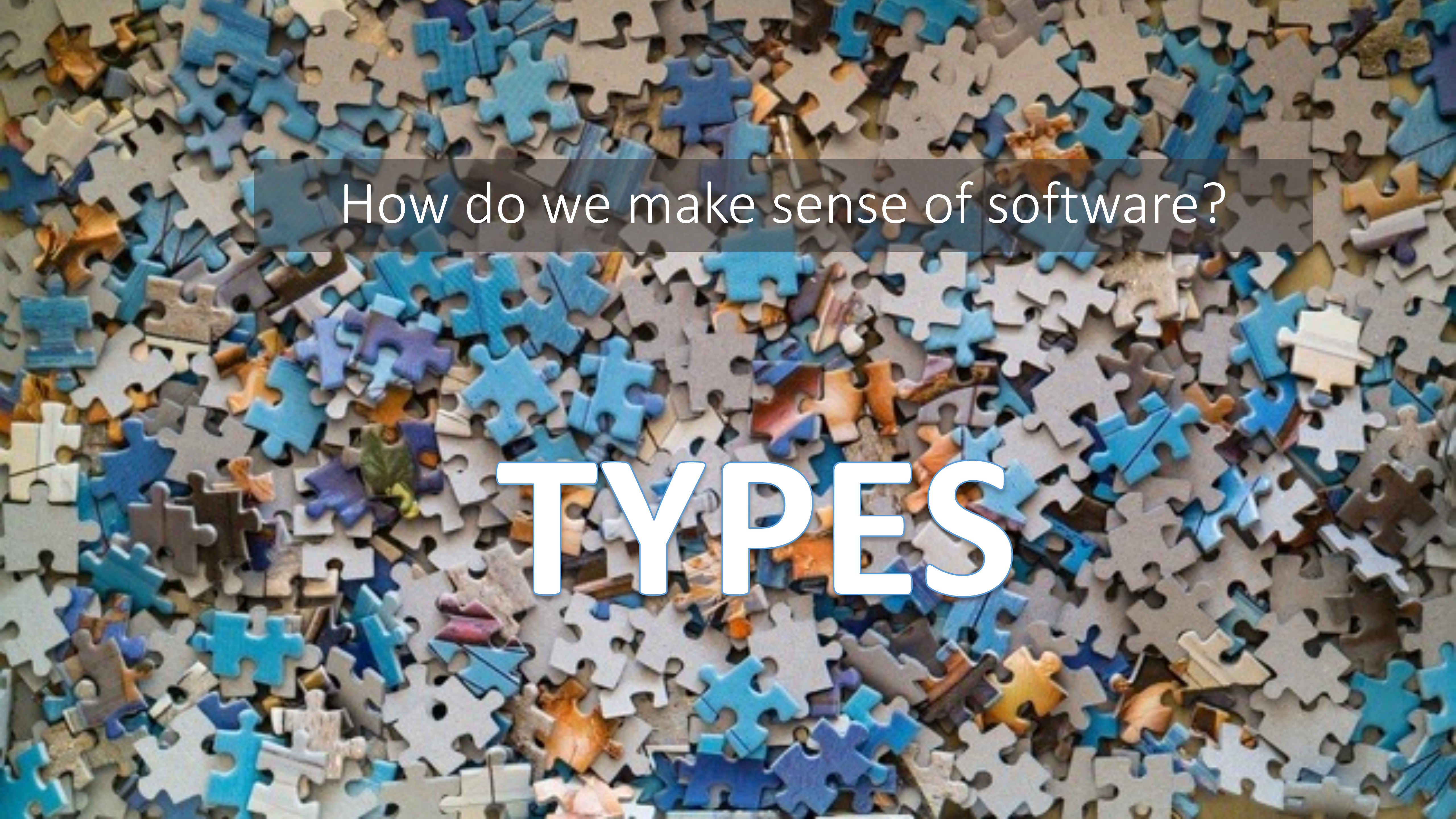
What are dependent types and what are they good for?

Stephanie Weirich

ENIAC President's Distinguished Professor of
Computer and Information Science
University of Pennsylvania



How do we make sense of software?

