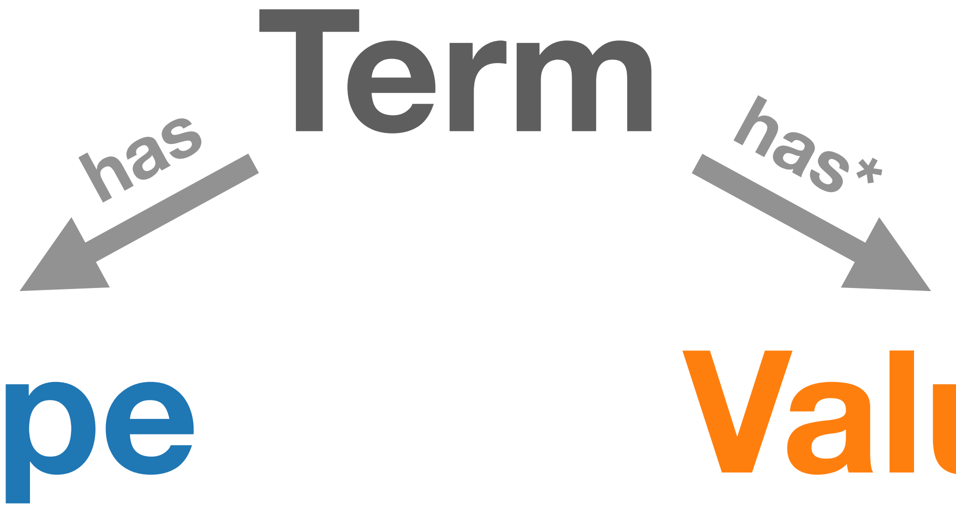
- **What is a type system?**
- Why should cognitive scientists care about types?
- What constitutes the effects of learning?
- What does this model lack?

What is a type system?

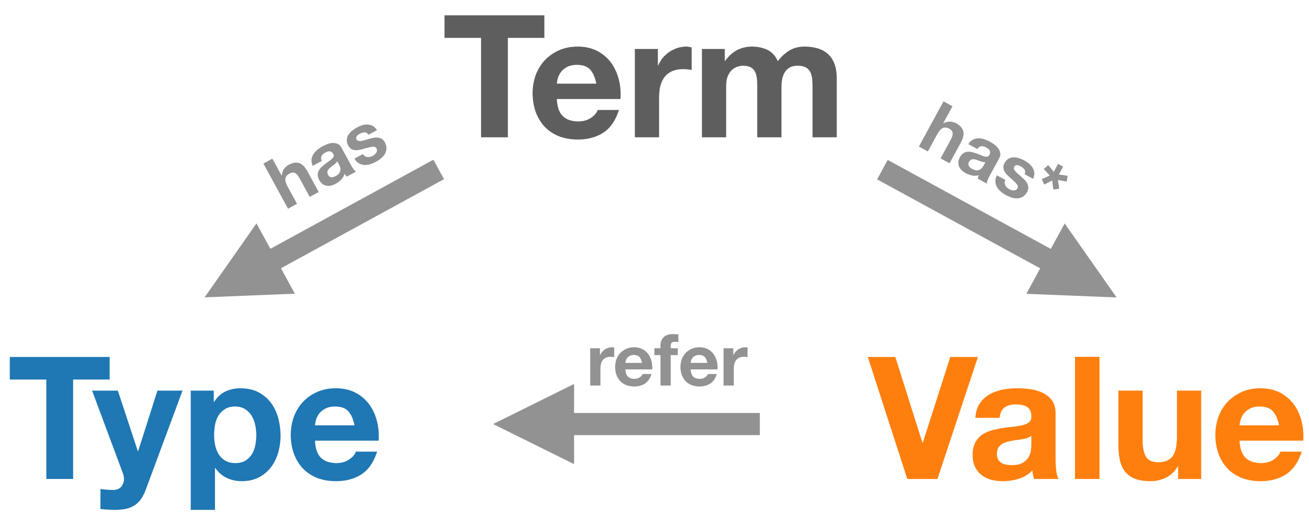


Spoken: In a type system, *types* are the set of values that can inhabit a *term*, where a term is a syntactic construct that — at any point during its existence at runtime — possesses exactly one *value* with its ascribed *type*.

What is a type system?



What is a type system?



Spoken: We call values of a type *inhabitants*.

What is a type system?

Type ← refer — **Value**

Spoken: For example:

What is a type system?

Type ← refer — **Value**

Type	Value
`String`	`"hello world"`
`Int`	`12`

Spoken: The phrase "hello world" inhabits the type "String", and the number 12 inhabits the type "Int".

What is a type system?

Type ← refer — **Value**

primitive*	`String`	`"hello world"`
	`Int`	`12`

* `String` is often non-primitive, an alias for a list of characters.

Spoken: Types like these are called *primitive data types.*

What is a type system?

Type ← refer — **Value**

	Type	Value
primitive*	`String`	`"hello world"`
	`Int`	`12`
algebraic	`Color`	**`Red`** **`Blue`** **`Green`**
	`Car`	**`Car{`**`model="prius", year=2013`**`}`**

Spoken: *Algebraic data types* allow us to express variants, such as "red" being a "color", or {alternatively} "blue", "green", and other colors. We can also express structures of typed data, such as "car" consisting of relevant typed details.

What is a type system?

Type ← refer — **Value**

	Type	Value
primitive*	`String`	`"hello world"`
	`Int`	`12`
algebraic	`Color`	**`Red`** **`Blue`** **`Green`**
	`Car`	**`Car{`**`model="prius", year=2013`**`}`**
composite	`(String, Int)`	**`(`**`"CogSci", 2018`**`)`**
	`List Int`	**`[`**`0, 1, 3, 6`**`]`**
	`Maybe String`	**`Some(`**`"LaPI"`**`)`**

Spoken: *Composite types* are defined in terms of other types. The "Maybe" type at the bottom, sometimes called "optional", is either {an empty value} or {some value of a particular type}. These composite types are also algebraic.

What is a type system?

