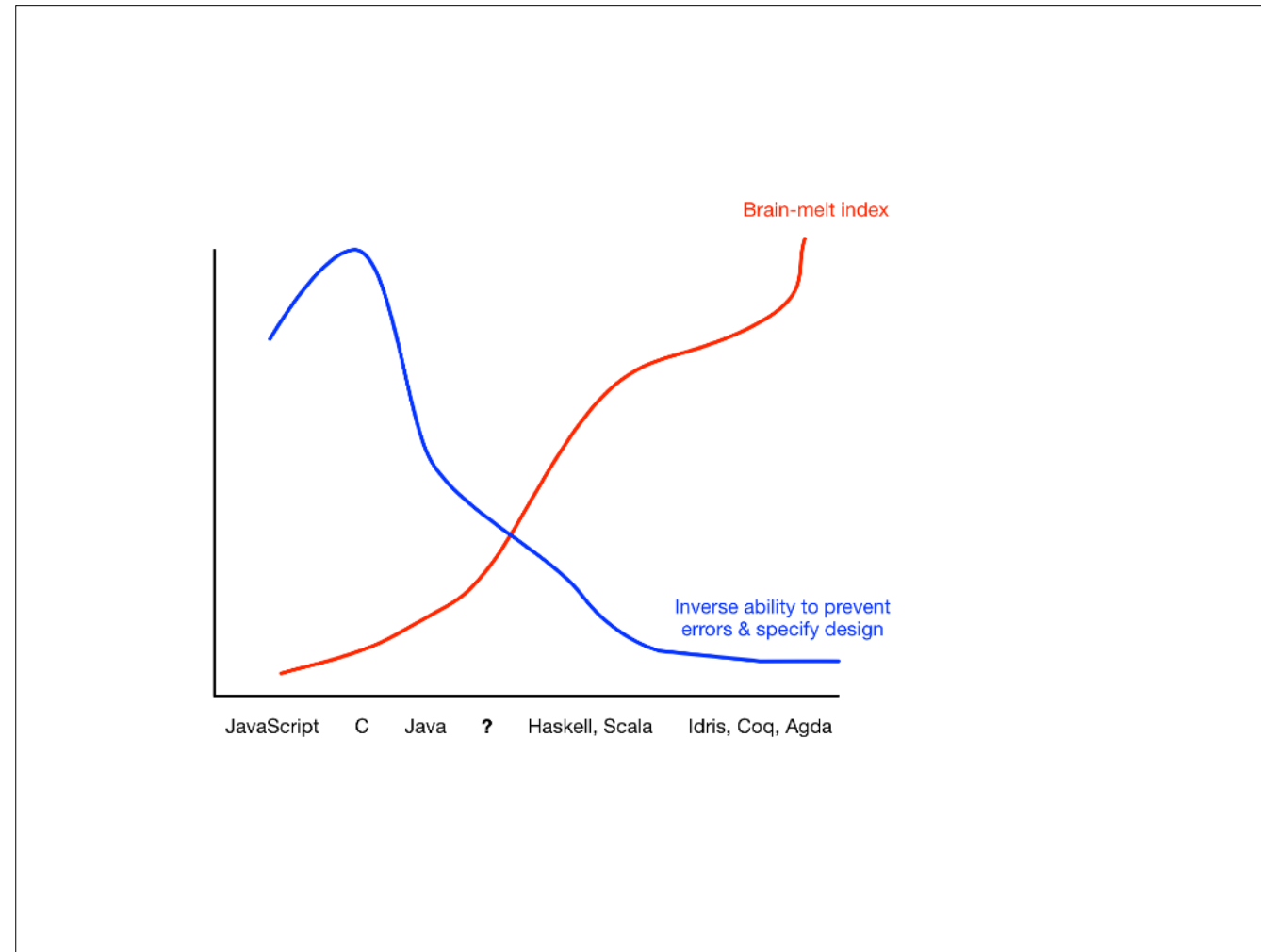
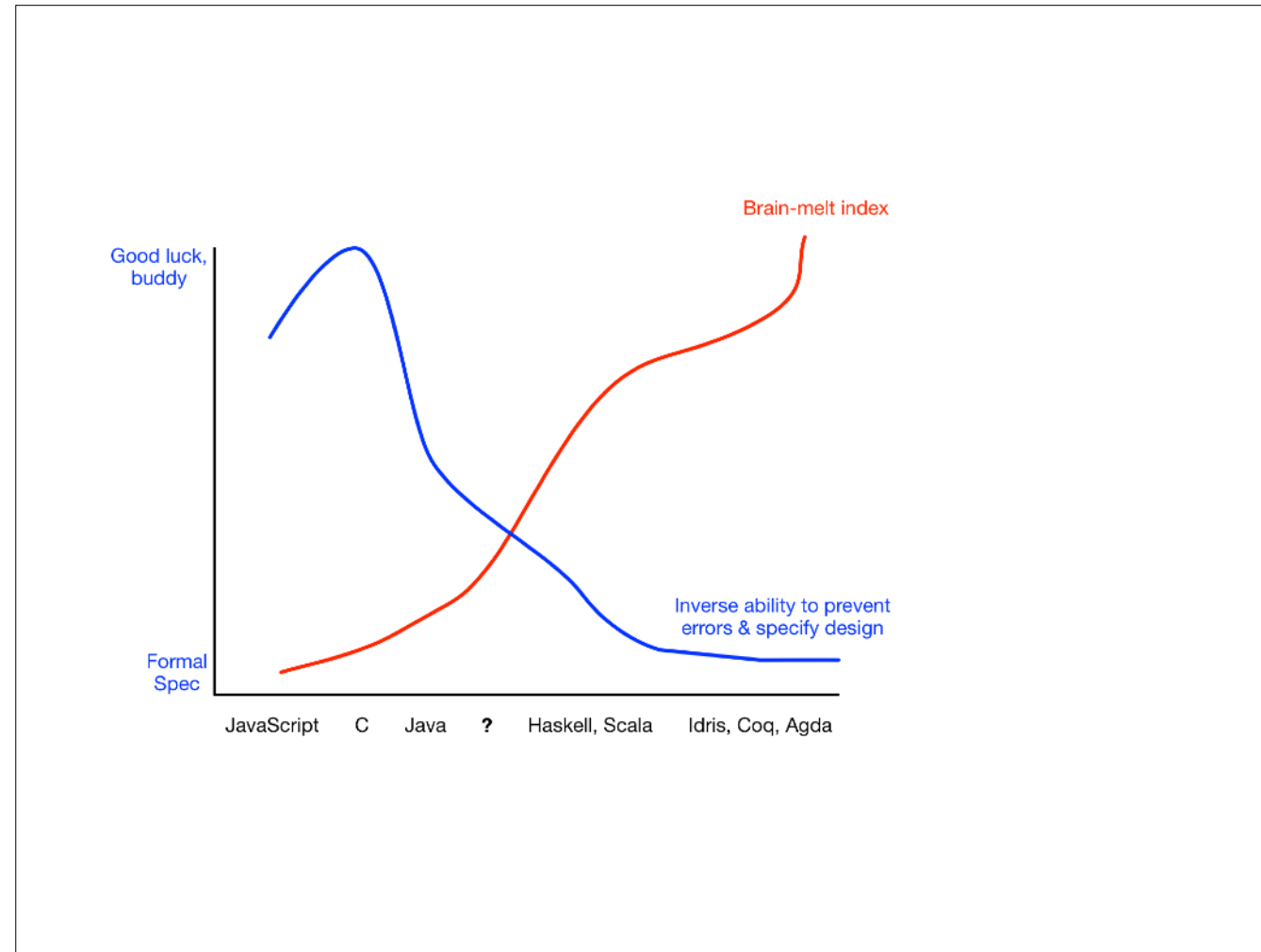