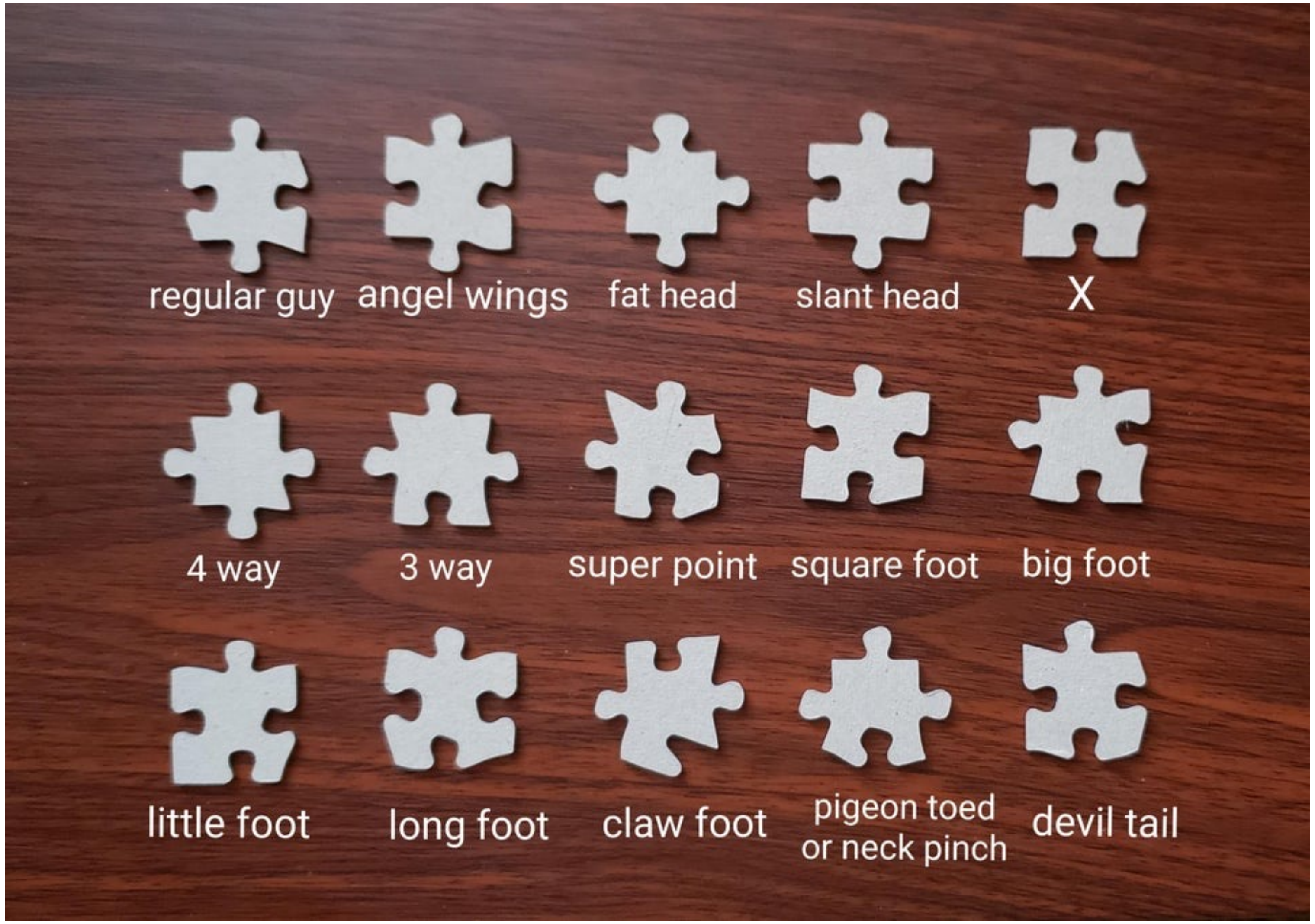
TYPES

Static typing is by far the most widely used program verification technology in use today

Principle: Static type systems help us create and maintain reliable software, at scale