Type ← refer — **Value**

	Type	Value
primitive*	`String`	`"hello world"`
	`Int`	`12`
algebraic	`Color`	**`Red`** **`Blue`** **`Green`**
	`Car`	**`Car{`**`model="prius", year=2013`**`}`**
composite	`(String, Int)`	**`(`**`"CogSci", 2018`**`)`**
	`List Int`	**`[`**`0, 1, 3, 6`**`]`**
	`Maybe String`	**`Some(`**`"LaPI"`**`)`**

Spoken: The value-type relation is like that of an {next slide} *entity*…

What is a type system?

Type ← refer **Value**

entity

Spoken: ...*entity* with its referent {next slide} *symbol.*

What is a type system?

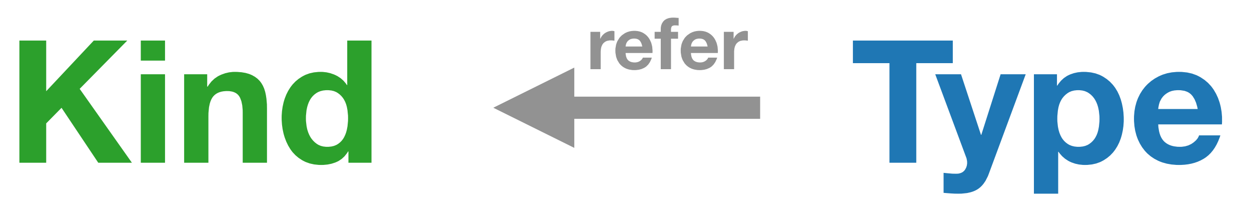
Type ← refer — **Value**

symbol | **entity**

`Dog`

Spoken: Here an aptly-named creature "Inu" is a real-world entity bearing the abstract concept of "dog".

What is a type system?



Spoken: There is also the *kind* system — the "type system for types".

What is a type system?

Kind	← refer —	Type
TYPE		`Int`
TYPE		`List Int`
TYPE		`Maybe String`

Spoken: The "TYPE" kind is for types whose values exist at runtime. Historically, this kind is written as a star.

What is a type system?

Kind	← refer	Type+
Type		`Int`
Type		`List Int`
Type		`Maybe String`
Type → Type		`List`
Type → Type		`Maybe`

Spoken: With kinds, we can reason about what are called *higher-kinded types*. "Maybe" is a type operator that, when given a type like "String", yields a type for optional strings.

What is a type system?

Kind	← refer	Type+
Type		`Int`
Type		`List Int`
Type		`Maybe String`
Type → Type		`List`
Type → Type		`Maybe`
ERROR!		`List Maybe`

Spoken: The kind system prevents nonsense at the type-level. What does a list of "Maybe"s mean? Perhaps the programmer meant something like this:

What is a type system?

Kind	← refer	Type+
TYPE		`Int`
TYPE		`List Int`
TYPE		`Maybe String`
TYPE → TYPE		`List`
TYPE → TYPE		`Maybe`
TYPE		`List (Maybe Int)`

What is a type system?

Kind ← refer	**Type+**
TYPE	`Int`
TYPE	`List Int`
TYPE	`Maybe String`
TYPE → TYPE	`List`
TYPE → TYPE	`Maybe`
TYPE	`List (Maybe Int)`
TYPE → CONSTRAINT	`Ord`

Spoken: Kinds help us express constraints. For example:

What is a type system?

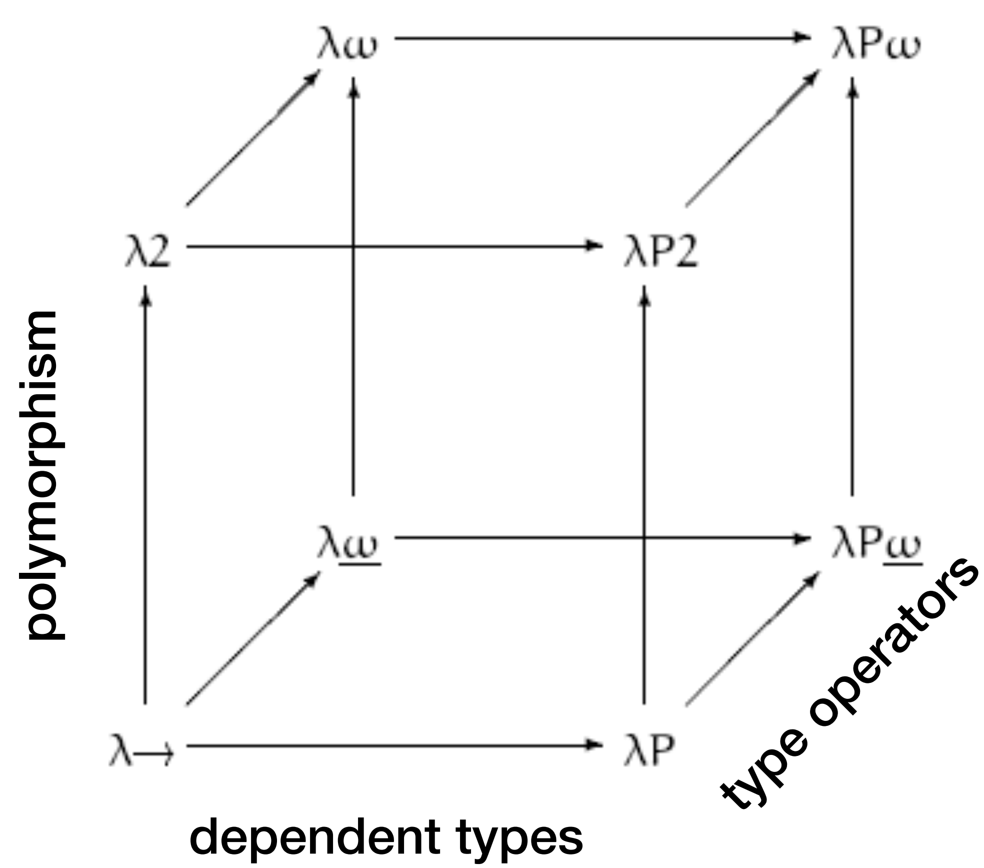
Kind	← refer	Type+
Type		`Int`
Type		`List Int`
Type		`Maybe String`
Type → Type		`List`
Type → Type		`Maybe`
Type		`List (Maybe Int)`
Type → Constraint		`Ord`

sort :: **Ord** $T \Rightarrow \cdots$

(**Ord** T: type class constraint)