“X” Marks the Spot

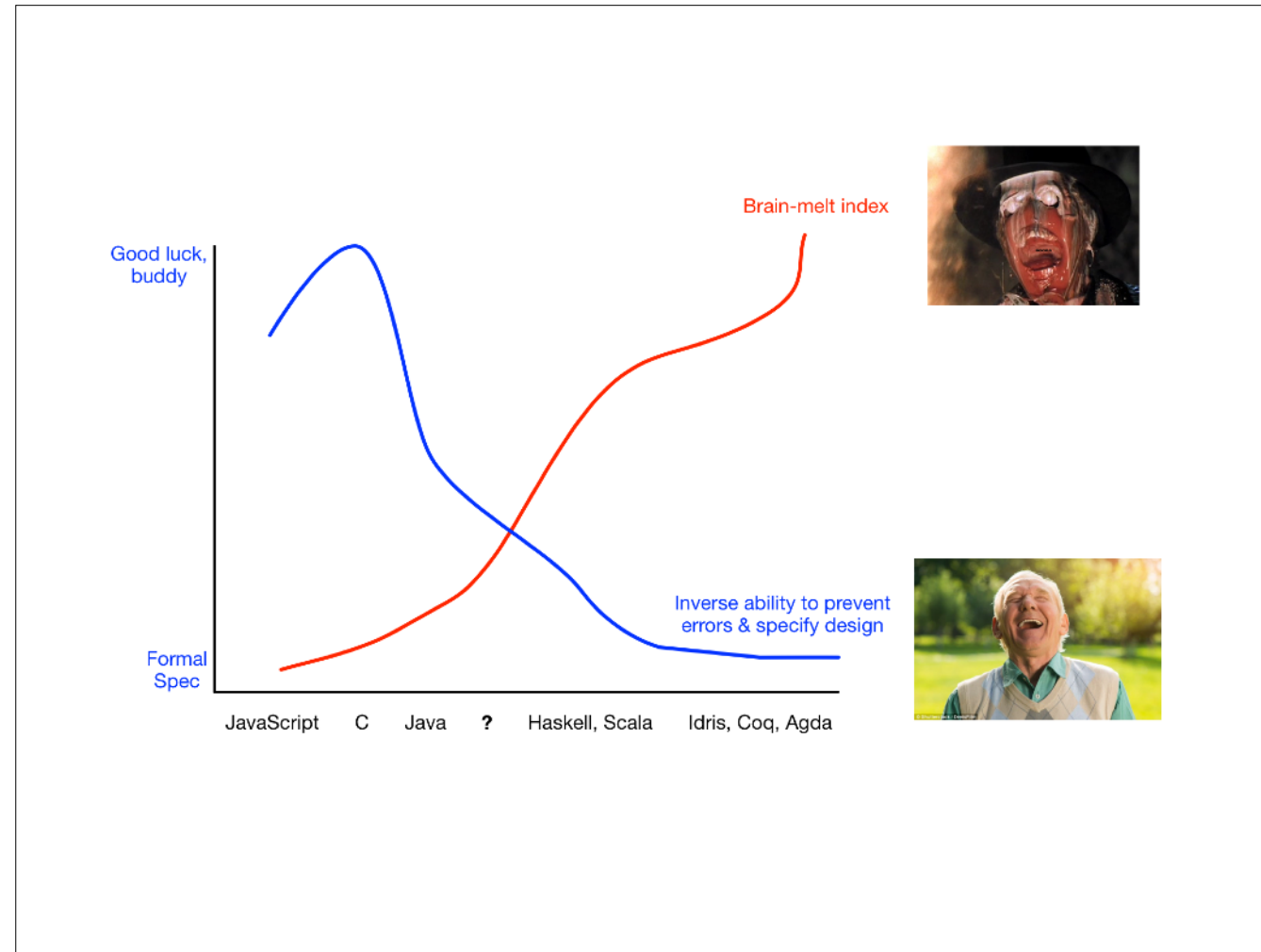
(Here's a chart I drew)