Types let us make distinctions between things that look similar

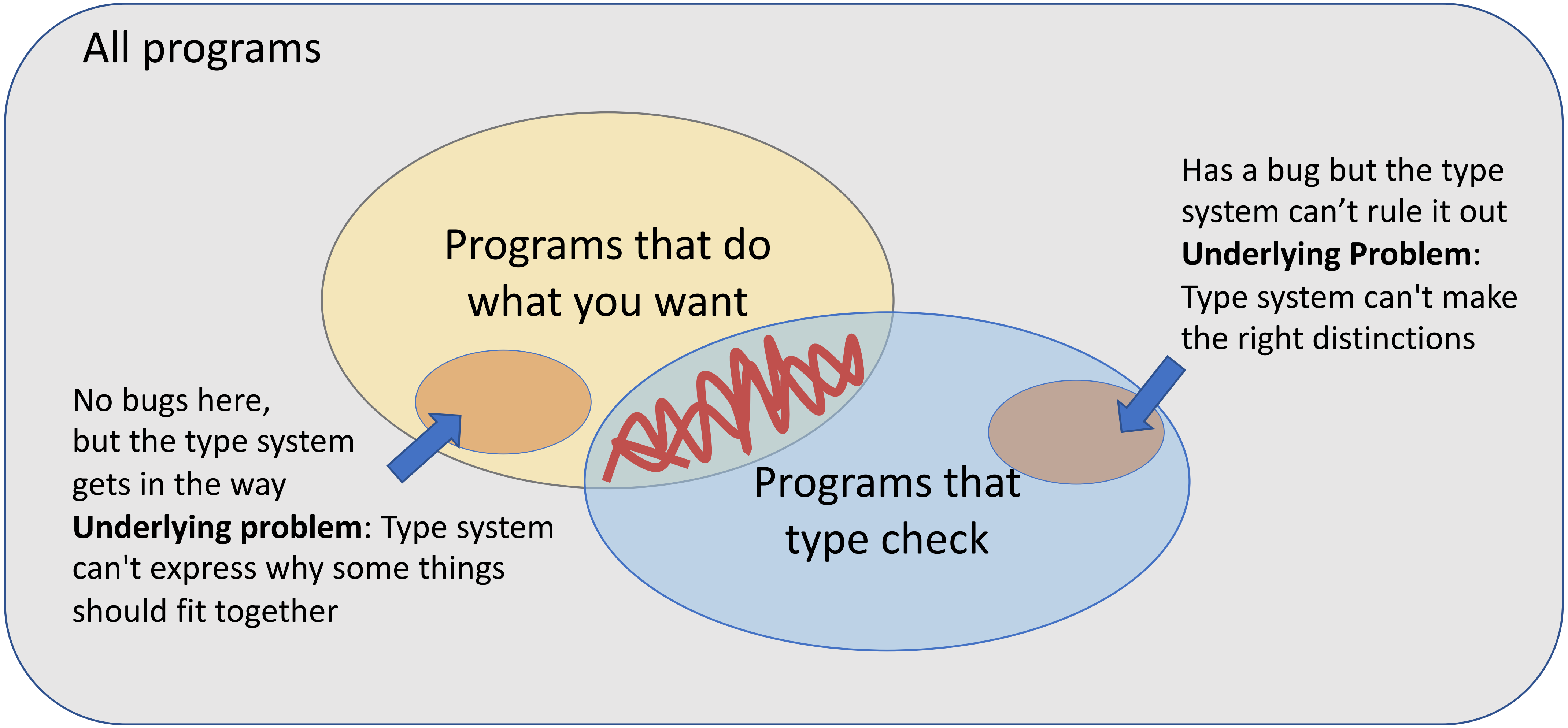
Types tell us how things should fit together



