Arcs Tech Talk - Types and Arcs Recording

Types for fun and profit*

Part 1: Types and the Arcs Type Lattice

go/arcs-types-for-fun-and-profit

jopra@

Please call me ‘J’, they, them

*Neither fun nor profit are guaranteed, conditions may apply :P

(Just to make sure we're on the same page)

Note: some of this is open to interpretation and I do make mistakes, please feel free to stop me if something seems off or there are questions.

What do I mean when I say ‘Type’

Broadly: “a category of people or things having common characteristics.”

- Oxford Languages and Google - English

What do I mean when I say ‘Type’

In Programming: “a category of ‘values’ having common characteristics.”

Aside: the definition of value is almost as vague as thing, but it makes me feel better Wikipedia: Value

What do I mean when I say ‘Type’

In Programming: “a category of ‘values’ having common characteristics.”

Examples from Kotlin:

```
val x: Int = 3

val value: (it: String) -> Int = {it.length}
```

What do I mean when I say ‘Type’

In Programming: “a category of ‘values’ having common characteristics.”

Examples from Arcs:

x: Text

oncall: reads Employee {name: Text, employee_id: Long, uname: Text}

What do I mean when I say ‘Type’

A syntactic phrase/expression that describes what ‘actions’ can be performed on/with a value.

Examples from Arcs in Kotlin:

```
with                    oncall: reads Employee {name: Text, employee_id: Long, uname: Text}
println("${oncall.name}")
println("${oncall.employee_id}")
oncall.name = "Some other name" // May be allowed but doesn't write to the DB
```

What do I mean when I say ‘Type’

A syntactic phrase/expression that describes what functions can be called with a value as an argument (+/receiver).

Examples from Arcs in Kotlin:

```
// with oncall: reads Employee {name: Text, employee_id: Long, uname: Text}
println(“${oncall.name}”)
println(“${oncall.employee_id}”)
oncall.name = “Some other name” // May be allowed but doesn’t write to the DB
```

What do I mean when I say ‘Type system’

A model that relates Types to the set of allowed/expected behaviours in a context (normally a programming language).

This links syntax with semantics

What do I mean when I say ‘Type theory’

The study of Types & Type systems

What do I mean when I say ‘a Type theory’

A model that relates Types to the set of allowed/expected behaviours.

This links syntax with semantics.

What do I mean when I say ‘a Type theory’

Normally you can just think about a Type theory as a family of Type systems that share a bunch of properties, rules and conventions

But what are Types for?

SAFETY

But what are Types for?

But what is Type Safety?

[Type] Safety: Progress + Preservation

Progress: A well typed program will not get into an invalid state
This means will work until it termination or error (infinite loops & errors typically don't count)

Preservation: Evaluation won't break any 'rules' of the type system

In other words: Safety means that the language won't violate it's own abstractions

But what is Type Safety?

[Type] Safety: Progress + Preservation

Mostly we care about preservation.

We're interested in catching human error, before runtime to minimize the exhaustiveness of testing needed to gain confidence that a program is correct.

But what is Type Safety?

But what is Type Safety for?

Bugs often caught using Types:

- Illegal operations (i.e. division by string, negation of unsigned integers, array access into an Int etc.)
- Buffer overflow
- Logic errors:
  - e.g. Mar climate orbiter vs units, units: A domain-specific type system for dimensional analysis
- Memory safety (Rust can catch use after free at compile time [at a ‘small’ cost])

There’s also some nice side benefits:

- Intellisense + type directed look up + feedback
- Rust & Haskell’s derivation systems for generating code where needed based on types

Note: **Some languages let you break the rules**

Note: **Some languages have very flexible rules**

Note: **Some languages don’t have (type based) rules**

A refresher on Arcs' Type system

- Primitive Types:
  - Text, URL, Number, BigInt, Boolean, Bytes, Instant, Duration
  - Kotlin Types: Byte, Short, Int, Long, Char, Float, Double
- Types:
  - Unions: (t or s or u)
  - Tuples [Products]: (t, s)
  - Collections: [t]
  - Ordered Lists: List<t>
  - Singletons: ![e]
  - Nullables: t?
  - References: &e
  - Refinements: e [p]
- Schemas + Entity Types:

  Name { field: t }

Lesser known Types:

- Interfaces
- Mux type: #t
- Slot type: Slot {}
- Big Collection type: BigCollection<t>
- Type variables: ~t
- Constrained Type variables: ~t with u
- NullType (Paxel)
- Object
- More?