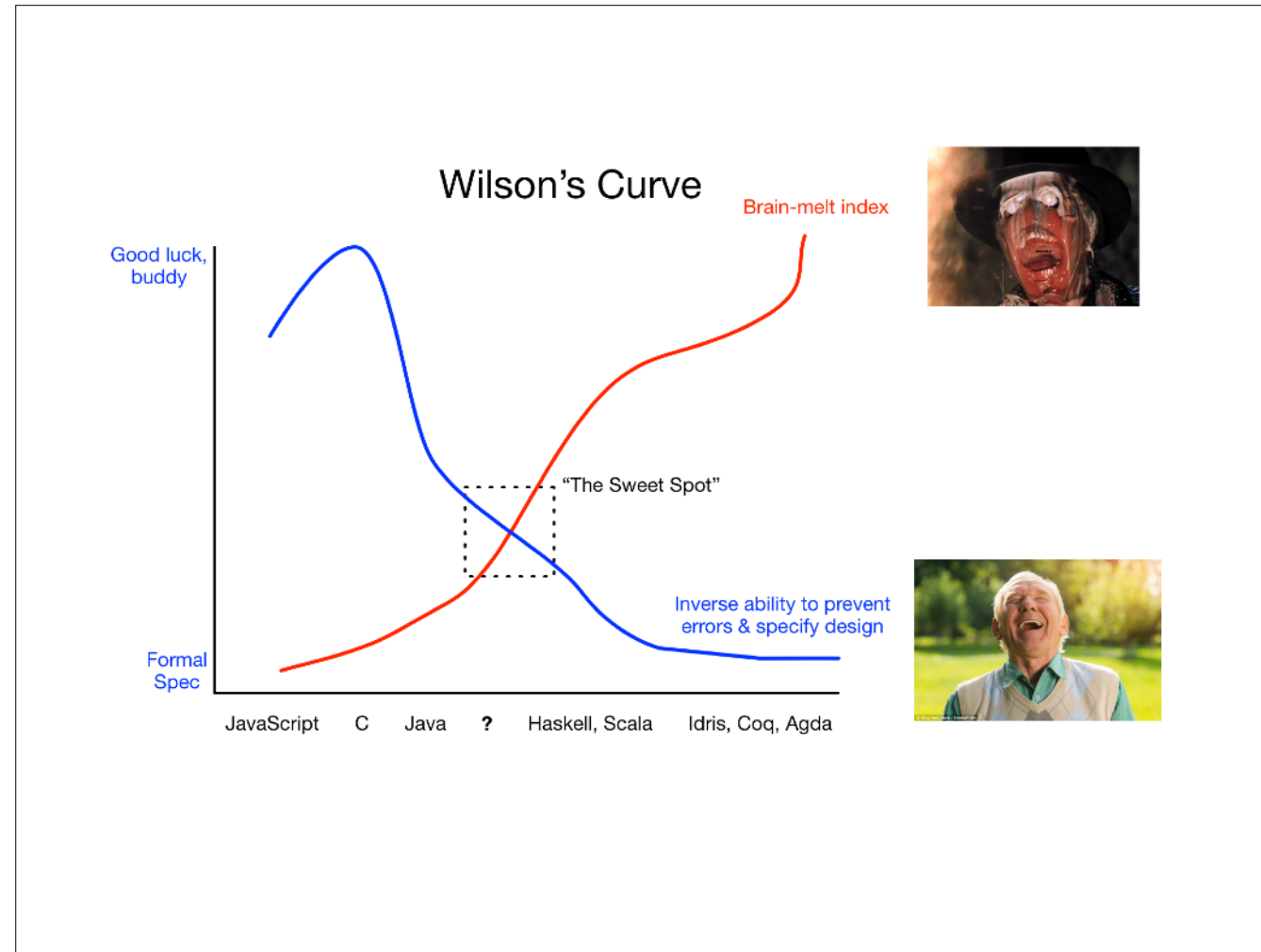
There are a series of languages across the bottom ranked roughly according to type system complexity. (Also, this is mostly a joke)



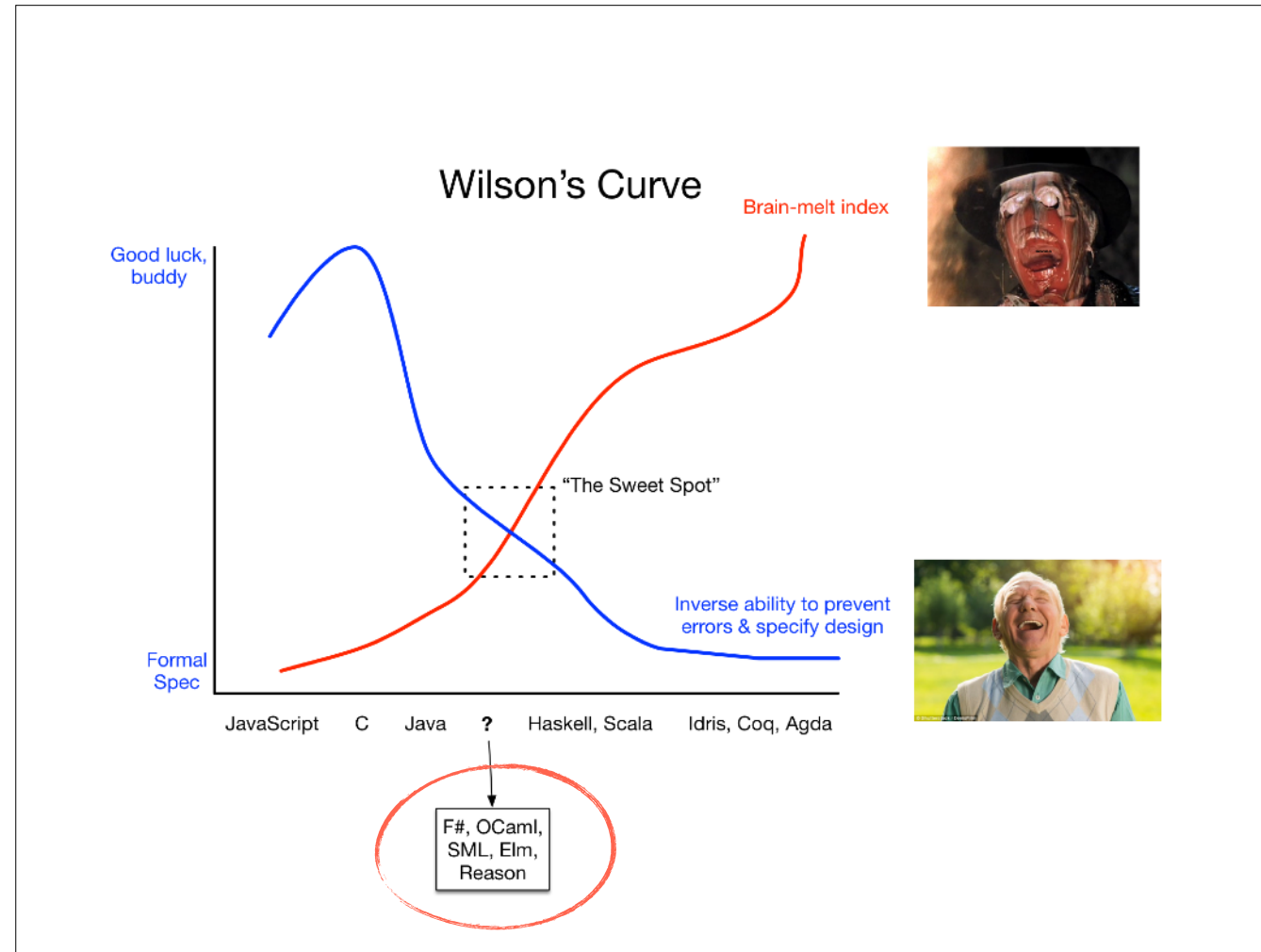
The blue line represents an inverse relationship. That is, as the line decreases, more and more things can be ruled-out at the language level. You can write down more detailed type information and more involved specification. This is also more finely-grained, you'll spend more time doing it.



Here I've added a scale to the red line, moving up and to the right. At the bottom, you're untroubled by the learning curve, sure there are some WATs, but you're also not writing proofs. If you're familiar with CBLs, you'll be in familiar territory at the left. At the upper end, your face melts off. Or maybe that's your jam (it's kinda mine).



Naturally, there's a spot where these lines cross. You can write down interesting properties of your program and squash some beefy bugs at the language level. Yet, you also haven't started the steepest ascent along the brain-melt curve. Roughly in this area, there's a mysterious language. It's beyond Java, yet it's not Haskell or Scala -- in no way taking a swipe at either of those.