Improving these leads to better type systems

u/PeranerdErndrerd

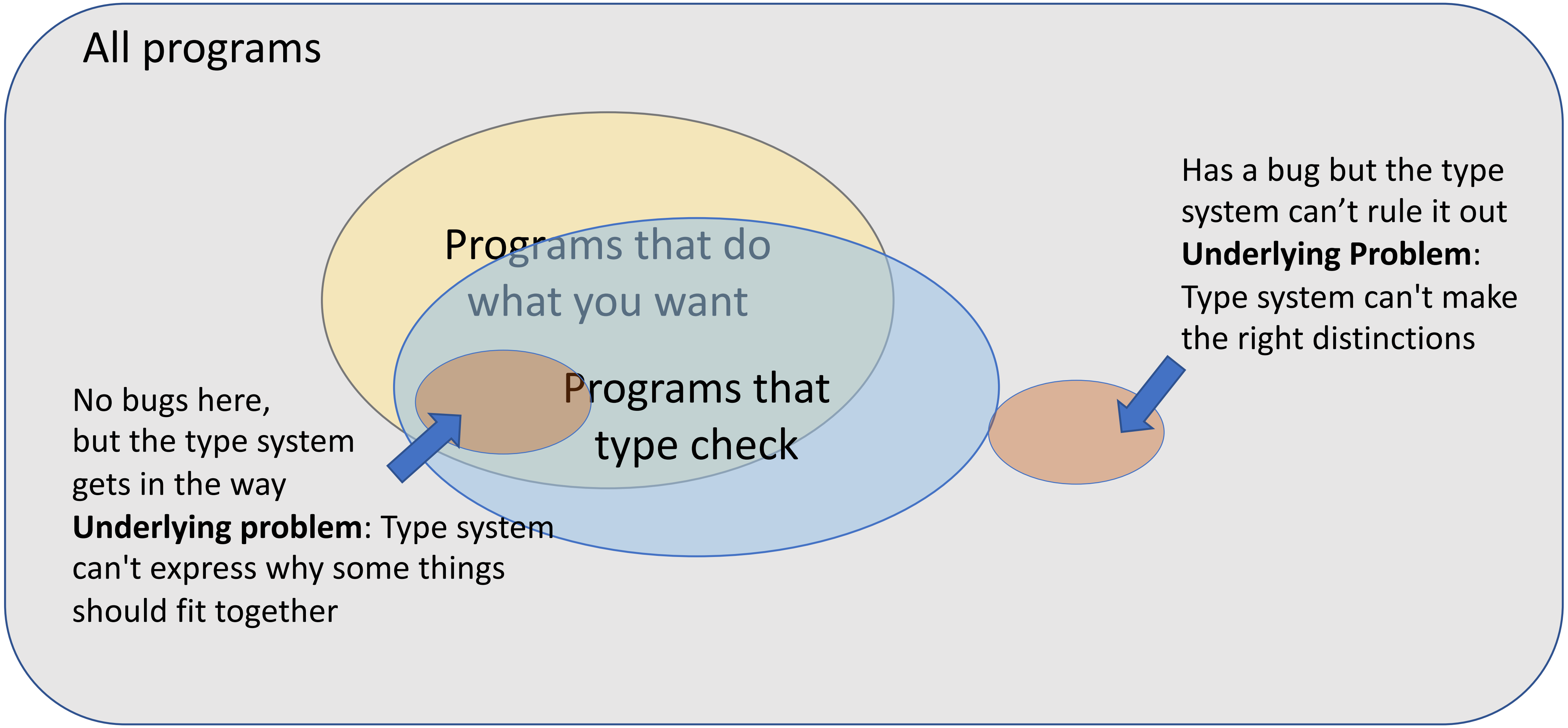
How to Design Type Systems



No bugs here,
but the type system
gets in the way
Underlying problem: Type system can't express why some things should fit together

Has a bug but the type system can’t rule it out
Underlying Problem:
Type system can't make the right distinctions

How to Design Type Systems



No bugs here,
but the type system
gets in the way
Underlying problem: Type system can't express why some things should fit together

Has a bug but the type system can't rule it out
Underlying Problem:
Type system can't make the right distinctions

Robin Milner's Types

A tool for practical programming

A Theory of Type Polymorphism in Programming

ROBIN MILNER

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978

The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types, entails defining procedures which work well on objects of a wide variety. We present a formal type discipline for such polymorphic procedures in the context of a simple programming language, and a compile time type-checking algorithm $\mathscr{W}$ which enforces the discipline. A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot "go wrong" and a Syntactic Soundness Theorem states that if $\mathscr{W}$ accepts a program then it is well typed. We also discuss extending these