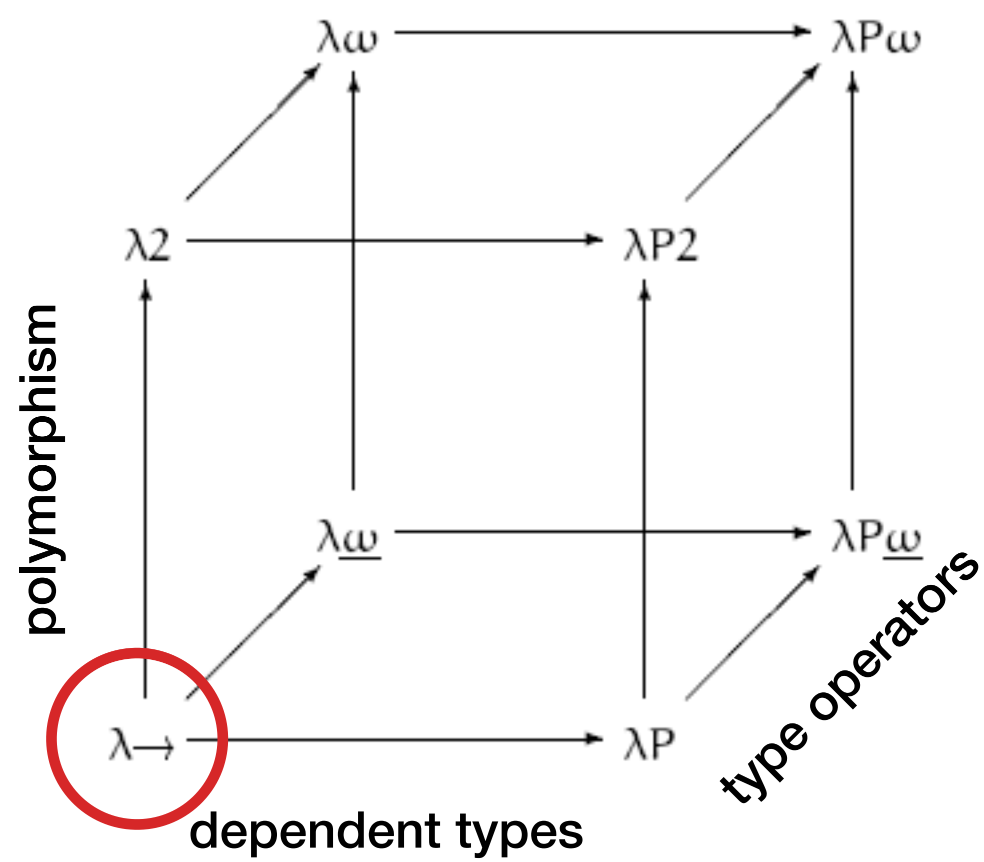
Spoken: We saw in the "sort" type earlier a *type class constraint*. It says "Ord t", making "sort"only valid when "t" satisfies whatever "Ord" requires of it. Ord requires a "compare" function which takes any two values of type "t" and returns one of the variants {less than}, {equal to}, or {greater than}.

What is a type system?



Spoken: The λ-cube here describes the type theory involved, along three orthogonal axes.

What is a type system?

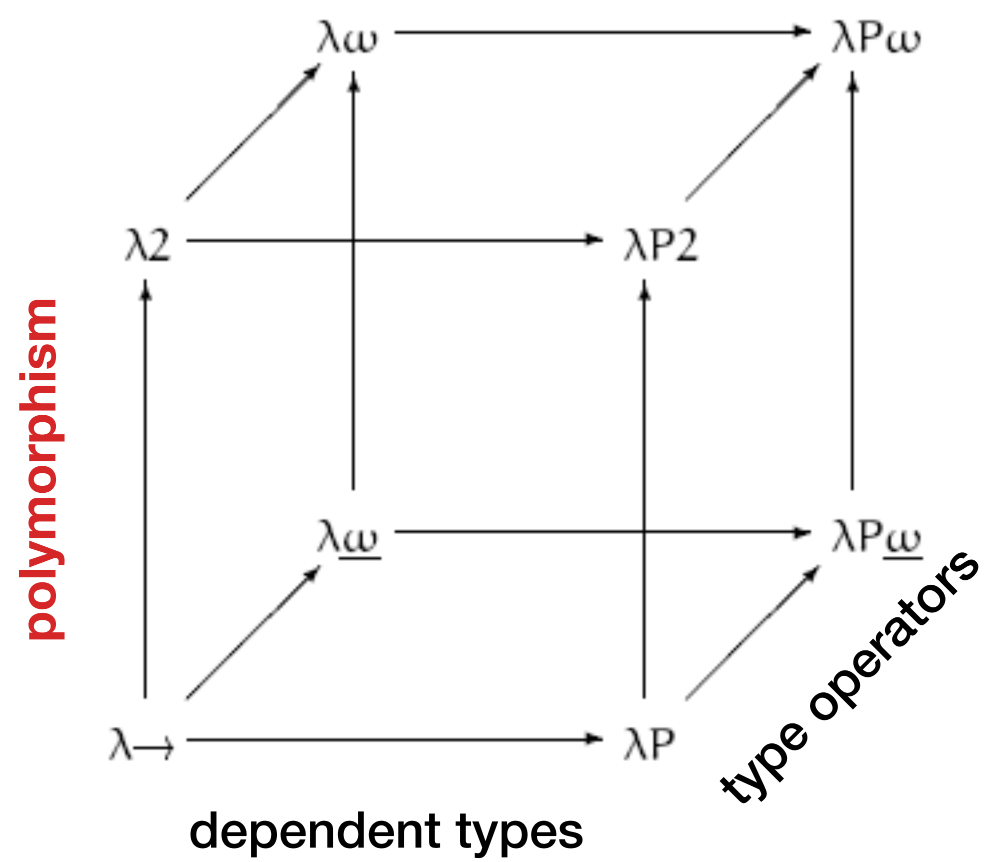


abstraction:

value → value

Spoken: All corners arise from the bottom-left, *simply-typed λ-calculus*. This gives us a starting point of abstractions — functions that take values and return values.