We yank off the mask and see that there's really a whole class of languages here. These -- and many more like them -- share a bunch of common features. They're nestled on primo real estate in the language arena.

“Programming languages near the "X" are a strong fit for general application development of the kind that most developers do.”

–Me, just now

**More on that later, now
let's talk about
something else...**

File that claim away as "unproven." I'll talk more about it later.


- ~~[humanizing joke to open]~~
- ~~[Reveal thesis]~~
- **Introduce Reason**
- Introduce ReasonReact

Okay, now you are “HERE.”

I’ve broken this talk into a few main parts. I want to introduce Reason & ReasonReact as really interesting technologies to watch. Then I also want to talk about some bigger ideas in software development.

I can summarize because I hear that in talks you should first tell people what you’re going to tell them, then tell them, and then tell them what you told them. It sounds funny, but watch for it!

The gist is this: in modern software development, we’re leaving a lot of correctness & information on the table — we’re not taking enough advantage of the tools that exist. Dynamic languages don’t let us write down properties of our program that we as developers know about and so we have to lean more heavily on testing. We can and should capture more business logic in the types we use to model.

- ~~[humanizing joke to open]~~
 - ~~[Reveal thesis]~~
 - **Introduce Reason**
 - Introduce ReasonReact
- 

Okay, now you are “HERE.”

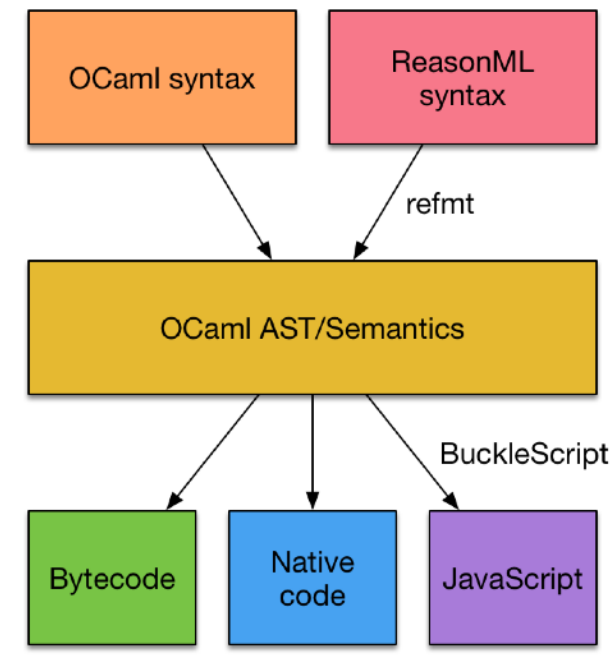
I’ve broken this talk into a few main parts. I want to introduce Reason & ReasonReact as really interesting technologies to watch. Then I also want to talk about some bigger ideas in software development.

I can summarize because I hear that in talks you should first tell people what you’re going to tell them, then tell them, and then tell them what you told them. It sounds funny, but watch for it!

The gist is this: in modern software development, we’re leaving a lot of correctness & information on the table — we’re not taking enough advantage of the tools that exist. Dynamic languages don’t let us write down properties of our program that we as developers know about and so we have to lean more heavily on testing. We can and should capture more business logic in the types we use to model.



Now I'm going to talk about ReasonML.



A sort of "man-in-the-middle" (kind and gentle) attack on JavaScript!



There are a lot of tools in a modern JS app. A key feature of Reason is that it brings many of these tools into either the language itself or tools that it ships with.

Reason is typed and thus it fills the role of TS or Flow. ReasonReact includes features that work much like redux. Prettier is replaced by the same tool that converts Reason into OCaml code, refmt. JSX syntax is supported via a built-in syntax extension system in OCaml.

Much of the supporting tooling is brought into the language.



There are a lot of tools in a modern JS app. A key feature of Reason is that it brings many of these tools into either the language itself or tools that it ships with.

Reason is typed and thus it fills the role of TS or Flow. ReasonReact includes features that work much like redux. Prettier is replaced by the same tool that converts Reason into OCaml code, refmt. JSX syntax is supported via a built-in syntax extension system in OCaml.

Much of the supporting tooling is brought into the language.

Reason

- A new front-end syntax for OCaml (with a toolchain and ecosystem)
- Created by Facebook
- Supports React (ReasonReact)
- Compiles to:
 - Native
 - Bytecode
 - JavaScript

OCaml itself goes back a while. Wikipedia says it dates from 1996. The ML family itself goes back to the 70s.

New language!?

- Yes. *But*, Reason has been created to layer a more familiar syntax, one similar to ES6, onto OCaml
- Types can take a little while to learn, but this is similar to learning how to best use TypeScript or Flow

Wadler's Law

In any language design, the total time spent discussing a feature in this list is proportional to two raised to the power of its position.

0. Semantics

1. Syntax

2. Lexical syntax

3. Lexical syntax of comments

It's a snarky hot-take. But think of it the other way around: If you can effectively disarm 3 through 1, then you've tricked everyone into talking about what matters, semantics.

If your language looks pretty much like JS, then people will evaluate it on other features.

See also “The Language Strangeness Budget” by Steve Klabnik.

Features

- Flexible type system while still being fully inferred
- Records
- *Real* modules
- Fast type checking & compilation (provided ReasonReact “hello world” takes .8 seconds)

Cheat sheet

- A little long for a talk, but a quick glance:
- <https://reasonml.github.io/docs/en/syntax-cheatsheet.html>
- tl;dr it *looks* quite similar to ES6

Lightweight

```
type planet = {  
  name: string,  
  house: string,  
  spice: int,  
};  
  
let dune = {name: "Arrakis", house: "Atreides", spice: 100_000_000};  
  
let geidiPrime = {name: "Giedi Prime", house: "Harkonnen", spice: 0};  
  
let _ = Js.log(dune.name);  
let _ =  
  if (geidiPrime.spice > 0) {  
    Js.log("uh-oh");  
  } else {  
    Js.log("[sad trombone]");  
  };  
};
```

Lightweight

```
// Generated by BUCKLESCRIPT VERSION 4.0.0, PLEASE EDIT WITH CARE
'use strict';

console.log("Arrakis");

console.log("[sad trombone]");

var dune = /* record */[
  /* name */"Arrakis",
  /* house */"Atreides",
  /* spice */100000000
];

var geidiPrime = /* record */[
  /* name */"Giedi Prime",
  /* house */"Harkonnen",
  /* spice */0
];

exports.dune = dune;
exports.geidiPrime = geidiPrime;
/* Not a pure module */
```

Deep compiler smarts, (or trickery). Types are completely compiled away. Records become Arrays. The compiler does lots of constant folding and dead code elimination.

The next few slides will be a few techniques that you can employ because you have a nice type system. You could call these “Type System Hacks” if you want.