Subtyping / Assignability

Assuming there are two types *t* and *u*. *t* is a subtype of u where any value of *u* is also a value of *t*.

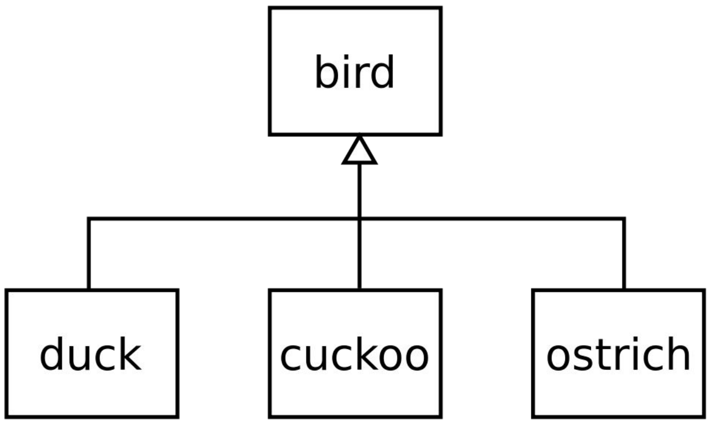
Example:
t = Int, u = Long.
Int is a subtype of Long

x: Int = 3

y: Long = 3

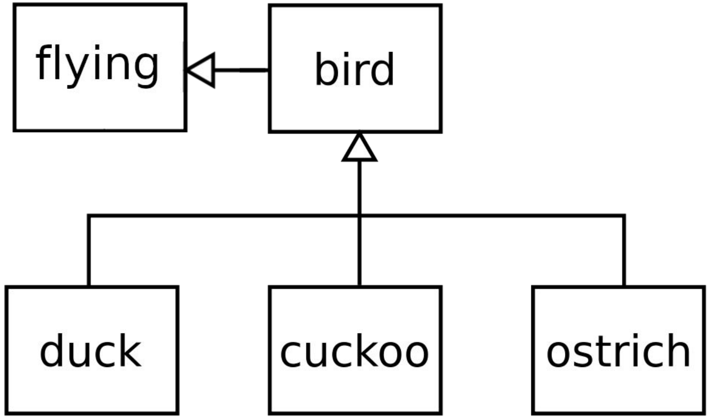
Subtyping / Assignability

Another example:
All ducks are birds, so you can accept a duck wherever you need a bird.



Subtyping / Assignability

Counter example:
All birds can fly, so you can accept a bird wherever you need something that can fly.



Subtyping / Assignability: type safe 'duck' types

Subtypes + Supertypes

We can write ‘X is a subtype of Y’ as

$$X <: Y$$

Or

$$X < Y$$

Subtypes + Supertypes

It’s a pre**order** just like ‘<’, so there’s a corresponding ‘>’

We can write ‘X is a supertype of Y’ as

$$X :> Y$$

Or

$$X > Y$$

Meet = (greatest) shared subtype

If two types share a subtype*, that's called the *meet* (or *infinimum*)

i.e.

If $s :> j$, $t :> j$

then

j is the meet of s and t

The meet type meets the requirements of both s and t

Meet = (greatest) shared subtype

e.g.

The meet of s and t

Where s = Person Named {name: Text, fav: Colour} and t = Employee Named {name: Text, id: BigInt} is

Named {name: Text}

The meet type meets the requirements of both s and t

Join = (least) shared supertype

e.g.