What is a type system?

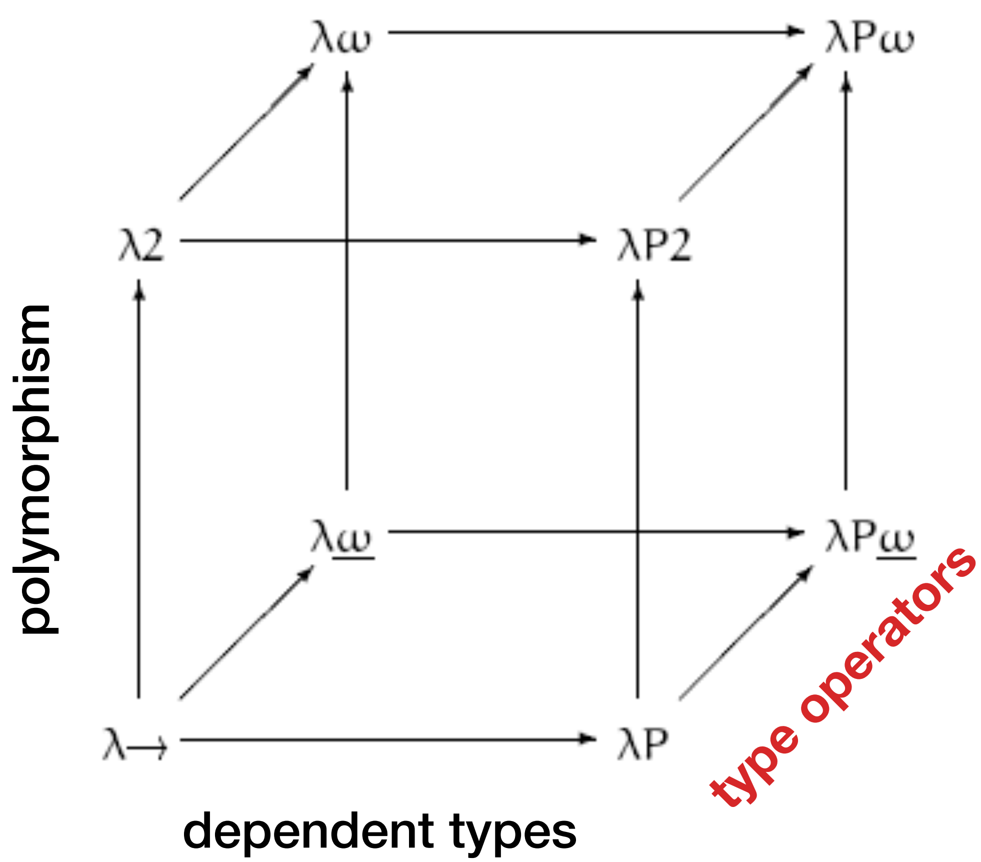


abstraction:
value → value

polymorphism:
type → value

Spoken: One axis is polymorphism, which lets us construct values according to any given type.

What is a type system?



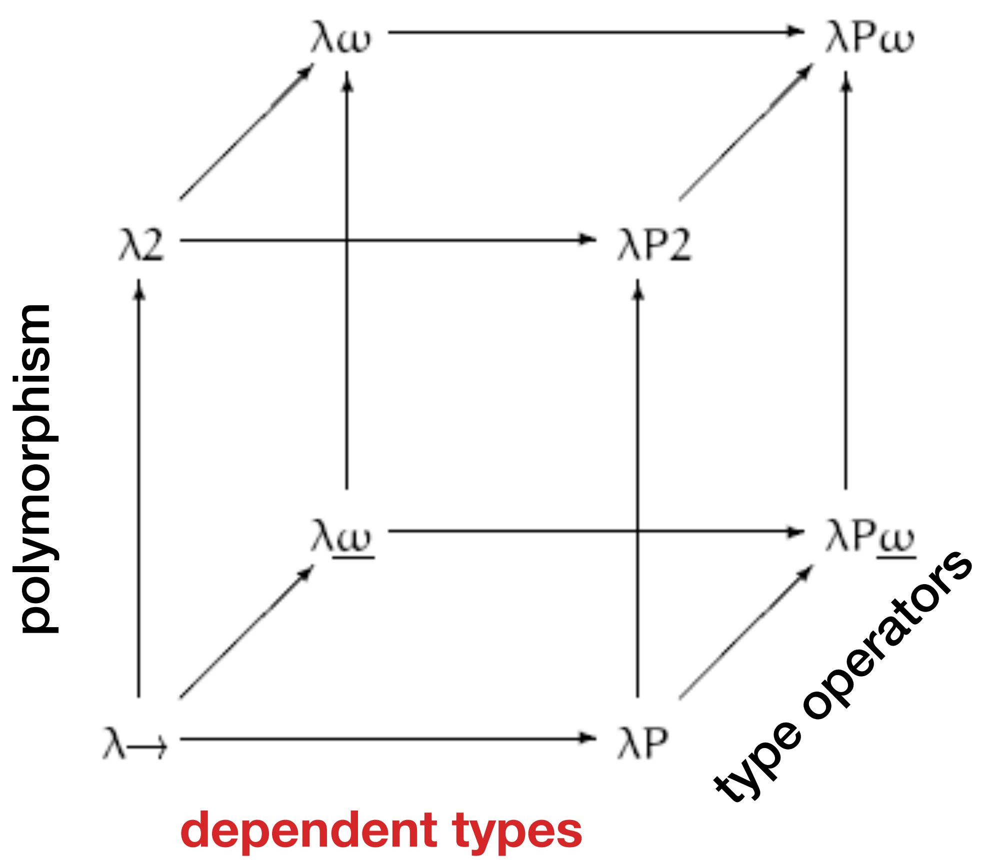
abstraction:
value → value

polymorphism:
type → value

type operators:
type → type

Spoken: Type operators give us the *composite types* we saw earlier, like "list" and "maybe". They take types as arguments and return another type. With type operators comes {the kind system}.