Make illegal states unrepresentable (Making impossible states impossible)

```
module Money: {  
  type t('a);  
  let create: int ⇒ t('a);  
  let add: (t('a), t('a)) ⇒ t('a);  
} = {  
  type t('a) = int;  
  let create = amt ⇒ amt;  
  let add = (m1, m2) ⇒ m1 + m2;  
};  
  
type usd;  
type gbp;  
  
let tenQuid: Money.t(gbp) = Money.create(10);  
let tenBucks: Money.t(usd) = Money.create(10);  
  
let twenty = Money.add(tenBucks, tenBucks);  
let bad = Money.add(tenBucks, tenQuid); /* doesn't compile */
```

Money is using an identical representation for the underlying value. But, the value is marked at the type level with some extra information.

The technique going on here is about the fanciest that I would use in production code. In the above code “a” is a “phantom” type — because the type only exists at the type level. There’s no value that has type “usd” or “gbp.” Those types are said to be “uninhabited.”

Make illegal states unrepresentable (Making impossible states impossible)

```
module Money: {  
  type t('a);  
  let create: int ⇒ t('a);  
  let add: (t('a), t('a)) ⇒ t('a);  
} = {  
  type t('a) = int;  
  let create = amt ⇒ amt;  
  let add = (m1, m2) ⇒ m1 + m2;  
};  
  
type usd;  
type gbp;  
  
let tenQuid: Money.t(gbp) = Money.create(10);  
let tenBucks: Money.t(usd) = Money.create(10);  
  
let twenty = Money.add(tenBucks, tenBucks);  
let bad = Money.add(tenBucks, tenQuid); /* doesn't compile */
```



Money is using an identical representation for the underlying value. But, the value is marked at the type level with some extra information.

The technique going on here is about the fanciest that I would use in production code. In the above code “a” is a “phantom” type — because the type only exists at the type level. There’s no value that has type “usd” or “gbp.” Those types are said to be “uninhabited.”

Make illegal states unrepresentable (Making impossible states impossible)

```
function create(amt) {  
  return amt;  
}  
  
function add(m1, m2) {  
  return m1 + m2 | 0;  
}  
  
var Money = /* module */[  
  /* create */create,  
  /* add */add  
];  
  
var twenty = 20;  
var tenQuid = 10;  
var tenBucks = 10;  
  
exports.Money = Money;  
exports.tenQuid = tenQuid;  
exports.tenBucks = tenBucks;  
exports.twenty = twenty;  
/* No side effect */
```

Here we can see that:

1. We didn't get "bad," it won't compile
2. Type info really is gone. "Add" and "create" don't really do anything (types are "erased")

Lots of cool uses!

```
module type User = {  
  type t('a);  
  type anonymous;  
  type admin;  
  type regular;  
  /* some API omitted */  
  let login: (username, email) => t(anonymous);  
  let authenticate: (t(anonymous), password) => option(either(t(admin), t(regular)));  
  let deleteDatabase: t(admin) => unit;  
};
```

We can apply this same phantom type technique to restrict privileges in a way that the compiler can check. Here we have a user type that carries a phantom “user type” tag. We can then define critical functionality to only accept users with the right privileges.

A fresh guest user is anonymous until we check their authentication. This grants that user either admin status or regular status (or they didn’t log in successfully). We can then limit the delete database action to admin-only.

Lots of cool uses!

```
module type User = {  
  type t('a);  
  type anonymous;  
  type admin;  
  type regular;  
  /* some API omitted */  
  let login: (username, email) => t(anonymous);  
  let authenticate: (t(anonymous), password) => option(either(t(admin), t(regular)));  
  let deleteDatabase: t(admin) => unit;  
};
```

User is initially anonymous

Must handle these cases

Only admins can delete the DB

We can apply this same phantom type technique to restrict privileges in a way that the compiler can check. Here we have a user type that carries a phantom “user type” tag. We can then define critical functionality to only accept users with the right privileges.

A fresh guest user is anonymous until we check their authentication. This grants that user either admin status or regular status (or they didn’t log in successfully). We can then limit the delete database action to admin-only.

Lots of cool uses!

```
let chris = {
  open User;
  let anon = login(u("cjw"), e("cwilson@8thlight.com"));
  switch (authenticate(anon, p("admin"))) {
  | Some(Left(usr)) => usr
  | Some(Right(_)) => failwith("Wrong password!")
  | _ => failwith("Chris couldn't log in!")
  };
};

let eve = {
  open User;
  let anon = (login(u("eve"), e("eve@example.com")));
  switch(authenticate(anon, p("regular"))) {
  | Some(Right(usr)) => usr
  | Some(Left(_)) => failwith("not allowed!")
  | _ => failwith("Even couldn't log in")
  }
};

let _ = User.deleteDatabase(chris);
/* let _ = User.deleteDatabase(eve); doesn't compile! */
```

Here we're using this technique. In a real app we'd present user-facing results when someone fails to log in. But the compiler helps us make sure to check this situation.

Note that when we write code where Eve tries to delete the database, the compiler is able to catch this because she's only logged in as a regular user!

Lots of cool uses!

```
    User.t(admin)
let chris = {
  open User;
  let anon = login(u("cjw"), e("cwilson@8thlight.com"));
  switch (authenticate(anon, p("admin"))) {
  | Some(Left(usr)) => usr
  | Some(Right(_)) => failwith("Wrong password!")
  | _ => failwith("Chris couldn't log in!")
  };
};

    User.t(regular)
let eve = {
  open User;
  let anon = (login(u("eve"), e("eve@example.com")));
  switch(authenticate(anon, p("regular"))) {
  | Some(Right(usr)) => usr
  | Some(Left(_)) => failwith("not allowed!")
  | _ => failwith("Even couldn't log in")
  }
};

let _ = User.deleteDatabase(chris);
/* let _ = User.deleteDatabase(eve); doesn't compile! */
```

Real world: Provide
feedback to user
on failure



Here we're using this technique. In a real app we'd present user-facing results when someone fails to log in. But the compiler helps us make sure to check this situation.

Note that when we write code where Eve tries to delete the database, the compiler is able to catch this because she's only logged in as a regular user!

Hiding things with polymorphism

```
let f: 'a .(int, int, 'a) => (int, 'a) = (a, b, z) => (a + b, z);

let g: (int, int) => int =
  (i, j) => {
    let (n, m) = f(i, j, 3);
    n + m;
  };
```

Polymorphism of the kind in ML languages can be sort of subtle. Here I'm insisting that the “a” type variable inside “f” is fully polymorphic — it is standing in for an unknown yet specific type. Because of this, we can't make use of it within the body of “f.” We use it like an int because it might not be, similar logic applies for any other type. This essentially “hides a value in plain sight” not because we've made it private, but because we've “forgotten” this information at the type level. Elsewhere in the code, we can “remember” by the way that we use the function.