Milner's Contributions

Semantic soundness theorem
"Well-typed programs cannot go wrong"

Syntactic soundness theorem
Algorithm W that decides when programs are well-typed

Semantic Soundness Theorem: "Well-typed Programs cannot go wrong"

- Semantics includes a value "wrong" that corresponds to run-time failure
  - EXAMPLE: using a function value as a conditional
- Type system assigns types to some expressions and semantic values, but "wrong" does not have a type
- **Semantic Soundness Theorem**: the semantic value of a well-typed expression is well-typed (with the same type)
- Corollary: If an expression type checks, it **cannot** evaluate to "wrong"

Syntactic Soundness Theorem: Algorithm W

- Key feature of Milner's language is type polymorphism
  ```
  map : ∀ α β. (α -> β) -> α list -> β list
  ```
- Algorithm W calculates principal types (every function has a "best" or "most polymorphic" type)
- Complete type inference: no type annotations required!
- **Syntactic soundness theorem**: Algorithm W correctly determines whether programs are well-typed
- Foundation for ML and Haskell languages, used to this day

Can Dependent Types improve on Milner's type system?

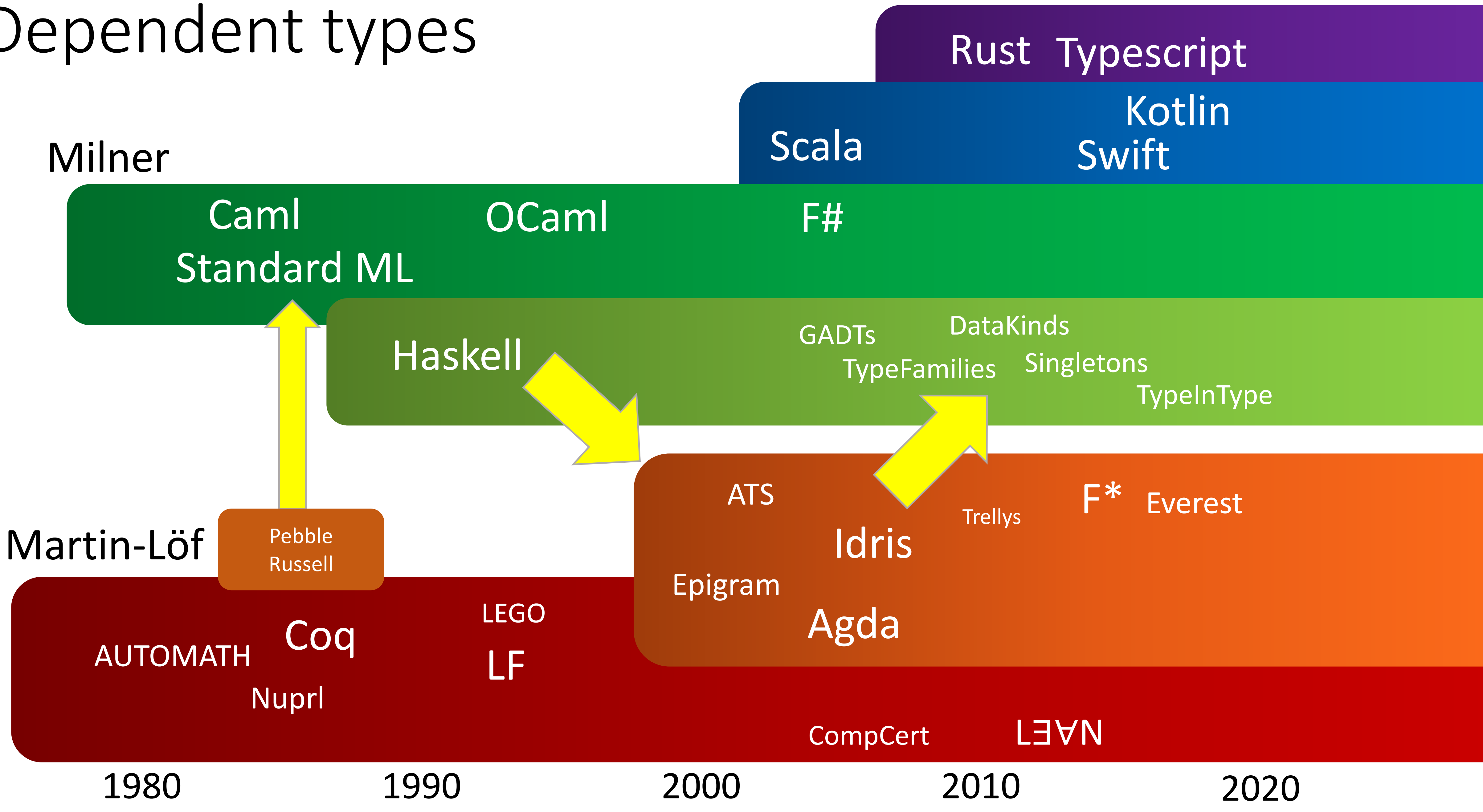
Semantic soundness: Well-typed programs cannot "go wrong"