What is a type system?



abstraction:
value → value

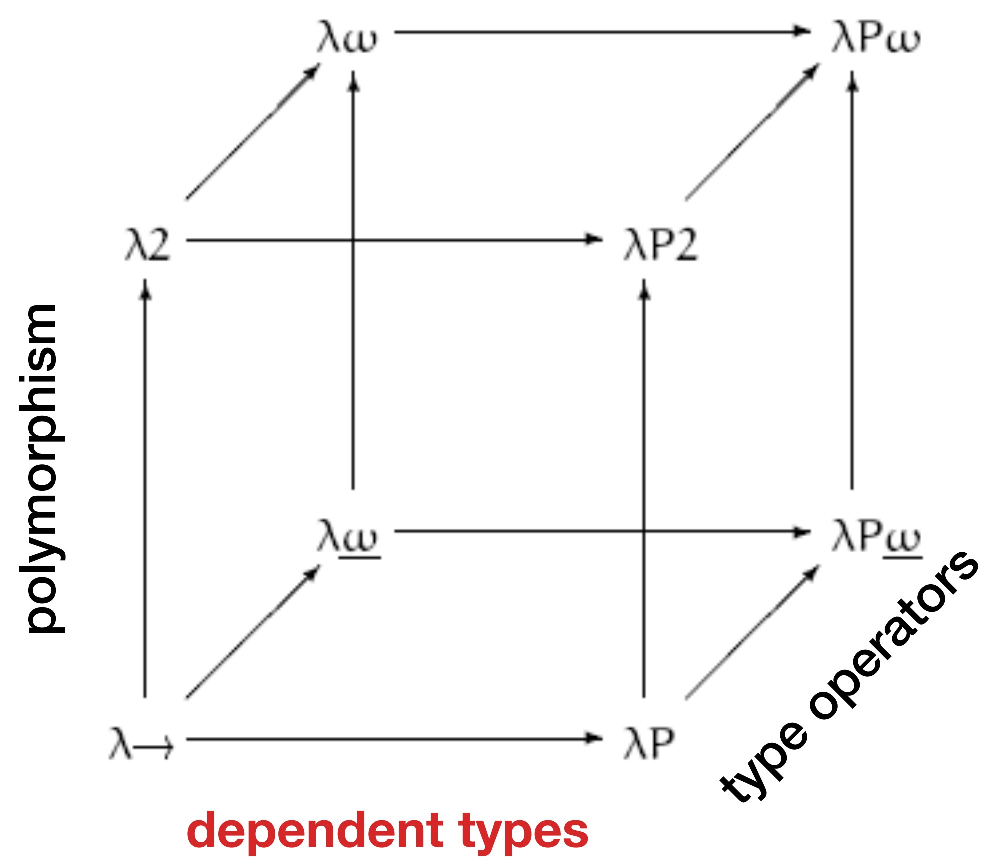
polymorphism:
type → value

type operators:
type → type

dependent types:
value → type

Spoken: Dependent types allow for first-order logic at the type-level that depend on values that may exist at runtime. For example:

What is a type system?