Hiding things with polymorphism

```
let f: 'a .(int, int, 'a) => (int, 'a) = (a, b, z) => (a + b, z);
```

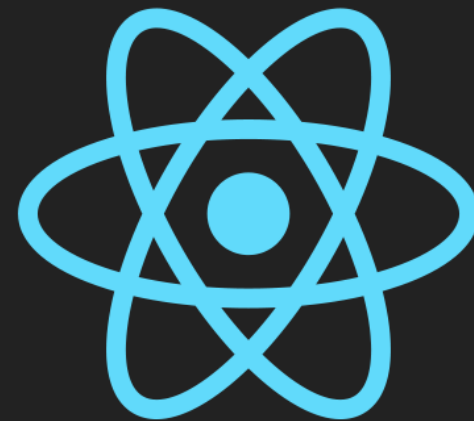
```
let g: (int, int) => int =  
  (i, j) => {  
    let (n, m) = f(i, j, 3);  
    n + m;  
  };
```

z is “hidden” in
the body of *f*. We
can’t do anything
with it!

Forcing ‘a to be
fully polymorphic

But outside of *f*,
we can know the
type & use it

Polymorphism of the kind in ML languages can be sort of subtle. Here I’m insisting that the “a” type variable inside “f” is fully polymorphic — it is standing in for an unknown yet specific type. Because of this, we can’t make use of it within the body of “f.” We use it like an int because it might not be, similar logic applies for any other type. This essentially “hides a value in plain sight” not because we’ve made it private, but because we’ve “forgotten” this information at the type level. Elsewhere in the code, we can “remember” by the way that we use the function.



ReasonReact

TodoList.re

```
module Style = Style.TodoList;

type action =
| Create
| Delete(Todo.t)
| Noop
| UpdatePendingString(string)
| Toggle(Todo.t);

type t = {
  todos: list(Todo.t),
  pendingTodo: string,
};

let component = ReasonReact.reducerComponent("TodoList");
```

styles imported
from central module
(are type checked)

TodoList.re

```
module Style = Style.TODOList;
```

```
type action =  
  | Create  
  | Delete(Todo.t)  
  | Noop  
  | UpdatePendingString(string)  
  | Toggle(Todo.t);
```

action type is an
explicit enum/sum
type

```
type t = {  
  todos: list(Todo.t),  
  pendingTodo: string,  
};
```

t is a convention:
the type of "this module"

```
let component = ReasonReact.reducerComponent("TodoList");
```


Type specifies
reducer vs. stateless

```

let reducer = (action, state) =>
  switch (action) {
  | Noop => None
  | UpdatePendingString(text) => Some({...state, pendingTodo: text})
  | Delete(todo) =>
    let todos = Belt.List.keep(state.todos, item => item != todo);
    Some({...state, todos});
  | Create =>
    switch (String.trim(state.pendingTodo)) {
    | "" => None
    | txt =>
      let todos = state.todos @ [{text: txt, finished: false}];
      Some({pendingTodo: "", todos});
    }
  | Toggle(todo) =>
    let toggleTodo: Todo.t => Todo.t = (
      item => item == todo ? {...item, finished: ! item.finished} : item
    );
    let todos = List.map(toggleTodo, state.todos);
    Some({...state, todos});
  };

```

Extracted “reducer”
function for testing



```
let reducer = (action, state) =>
  switch (action) {
    | Noop => None
    | UpdatePendingString(text) => Some({...state, pendingTodo: text})
    | Delete(todo) =>
      let todos = Belt.List.keep(state.todos, item => item != todo);
      Some({...state, todos});
    | Create =>
      switch (String.trim(state.pendingTodo)) {
        | "" => None
        | txt =>
          let todos = state.todos @ [{text: txt, finished: false}];
          Some({pendingTodo: "", todos});
      }
    | Toggle(todo) =>
      let toggleTodo: Todo.t => Todo.t = (
        item => item == todo ? {...item, finished: ! item.finished} : item
      );
      let todos = List.map(toggleTodo, state.todos);
      Some({...state, todos});
  };
```

```
let make = _children => {  
  ...component,  
  initialState: () => {todos: [], pendingTodo: ""},  
  reducer: (action, state) =>  
    Belt.Option.mapWithDefault(  
      reducer(action, state), ReasonReact.NoUpdate, newState =>  
        ReasonReact.Update(newState)  
    ),  
}
```

```
let make = _children => {  
  ...component,  
  initialState: () => {todos: [], pendingTodo: ""},  
  reducer: (action, state) =>  
    Belt.Option.mapWithDefault(  
      reducer(action, state), ReasonReact.NoUpdate, newState =>  
        ReasonReact.Update(newState)  
    ),  
}
```

**Init component w/
empty todo list**

**Call reducer func.
state updates are
distinct cases**

**New state
to update**

Default

```

render: self => {
  let todoItems = items =>
    List.map(
      (todo: Todo.t) =>
        <Todo
          key=("key-" ++ string_of_int(Random.bits()))
          text=todo.text
          finished=todo.finished
          onDelete=(_mouseEvent => self.send>Delete(todo)))
          onClick=(_mouseEvent => self.send>Toggle(todo)))
        />,
      items,
    );

  let onKeyDown = event =>
    if (ReactEventRe.Keyboard.keyCode(event) == 13) {
      ReactEventRe.Keyboard.preventDefault(event);
      self.send(Create);
    } else {
      self.send>Noop);
    };

  let onChange = event => {
    let str: string = ReactDOMRe.domElementToObj(
      ReactEventRe.Form.target(event),
    )##value;
    self.send(UpdatePendingString(str));
  };

```

```

render: self => {
  let todoItems = items =>
    List.map(
      (todo: Todo.t) =>
        <Todo
          key=("key-" ++ string_of_int(Random.bits()))
          text=todo.text
          finished=todo.finished
          onDelete=(_mouseEvent => self.send(Delete(todo)))
          onClick=(_mouseEvent => self.send(Toggle(todo)))
        />,
      items,
    );

  let onKeyDown = event =>
    if (ReactEventRe.Keyboard.keyCode(event) == 13) {
      ReactEventRe.Keyboard.preventDefault(event);
      self.send(Create);
    } else {
      self.send(Noop);
    };

  let onChange = event => {
    let str: string = ReactDOMRe.domElementToObj(
      ReactEventRe.Form.target(event),
    )##value;
    self.send(UpdatePendingString(str));
  };

```

Create a Todo component for every todo in our state

Pass event handlers w/ binding to reducer actions

Create new todo on enter (otherwise don't)

when text is entered, update the pending todo


```

<div className=Style.body>
  <h1 className=Style.title>
    (ReasonReact.string("TODO LIST (v4.1, Shareware)"))
  </h1>
  (ReactDOMRe.createElement("label", [|ReasonReact.string("Todo")|]))
  <input
    className=Style.input
    name="input"
    placeholder="Add new TODO"
    value=self.state.pendingTodo
    onKeyDown
    onChange
  />
  <h2 className=Style.listHeading> (ReasonReact.string("Items")) </h2>
  <ul className=Style.list>
    (
      switch (self.state.todos) {
      | [] => <i> ("Nothing to do" |> ReasonReact.string) </i>
      | todos =>
        todoItems(todos) |> Belt.List.toArray |> ReasonReact.array
      }
    )
  </ul>
  <hr />
  <small className=Style.badgerText>
    (ReasonReact.string("Upgrade to TODO LIST PRO today!!!"))
  </small>
</div>;}};

```

This hooks up the event handlers that we've seen previously & lays out the todo list. This is the end of the TodoList component.