- Type system approximates "buggy program" by "going wrong"
- "Buggy" defined by the programming language semantics
- What if: buggy could be defined by the program itself?
- *Dependent type systems enable more distinctions*

Syntactic soundness: Algorithm W

- Types are properties of values, and there is a "best" type
- What if: types could express relationships between values?
- *Dependent type systems enable more programs to type check, but less automatically*

Dependent types



What are dependent types good for in practical programming?

"Dependent" Haskell

GADTs DataKinds TypeFamilies Singletons TypeInType

ATS Idris Trellys F* Epigram Agda

Application-specific **distinctions** between values

Access control

```haskell
data User = Admin | Normal
data Account = Account { userType :: User ... }

-- run code with any user account
doUserStuff :: Account -> IO ()
-- only admin accounts allowed
doAdminStuff :: Account -> IO ()
-- make sure this runs *without* admin privileges
doPublicStuff :: Account -> IO ()
```

Access conrol

```
data User = Admin | Normal
data Account (u :: User) = Account { userType :: … }

-- run code with any user account
doUserStuff :: Account u -> IO ()
-- only admin accounts allowed
doAdminStuff :: Account Admin -> IO ()
-- make sure this runs *without* admin privileges
doPublicStuff :: Account Normal -> IO ()
```

Red-Black Trees

```
data Tree :: Type where
  E  :: Tree
  NR :: Tree -> A -> Tree -> Tree    -- red node
  NB :: Tree -> A -> Tree -> Tree    -- black node

data RBT :: Type where
  Root :: Tree -> RBT

insert :: RBT -> A -> RBT
insert (Root t) x = …
```

1. Red nodes have black children
2. Empty trees are black
3. Root is black
4. Both children of a node have same black height

Red-Black Trees

```
data Tree :: Color -> Type where
  E  :: Tree Black
  NR :: Tree Black -> A -> Tree Black -> Tree Red
  NB :: Tree c1 -> A -> Tree c2 -> Tree Black

data RBT :: Type where
  Root :: Tree Black -> RBT

insert :: RBT -> A -> RBT
insert (Root t) x = …
```

1. Red nodes have black children
2. Empty trees are black
3. Root is black
4. Both children of a node have same black height

Red-Black Trees

```
data Tree :: Color -> Nat -> Type where
  E  :: Tree Black Zero
  NR :: Tree Black n -> A -> Tree Black n -> Tree Red n
  NB :: Tree c1 n -> A -> Tree c2 n -> Tree Black (Suc n)

data RBT :: Type where
  Root :: Tree Black n -> RBT

insert :: RBT -> A -> RBT
insert (Root t) x = …
```

1. Red nodes have black children
2. Empty trees are black
3. Root is black
4. Both children of a node have same black height

More Examples (distinctions)

- Verified data structures
  (Merkle Trees, Braun Trees, etc)
- Units of measure (2 inches vs. 2cm)
- Data provenance (tracking unsanitized inputs)
- Refinement types (positive integers, nonempty lists)
- Well-scoped and strongly-typed ASTs