abstraction:
value → value

polymorphism:
type → value

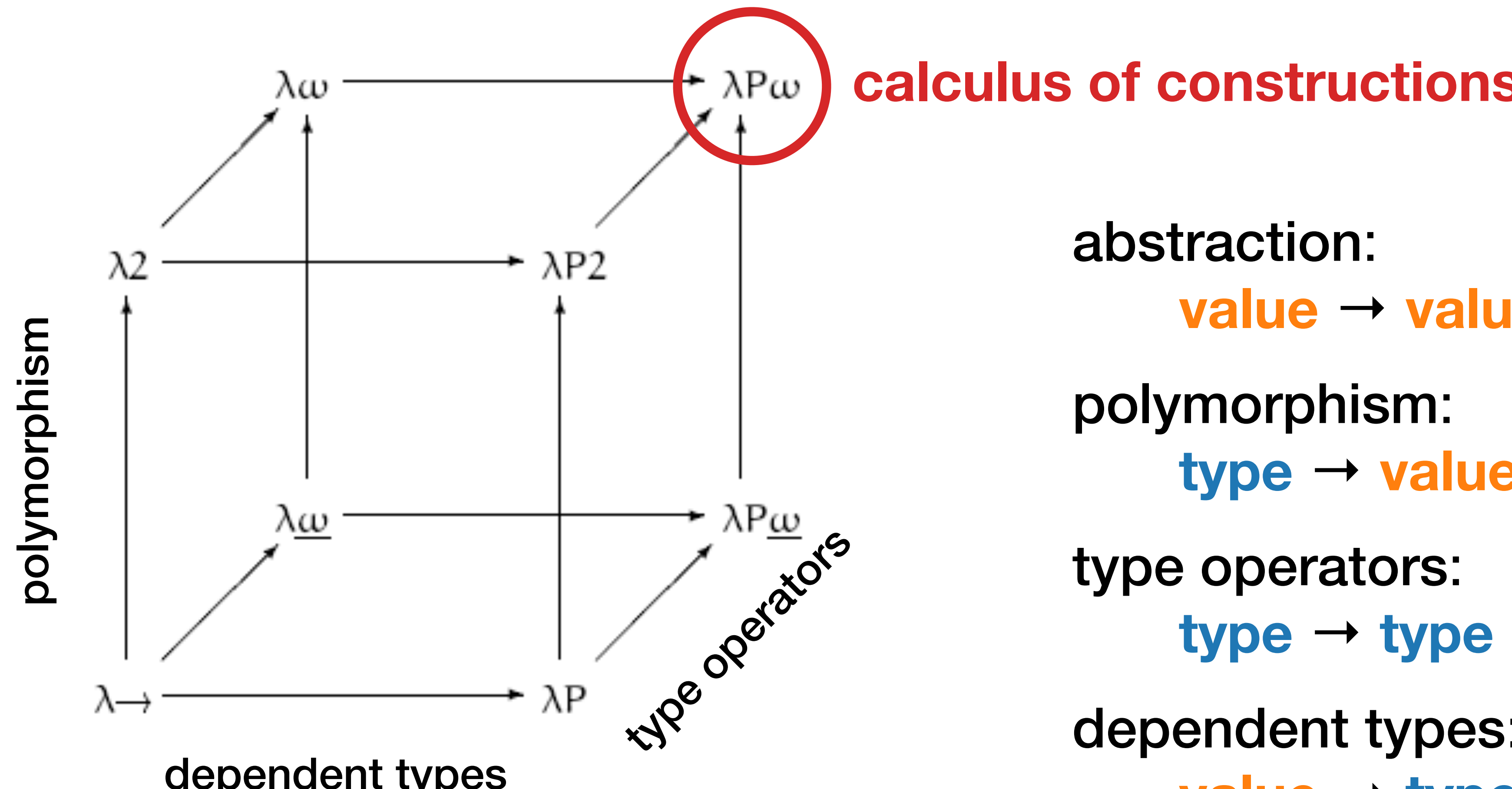
type operators:
type → type

dependent types:
value → type

$$\textbf{sort} :: \textbf{Ord}\ T \Rightarrow \underbrace{(i : [T])}_{\text{input type}} \rightarrow \underbrace{\left(v : [T] \mid \textbf{elems}(i) = \textbf{elems}(v)\ \wedge\ \textbf{nondecreasing}(v)\right)}_{\text{output type}}$$

Spoken: The return type of "sort" is every list of type "t" that shares exactly all elements of the input and is also non-decreasing. This is called a "dependent function": it universally quantifies over the input and mandates that the output type is satisfied.

What is a type system?



abstraction:
value → value

polymorphism:
type → value

type operators:
type → type

dependent types:
value → type

$$\textbf{sort} :: \textbf{Ord}\ T \Rightarrow \underbrace{(i : [T])}_{\text{input type}} \rightarrow \underbrace{\left(v : [T] \mid \textbf{elems}(i) = \textbf{elems}(v)\ \wedge\ \textbf{nondecreasing}(v)\right)}_{\text{output type}}$$

Spoken: The calculus of constructions, where all of these features are present, is the basis of many theorem provers and some programming languages — including Agda, Coq, and Idris.

Concept representation in a type system

- What is a type system?
- **Why should cognitive scientists care about types?**
- What constitutes the effects of learning?
- What does this model lack?

Why should cognitive scientists care about types?

Why should cognitive scientists care about types?

- conceptual role (expressivity)

Spoken: Types tell the story, so the *naming* of {variables and procedures} becomes less important. A symbol carries no meaning without its relationships. In parentheses I've denoted the relevant programming lingo.

Why should cognitive scientists care about types?

- conceptual role (expressivity)
- no nonsense values (make illegal states unrepresentable)

Spoken: Types make illegal states unrepresentable — e.g. if I enforce "attendance" as a natural number and not an integer, I cannot assign an invalid negative number.

Why should cognitive scientists care about types?

- conceptual role (expressivity)
- no nonsense values (make illegal states unrepresentable)
- implementation is irrelevant (illegal behavior cannot compile)

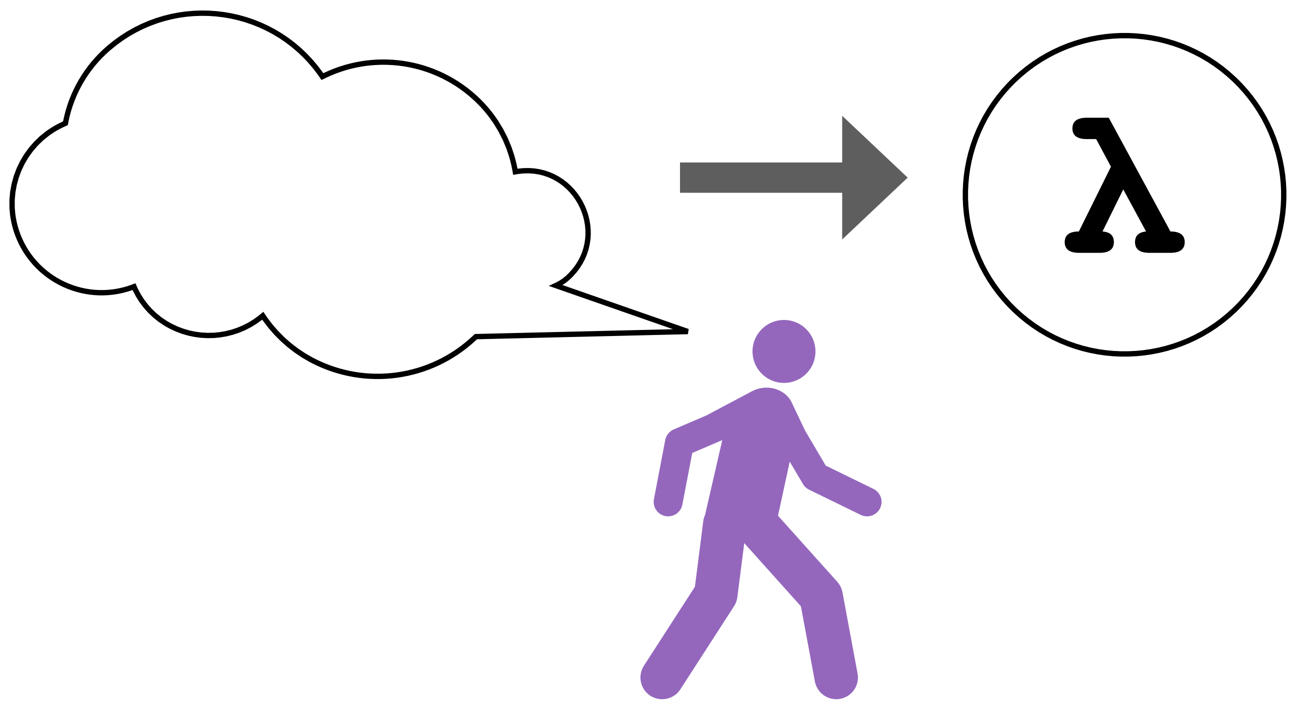
Spoken: Illegal behavior cannot compile — e.g. sort must return a list that is non-decreasing. An implementation of sort that is broken cannot exist in this framing.