Todo.re

```
type t = { text: string finished: bool };
let component = ReasonReact.statelessComponent("Todo");
let make = (~text, ~finished, ~onDelete, ~onClick, _children) => {
  ...component,
  render: _self => {
    let key = ("id-" ++ string_of_int(Random.bits()));

    <li className=Style.item>
      <input
        id=key
        className=Style.checkbox
        type_="checkbox"
        onChange=onClick
        checked=finished
      />
      <label htmlFor=key className=(finished ? Style.finished : Style.unfinished)>
        <span className=(finished ? Style.boxFinished : Style.boxUnfinished)>
          (ReasonReact.string(finished ? {j|✓|j} : {j|✖|j}))
        </span>
        (ReasonReact.string(text))
      </label>
      <span>
        <button className=Style.delete onClick=(evt => onDelete(evt))>
          (ReasonReact.string({j| x |j}))
        </button>
      </span>
    </li>}};
```

Todo.re

```
type t = { text: string finished: bool };
let component = ReasonReact.statelessComponent("Todo");
let make = (~text, ~finished, ~onDelete, ~onClick, _children) => {
  ...component,
  render: _self => {
    let key = ("id-" ++ string_of_int(Random.bits()));

    <li className=Style.item>
      <input
        id=key
        className=Style.checkbox
        type_="checkbox"
        onChange=onClick
        checked=finished
      />
      <label htmlFor=key className=(finished ? Style.finished : Style.unfinished)>
        <span className=(finished ? Style.boxFinished : Style.boxUnfinished)>
          (ReasonReact.string(finished ? {j|✓|j} : {j|✖|j}))
        </span>
        (ReasonReact.string(text))
      </label>
      <span>
        <button className=Style.delete onClick=(evt => onDelete(evt))>
          (ReasonReact.string({j| x |j}))
        </button>
      </span>
    </li>}};
```

stateless/reducer
explicit in type

props become
normal Reason
named args

Quoted string (with
unicode & interpolation)

Show the app

All your ReactJS knowledge, codified.

- I think the slogan fits — mostly.
- JSX supported via normal Reason syntax extension system (a similar desugaring as JSX in JavaScript)

```
<div foo={bar}> {child 1} {child2} </div>
```

becomes:

```
ReactDOMRe.createElement("div",  
  ~props=ReactDOMRe.props(~foo=bar, ()),  
  [|child1,child2|]);
```

All your ReactJS knowledge, codified.

- Lifecycle events are typed — I think this improves usability & prevents some trouble spots
- Can always use the escape hatch of implementing a component in JS.

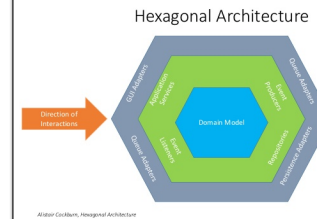
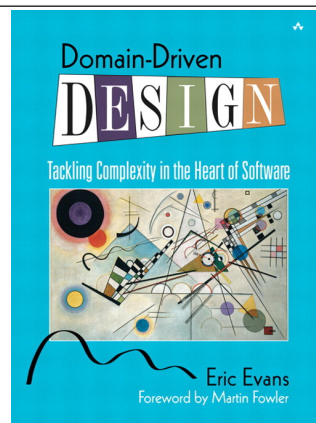
...a strong fit...

Now I'm going to call back the the claim that I made earlier in the talk.

- Polymorphism
- Encapsulation
- Well-defined interface
- Explicit dependencies
- Separate compilation (incremental builds)
- Tooling (IDE, editors, linters, formatters, etc.)
- Learning curve
- Library ecosystem
- Syntactic extension (macros, quasiquoters, etc.)
- Automatic memory management (garbage collection)

This is a big list of features, but I think there's something good that happens when these all blend into a coherent whole. I think that every feature here is either a desirable characteristic or is already widespread. In fact, for some of these, languages that popularized that bullet themselves became very popular.

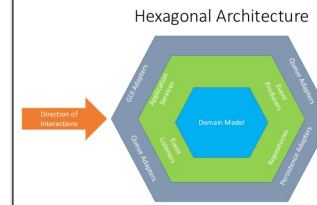
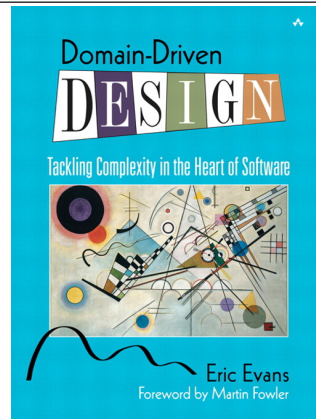
For business applications, these features are incredibly useful.



There are many currents within software development that seem to point to something. Defining interfaces, separating concerns, layering architecture, bounded context, ordering dependencies.

They all seem to point to some technology...

Do we just really want modules?

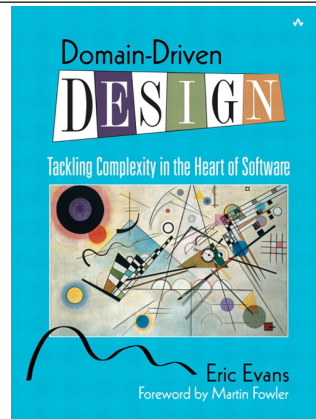


M

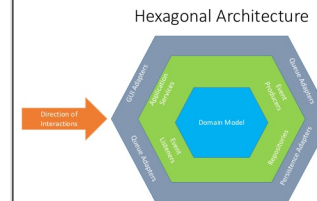
There are many currents within software development that seem to point to something. Defining interfaces, separating concerns, layering architecture, bounded context, ordering dependencies.

They all seem to point to some technology...

Do we just really want modules?



Microservices



Modules?

There are many currents within software development that seem to point to something. Defining interfaces, separating concerns, layering architecture, bounded context, ordering dependencies.

They all seem to point to some technology...

Do we just really want modules?