Application-specific **relationships** between values

Data Processing in Python

```
def read_db(classes_schema, students_schema):
     classes_data = load_table(classes_schema, 'classes.json')
     student_data = load_table(students_schema, 'students.json')
     name = raw_input("Whose students do you want to see:")
     students = [ student for class in classes_data  \
                          for student in student_data \
                          if (class['prof'] == name) \
                          and (student['id'] in class['students'])]
     map(lambda row: print(row['name']), students)
```

Data Processing in Haskell

```haskell
read_db classes_schema students_schema = do
     classes_data <- load_table classes_schema "classes.json"
     student_data <- load_table students_schema "students.json"
     name <- putStr "Whose students do you want to see?" >> getLine
     let students = [ student | class <- classes_data
                              , student <- student_data
                              , class.prof == name
                              , student.id `elem` class.students ]
     mapM_ (\row -> print row.name) students
```

```
read_db :: (HasField "students" (Row schema1) [Int],
            HasField "prof"     (Row schema1) String,
            HasField "id"       (Row schema2) Int,
            HasField "name"     (Row schema2) String) =>
        Schema schema1 -> Schema schema2 -> IO ()
read_db classes_schema students_schema = do
     classes_data <- load_table classes_schema "classes.json"
     student_data <- load_table students_schema "students.json"
     name <- putStr "Whose students do you want to see?" >> getLine
     let students = [ student | class <- classes_data
                               , student <- student_data
                               , class.prof == name
                               , student.id `elem` class.students ]
     mapM_ (\row -> print row.name) students
```

Types **relate** schema to data

```
load_table :: Schema schema -> FilePath -> IO [Row schema]

-- Schema type describes expected data format
classes_schema :: Schema [ Col "prof" String
                         , Col "students" [Int]
                         , Col "course_id" String ]
-- Ensures data is a list of rows in the specified format
class_data = [ "Weirich" :> [1,5,7] :> "CIS 552" :> Nil
             , "Zdancewic" :> [1,2] :> "CIS 341" :> Nil ]
```

Types describe **interfaces**

Informal specification of Web API

The endpoint at /users expects a GET request and returns a list of JSON objects describing users, with fields name, email, and registration_date,

or a single user specified by an id number

```
data User = User { name :: String,
                   email :: String,
                   registration_date :: Day }

type UserAPI =
   "users" :> Get '[JSON] [User]  :<|>
   "users" :> Capture "id" Int :> Get '[JSON] User
```

Haskell Servant Library

- API type expresses relationship between the server and client applications
- Server: Ensures web application provides exactly this functionality

```
server :: Server UserAPI
server = sendUserById :<|> return users

sendUserById id = if id < length users then return (users !! id)
                  else throwError err404
```

- Client: Receives values of the associated type

```
getUserById :<|> getUsers = client userAPI
```

Automatic marshalling

Server and client need to convert between type `User` and JSON formatted strings.

```
data User = User { name :: String,
                   email :: String,
                   registration_date :: Day }
   deriving (Eq, Show, Generic)

instance ToJSON User       -- uses Aeson library to derive
instance FromJSON User     -- based on Generic rep of type
```

Dependent Types track relationship between `User` and its generic representation.

More examples (relationships)

- Other rich interfaces (printf, regexp matching)
- Type-based reflection (Dynamic types, type-indexed maps, datatype-generic programming)
- Protocols, i.e. interfaces with effects (state machines, session types, algebraic effects)
- Improved performance (Express low-level memory layouts, eliminate dynamic array bounds and tag checks, verify C code)
- Embedded Domain-Specific Languages (Hardware/FPGAs, Music, Contract Law, etc.)

The **Future** of Dependent Types in Practical Programming

- Distinctions for describing resource usage: integration of linear and dependent types (already in Agda, Idris)
- New approaches to type and proof inference: SMT solvers (LiquidHaskell), tactic languages (Mtac2), algorithms (QuickLook)
- More type-directed program development: the type system as a guide either in the IDE or for automatic synthesis
- More flexibility in working with coercions and isomorphisms: new semantics for dependent types (HoTT, Higher-Inductive Types)

Summary: Static types provide practical benefits to programmers.

Dependent types provide practical benefits to type systems.

- Making application-specific **distinctions** between values
- Expressing application-specific **relationships** between values