The meet of s and t

Where s = Person {name: Text} and t = Employee {id: BigInt} is

Employee Person {id: BigInt, name: Text}

*The join type **joins** s and t and guarantees only things that are true of both*

Join = (least) shared supertype

If two types share a supertype*, that's called the *join* (or *supremum*)

i.e.

If $s <: j$, $t <: j$

then

j is the join of s and t

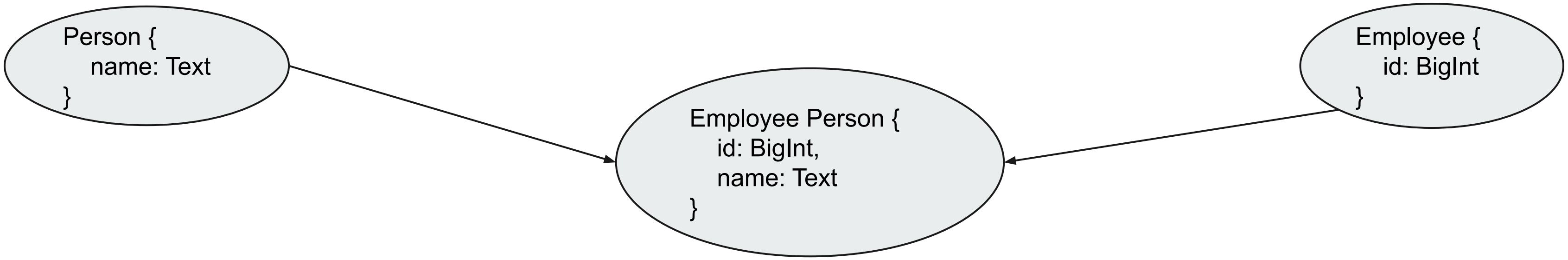
*The join type **joins** s and t and guarantees only things that are true of both*

Subtypes + Supertypes

These relationships give us a ‘lattice’