Key Idea 1

Programming languages give **more than composition**: they enable complex **declarations of relation** between computational artifacts.

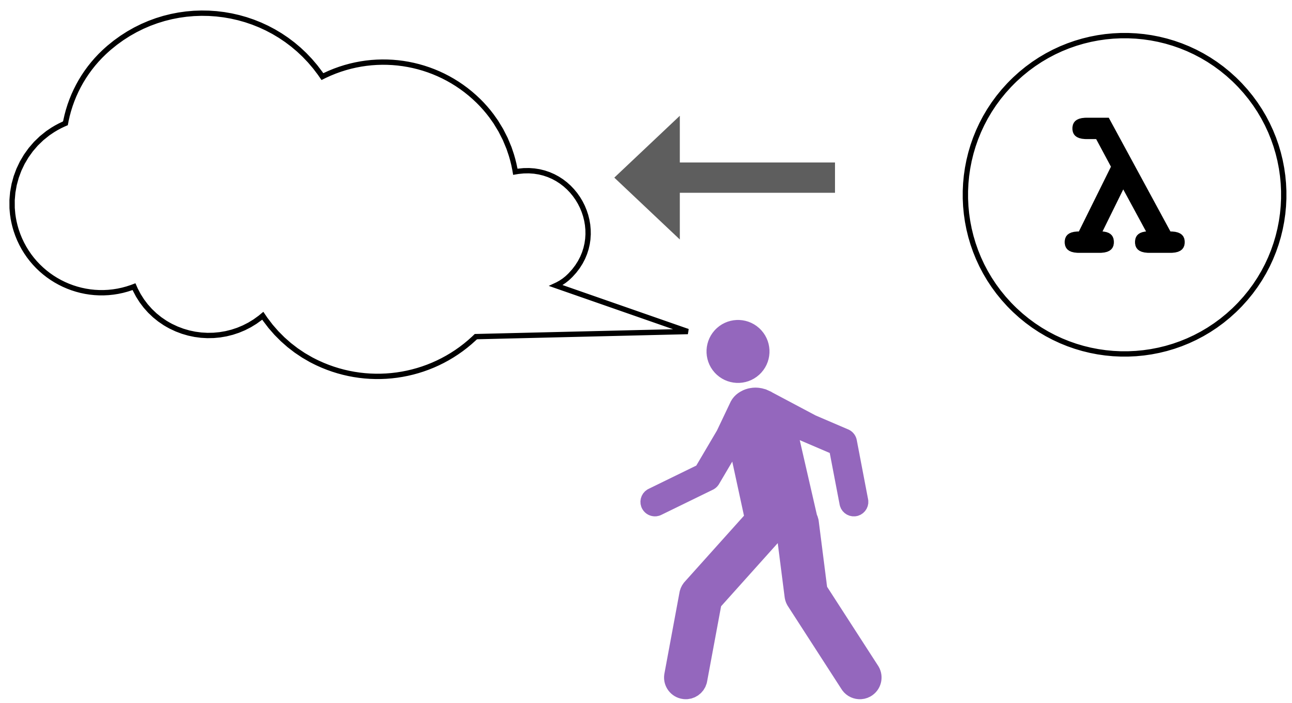
Spoken: There's a key idea here. (read.) This is perhaps best illustrated by thinking about the role of the programmer:

Programmers are translators

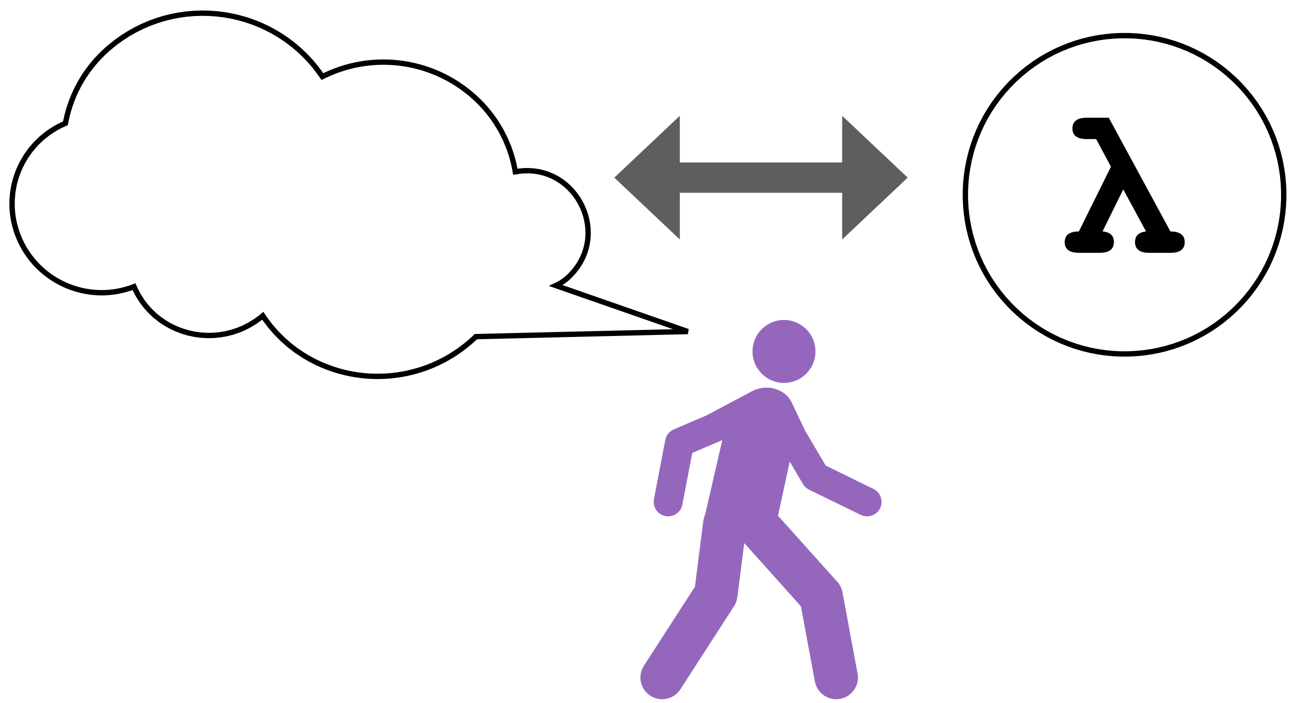


Spoken: I regard programmers as translators. They must translate a mental model into a programming language, and {next slide} vice-versa

Programmers are translators



Programmers are translators



Spoken: Many high-level programming languages prioritize ergonomics to make this translation process easier. For the programmer, an {entire computational workflow} can be modeled using {only type declarations}, without having to write any concrete code.