Modules

- “First class files”
- Not 1:1 with files: multiple modules can be in a single file
- Can be assigned to variables
- Optional, explicit interface (allows abstract types, & “private” functions)
- Can be ***opened*** (brought into scope) or ***included*** (mixed-in)
- Module functions, ***functors***, (functions that return a module)

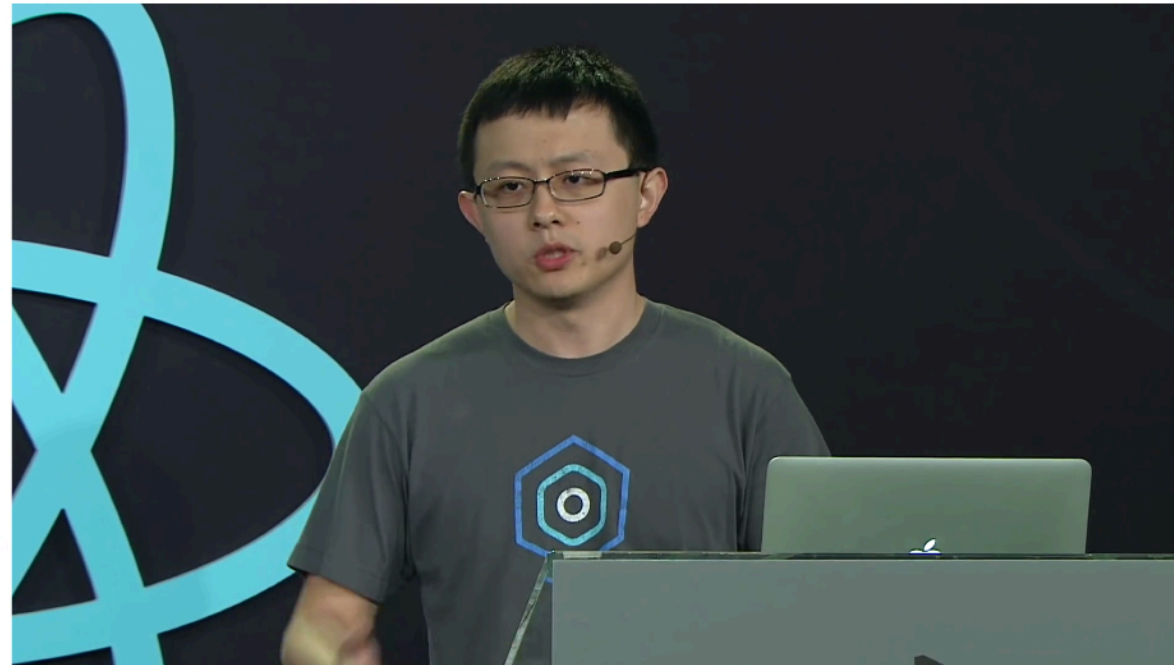
Modules are the mechanism that allow us to depend on interfaces with strong guarantees of encapsulation. They let us talk about groups of types and values as a unit. Depending on the language they are similar to packages, classes, or assemblies.

Modules v. Microservices

- | | |
|--|--|
| <ul style="list-style-type: none">• Microservice | <ul style="list-style-type: none">• Module |
| <ul style="list-style-type: none">• Strong encapsulation | <ul style="list-style-type: none">• Strong encapsulation |
| <ul style="list-style-type: none">• Well-defined interface | <ul style="list-style-type: none">• Well-defined interface |
| <ul style="list-style-type: none">• Explicit dependencies | <ul style="list-style-type: none">• Explicit dependencies |
| <ul style="list-style-type: none">• Independent deployment | <ul style="list-style-type: none">• Separate compilation |
| <ul style="list-style-type: none">• Team dynamics(?) | |

There are obviously differences, but I also think there's a lot of overlap. Modules are also nice in that they don't bring a network into the application — with all that extra complexity.

Taming the meta language



Cheng Lou - Taming the Meta Language - React Conf 2017

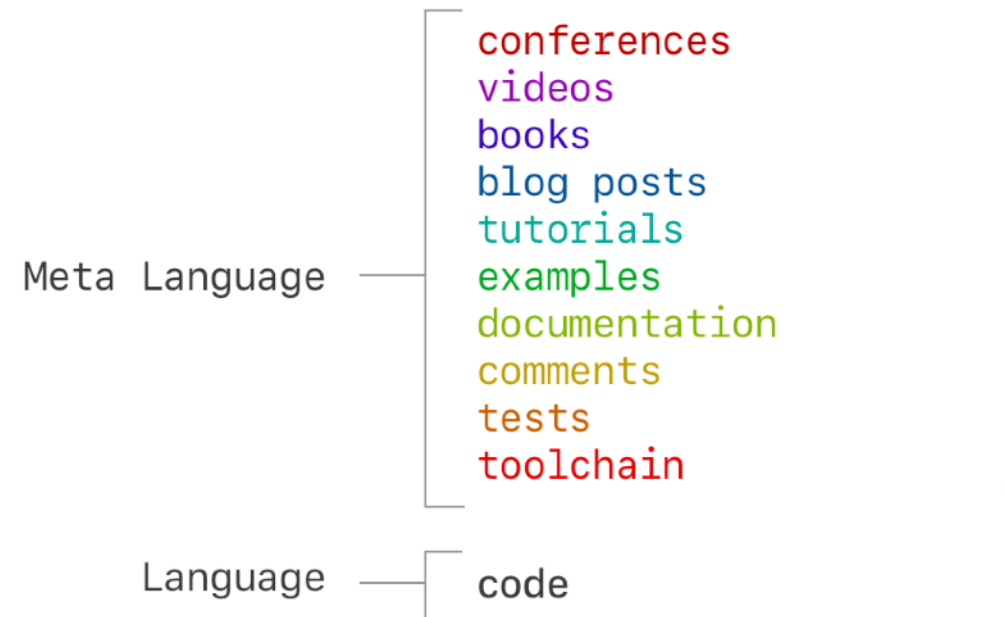
[next slide]

Taming the meta language

- A *Hickeyian* sort of talk: it's about more than what it's about
- Language is just a part of any real project
- Bring more and more proximate concerns *into* the code

Go watch the talk at some point. I think it applies beyond just Reason.

Taming the meta language



I took this slide from the talk. Bringing things down into the language that we would normally think of as separate gives us a lot of power. These things become codified, and more accessible. It also improves our ability to talk about the kinds of software we're creating.

Thanks!

- Chris Wilson <cwilson@8thlight.com>

References

- *Reason* <https://reasonml.github.io/>
- *ReasonReact* <https://reasonml.github.io/reason-react/>
- *Modules vs Microservices* <https://www.oreilly.com/ideas/modules-vs-microservices>
- *Domain-driven design* <http://fsharpforfunandprofit.com/ddd>
- *How Elm Slays a UI Antipattern* <http://blog.jenkster.com/2016/06/how-elm-slays-a-ui-antipattern.html>
- *Taming the Meta Language* https://youtu.be/_0T5OSSzxms
- *Understanding “Taming the Meta Language”* <http://frantic.im/meta-language>