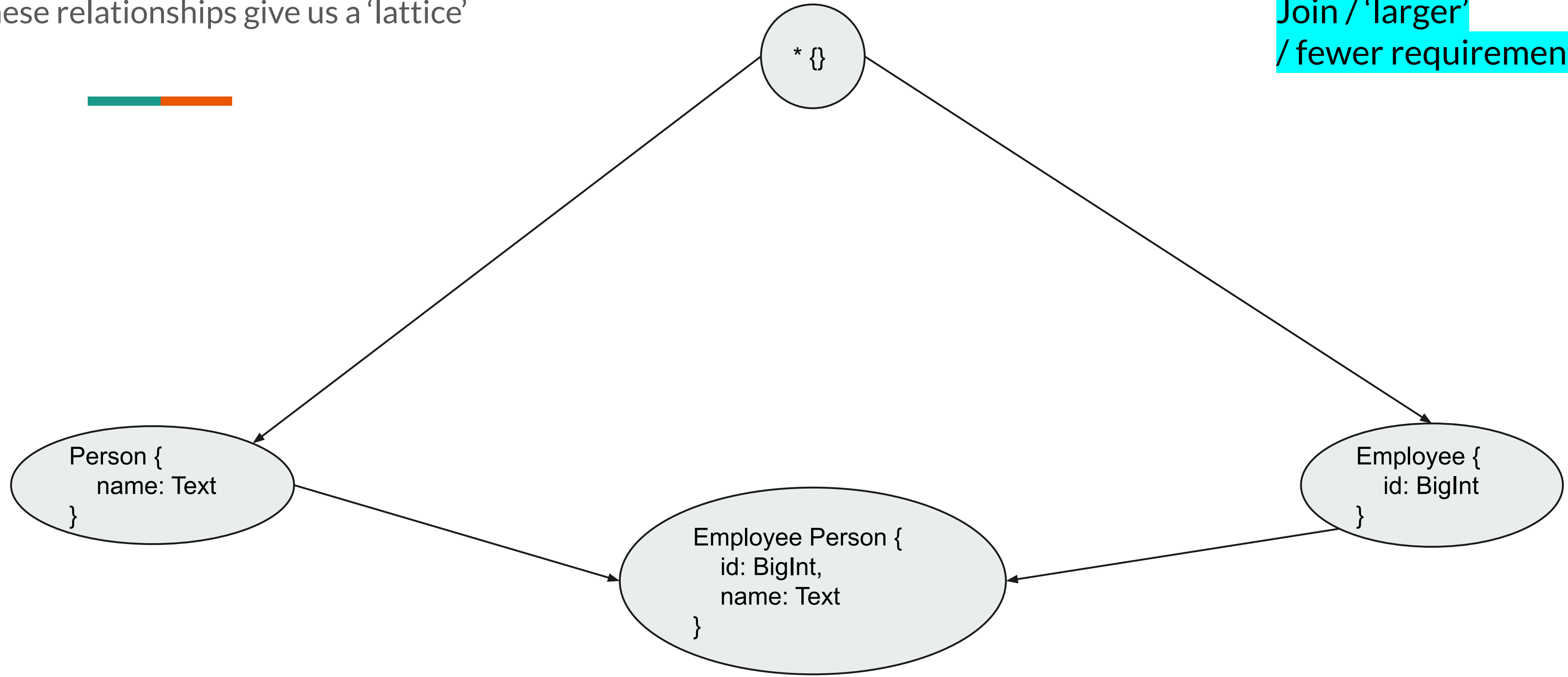
X ⟶ Y
means
X :> Y

Join / ‘larger’
/ fewer requirements



Meet / ‘smaller’