Key Idea 2

Type systems serve as a **framework** in which programmers **represent concepts**.

Spoken: Here's another key idea. (read.)

Concept representation in a type system

- What is a type system?
- Why should cognitive scientists care about types?
- **What constitutes the effects of learning?**
- What does this model lack?

What constitutes the effects of learning?

Spoken: We can *synthesize programs…*

What constitutes the effects of learning?

- Program synthesis

Spoken: ...from examples, or we can even use the type alone as a program synthesis task. For example:

What constitutes the effects of learning?

- Program synthesis

$$\textbf{sort} :: \textbf{Ord}\ T \Rightarrow (i : [T]) \rightarrow \left(v : [T] \mid \textbf{elems}(i) = \textbf{elems}(v) \wedge \textbf{nondecreasing}(v)\right)$$

```
sort = λxs . foldr f Nil xs
  where f = λt . λh . λacc .
    match acc with
      Nil → Cons h Nil
      Cons z zs → if h ≤ z
        then Cons h (Cons z zs)
        else Cons z (f zs h zs)
```

Spoken: In work by Polikarpova and others, a machine implemented sort when given an equivalent type-definition to the one I've shown you. (now slowly:) We can {start with the abstract idea of sort}, and {later} learn its implementation.

Polikarpova, Kuraj, & Solar-Lezama (2016)

What constitutes the effects of learning?

- Program synthesis
- Implementation-level refactoring

Spoken: Implementation-level refactoring can be performed by a learning process. For example:

What constitutes the effects of learning?

- Program synthesis
- Implementation-level refactoring

```
(define (add2 ℓ)
  (map (λ (x) (+ x 2))) ℓ)

(define (add3 ℓ)
  (map (λ (x) (+ x 3))) ℓ)
```

→

```
(define ((add-k k) ℓ)
  (map (λ (x) (+ x k))) ℓ)

(define add2 (add-k 2))
(define add3 (add-k 3))
```

Spoken: In collaboration with Kevin Ellis and others, who will be talking later today, we "compressed" common code into reusable helper functions, making useful concepts more accessible for future learning.

Ellis, Morales, Sablé-Meyer, Solar-Lezama, Tenenbaum (2018)

What constitutes the effects of learning?

- Program synthesis
- Implementation-level refactoring
- Type-level refactoring

Spoken: Type-level refactoring allows us to go...

What constitutes the effects of learning?

- Program synthesis
- Implementation-level refactoring
- Type-level refactoring

Incremental (Peano)

```
enum Nat {
    Zero,
    Succ(Nat),
}
let twenty: Nat = Succ(Succ(Succ(Succ(
                  Succ(Succ(Succ(Succ(
                  Succ(Succ(Succ(Succ(
                  Succ(Succ(Succ(Succ(
                  Succ(Succ(Succ(Succ(
                  Zero))))))))))))))))
                  ))))
```

Spoken: From a representation of natural numbers that is incremental…

What constitutes the effects of learning?

- Program synthesis
- Implementation-level refactoring
- Type-level refactoring

Incremental (Peano)

```
enum Nat {
    Zero,
    Succ(Nat),
}
let twenty: Nat = Succ(Succ(Succ(Succ(
                  Succ(Succ(Succ(Succ(
                  Succ(Succ(Succ(Succ(
                  Succ(Succ(Succ(Succ(
                  Succ(Succ(Succ(Succ(
                  Zero))))))))))))))))
                  ))))
```

Digital (Arabic numeral)

```
enum Digit {
    Zero,
    One,
    ...,
    Nine,
}
type Nat = [Digit];
let twenty: Nat = [Two, Zero];
```

Spoken: to a digital representation, as in the Arabic numeral system. A representation transformation like this corresponds to *conceptual change*.

What constitutes the effects of learning?

- Program synthesis
- Implementation-level refactoring
- Type-level refactoring
- Type generation

Spoken: Generating types, whether by {intentional learning} or by {creative imagination}, is fundamental to a type-based representation. For example:

What constitutes the effects of learning?

- Program synthesis
- Implementation-level refactoring
- Type-level refactoring
- Type generation

```
class Organism o where
    procreate :: ...   -- permits random mutation

type Environment = ...

evolution :: Organism o => (Environment, [o]) → (Environment, [o])
```

Spoken: Darwinian evolution can be discovered by creatively writing some types, and trying to resolve missing pieces with more types or by iterating on the definition existing types.

Concept representation in a type system

- What is a type system?
- Why should cognitive scientists care about types?
- What constitutes the effects of learning?
- **What does this model lack?**

What does this model lack?

- Learning the framework vs. learning within the framework

Spoken: We've been assuming a very sophisticated type system, but maybe it must be learned via a prototypical type system.

What does this model lack?

- Learning the framework vs. learning within the framework
  - what is innate?

Spoken: If the whole system is not innate, there must be faculty to learn it.

What does this model lack?

- Learning the framework vs. learning within the framework — what is innate?
- The language is formal

Spoken: If types, or "concepts", do not match, the type system does not {try harder} to {find a way} of fitting them — types either fit or they don't.

What does this model lack?

- Learning the framework vs. learning within the framework — what is innate?
- The language is formal
- Types must be fully formulated (no "holes")

Spoken: Types cannot have "holes" in their declarations, they must be completely valid. However, types can be iterated upon, as we saw earlier with the placeholder example.

Concept representation in a type system

Purpose:

- Learning programs ("child coder") is **more** than writing procedural code.
- Use type systems to **express meaning**, à la conceptual role semantics.
- Type systems provide a good representation for a computational study of **concept learning**.

Lucas E. Morales
lucasem@mit.edu
Learning as Program Induction, CogSci 2018