Spooking out illegal states with 👻 Types

@ostera

PLT / DistSys / UX

at Spotify

Data.

Rules.

Data

```
type ⚙️ = {
  maxCount: 🔢
};
```

Rules

```
let isValid : ⚙️ → 🚩
```

Rules

```
let isValid : ⚙️ → 🚩 = fun
  | { maxCount: c } when c > 0 ⇒ 👌
  | _ ⇒ 👎🏼
```

Data

```
let 🎃: ⚙️ = {
  maxCount: -20
};
```

```
let ☠️: ⚙️ = {
  maxCount: 10,
};
```

Rules

`isValid(🎃)` ⟹ ❌

`isValid(☠️)` ⟹ ✅

Rules

`isValid(`🎃`)` ⟹ ❌

`useConfig(`🎃`)` ⟹ 🏃‍♀️⏱️🐞

Type-Level Rules

```
useConfig(🐢)

Type Error: expected ⚙️
but found 🐢
```

Can we take these rules up to the Type Level?

🤔

Phantom Types

“A **phantom type** is a parametrised type whose parameters do not all appear on the right-hand side of its definition.”

– Haskell Wiki (https://wiki.haskell.org/Phantom_type)

Data

```
type t(🦁, 👻) =
  | A(🦁);
```

Data

```
type t(🦁, 👻) =
  | A(🦁);
```

👻 appears only on the left.

Why would this ever be useful?

🤔

“The fact that the phantom parameter is unused gives you the freedom to use it to encode additional information about your types.”

– Jane Street’s Tech Blog (https://blog.janestreet.com/howto-static-access-control-using-phantom-types/)

Data

```
let a: t(⚙️, 🎉) = A(⚙️);
let b: t(⚙️, 🦄) = A(⚙️);
```

Data

```
let a: t(⚙️, 🎉) = A(⚙️);
let b: t(⚙️, 🦄) = A(⚙️);
```

These are different
at the type level!

Data

Type variable!

Fixed type!

```
let useT: t(🎁, 🎉) = fun
  | A(🎁) ⇒ …do stuff…
```

Data

```
let useT: t(🎁, 🎉) = fun
  | A(🎁) ⇒ …do stuff…
```

`useT` is defined for any `t(🎁, 🎉)`.

Data

```
let a: t(⚙️, 🎉) = A(⚙️);
let b: t(⚙️, 🦄) = A(⚙️);

useT(a);
useT(b);
```

Data

```
let a: t(⚙️, 🎉) = A(⚙️);
let b: t(⚙️, 🦄) = A(⚙️);

useT(a); // 🙌 yes work!
useT(b); // 🤦‍♀️ no work!
```

Data

```
useT(b); // 🤦‍♀️ no work!

Type Error:
→ expected t(⚙️, 🎉)
→ found t(⚙️, 🦄)
```

“**Phantom types** are useful when you want to deal with different kinds of data that have the same representation but should not be mixed.”

– Martin Jambon’s Quota Answer (https://www.quora.com/What-are-good-applications-of-phantom-types)

Back to ⚙️

Data

```
type ⚙️ = {
  maxCount: 🔢
};
```

Data

```
type ⚙️(👻) = {
  maxCount: 🔢
};
```

Data

```
type ⚙️(👻) = {
  maxCount: 🔢
};

type ✅;
type 🤷‍♂️;
```

Brief Detour into Abstract Types

Abstract Types

```
type ✅;
```

Abstract Types

No constructors!

Has a name.

```
type ✅;
```

Abstract Types

```
let a: ✅ = ??
```

Can’t create data!

Abstract Types

`type` ✅;

Exists only on the type-level.

Abstract Types

```
let f: ✅ → 🚩;
```

Can use it on type signatures.

Abstract Types

```
let f: ✅ → 🚩;
```

We can’t create values of ✅ so we can’t call f

Why would this ever be useful?

🤔

Abstract Types

```
let f: ✅ → 🚩;
```

They helped us make f impossible to call!

Back to ⚙️

Data

```
type ⚙️(👻) = {
  maxCount: 🔢
};

type ✅;
type 🤷‍♂️;
```

Data

```
let 🎃: ⚙️(🤷‍♂️) = {
  maxCount: -20
};

let ☠️: ⚙️(✅) = {
  maxCount: 10,
};
```

Rules

```
let useConfig : ⚙️(✅) → 🌮
```

Type-Level Rules

```
useConfig(🎃)

Type Error:
→ expected ⚙️(✅)
→ found ⚙️(🤷‍♂️)
```

Data

```
let 🎃: ⚙️(✅) = {
  maxCount: -20
};
```

```
useConfig(🎃) ⟹ 🏃‍♀️⏱️🐞
```

Rules

```
let isValid : ⚙️ → 🚩
```

Rules

let ~~isValid~~ : ⚙️ → 🚩

Rules

```
let make : 🔢 → ⚙️(🤷‍♂️)
```

Rules

```
let make : 🔢 → ⚙️(🤷) =
    n ⇒ { maxConfig: n }
```

Data

```
let 🎃: ⚙️(🤷‍♂️) = make(-20);

let ☠️: ⚙️(🤷‍♂️) = make(10);
```

Data

```
let 🎃 = make(-20);

let ☠️ = make(10);
```

Type-Level Rules

```
useConfig(🎃)
useConfig(☠️)

Type Error:
→ expected ⚙️(✅)
→ found ⚙️(🤷‍♂️)
```

Rules

```
let validate : ⚙️(🤷‍♂️) → 🎁(⚙️(✅))
```

Rules

```
let validate : ⚙️(🤷‍♂️) → 🎁(⚙️(✅)) =
  fun
  | { maxCount: c } when c > 0 ⇒
    🎁(⚙️(✅))
  | _ ⇒ 🗑️
```

Data

```
let 🎃 = make(-20);

switch(validate(🎃)) {
  | Some(👍) ⇒ useConfig(👍)
  | None
};
```

Will not get called!

1. Our types carry validation information automatically through our program. 🍔

2. You can’t create invalid data with valid types.* 🚑

3. Compile time checks for our program rules! 🎉

Spook out your illegal states.

Thank you.

@ostera