/ more requirements

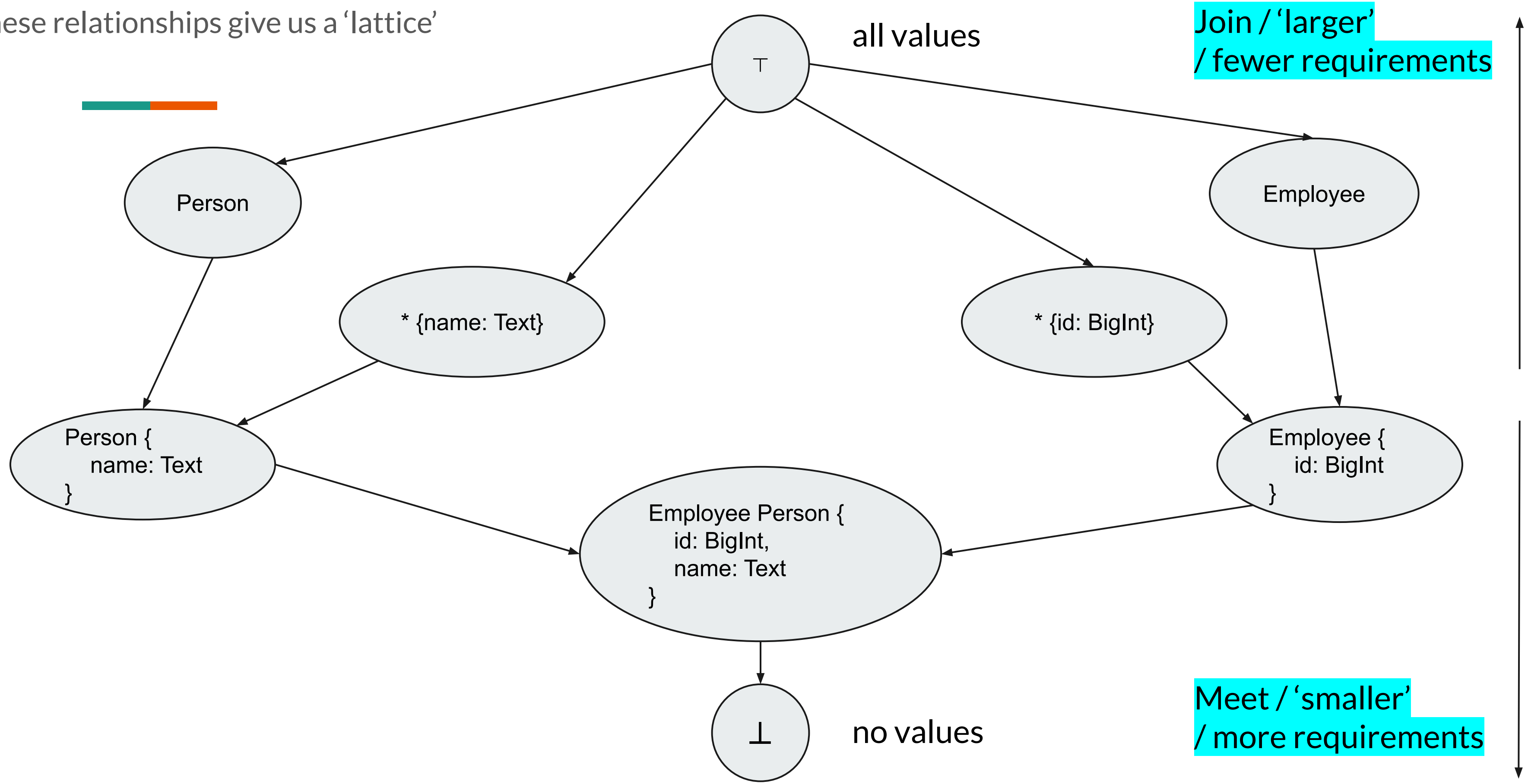
These relationships give us a 'lattice'



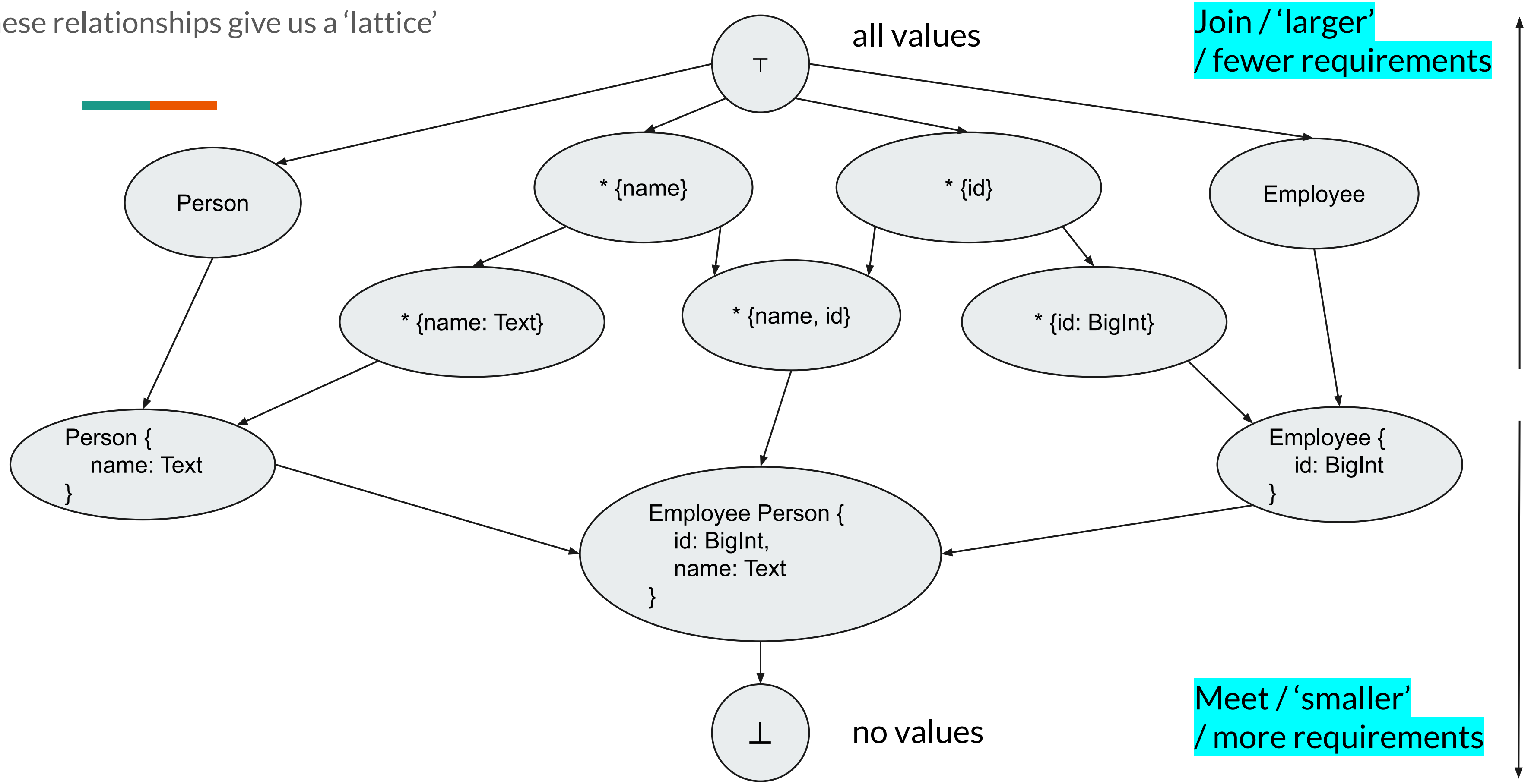
Join / 'larger'
/ fewer requirements

Meet / 'smaller'
/ more requirements

These relationships give us a 'lattice'



These relationships give us a ‘lattice’



What can you do with types?

BUILD MORE TYPES

What can you do with types?

- Refinement types
- Union types (oneof / unions)
- Product types (tuples, records, etc.)
- Dependent types
- Function types
- Intersection types
- Session types

What can you do with types?

- Refinement types
- Union types (oneof / unions)
- Product types (tuples, records, etc.)
- Dependent types
- Function types
- Intersection types
- Session types

Legend

We have these

We could have these

But that’s a topic for another talk!

What are we doing with Types?

Recent work:

- BigInt support
- Instant support

In progress:

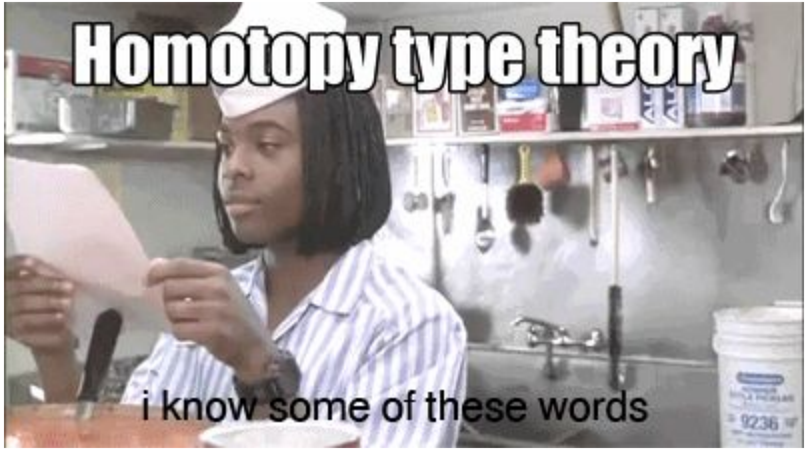
- Arcs Nullables
- BigInt, Instant and Duration Refinement Support

For the future:

- Arcs Requests for Type Features in October of 2020
- Relaxed Reads and Writes

Sources + Further reading

- The Arcs Type System - go/arcs-type-system
- Types and Programming Languages (Free PDF)
- On the Expressive Power of Prog. Languages
- Join & Meet wiki
- nLab: type theory in nLab
- Particularly exciting new research is in Intuitionistic and homotopy type theory



Sources + Further reading from Gogul

- Type Inference in Arcs
- Formal Methods in Arcs Dataflow Analysis
- Types & References

That’s all folks

Thanks for coming. Questions?