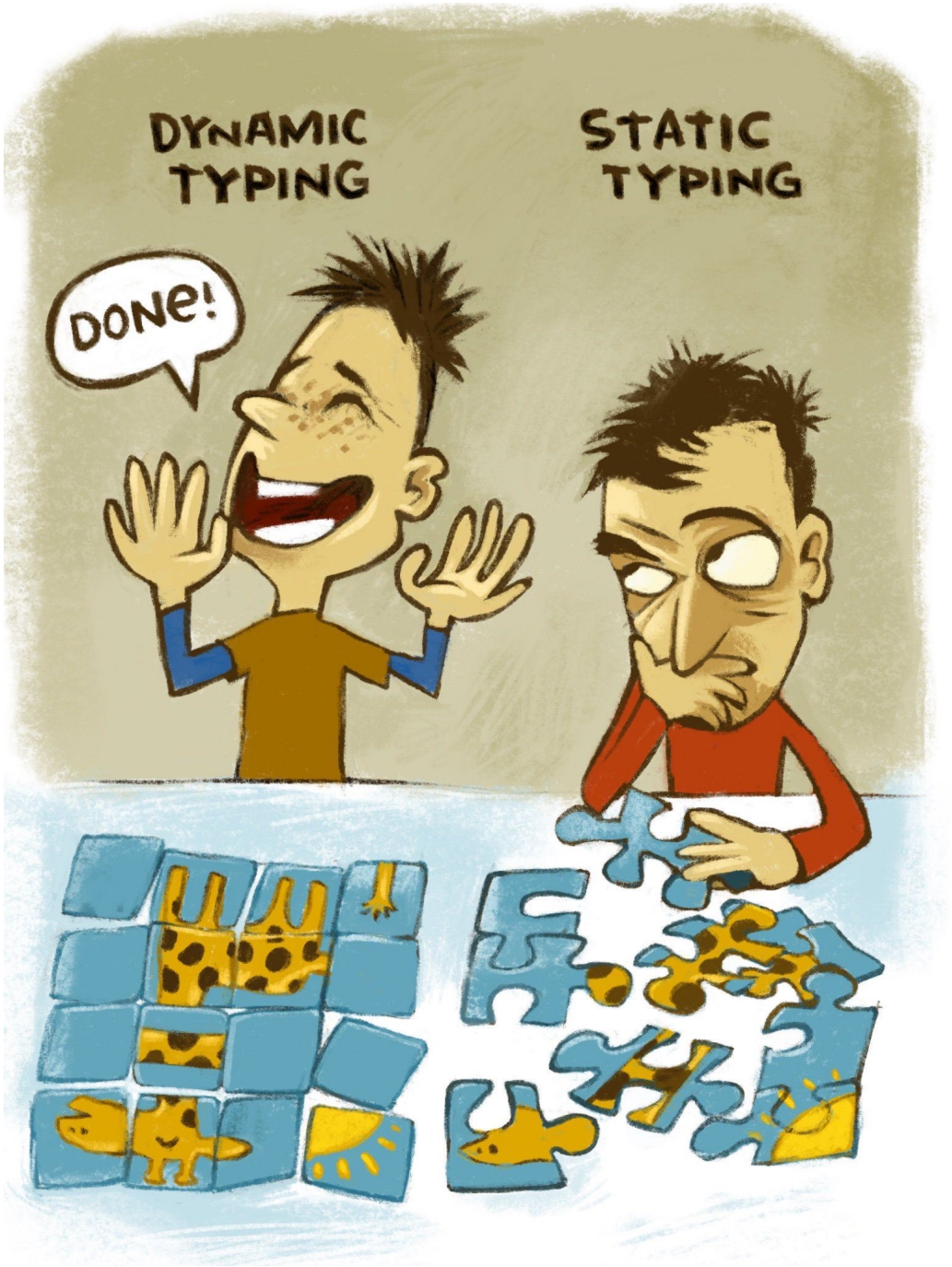
The power of Π

λ Days 2020 in Krakow

Thorsten Altenkirch

Functional Programming Laboratory
School of Computer Science
University of Nottingham

February 14, 2020


DYNAMIC TYPING
STATIC TYPING
DONE!

Typing disciplines

C	static but weak
Java, C#	static / dynamic but strong
Python, Scheme	dynamic and strong
Haskell, SML	static and strong

Haskell's type system
is not up to the job!

Because it lacks dependent types.

What are dependent types?

Conor McBride: Winging It, 2000?

The established social order

Proletarian revolution: do the types run away in terror?

No! They're overjoyed at their new articulacy.

Π-types

The title of this talk was stolen from a ICFP 08 paper by Nicolas Oury and Wouter Swierstra

$$\begin{aligned}
&\mathsf{zeroes} : (\mathsf{n} : \mathbb{N}) \to \mathsf{Vec}\ \mathbb{N}\ \mathsf{n} \\
&\mathsf{zeroes\ zero} = [] \\
&\mathsf{zeroes\ (suc\ n)} = 0 :: \mathsf{zeroes\ n}
\end{aligned}$$

Dependent function type, also called Π-type.

$$\Pi_{\mathsf{n} : \mathbb{N}} \mathsf{Vec\ A\ n}$$

Why is it called Π-type?

$$\text{Fin n} = \{0, 1, .. \text{ n-1}\}$$
$$\text{f} : \text{Fin 3} \to \mathbb{N}$$
$$\text{f 0} = 2$$
$$\text{f 1} = 3$$
$$\text{f 2} = 4$$

$$(\text{x} : \text{Fin 3}) \to \text{Fin (f x)} = \Pi_{i=0}^{i<3} f\ i = 24$$

The same idea works for dependent pairs = Σ-types.

What is the power of Π?

Good programs that can't be typed without Π

$$
\begin{aligned}
&\mathsf{printf} : (\mathsf{s} : \mathsf{String}) \to \mathsf{Args}\ \mathsf{s} \\
&\mathsf{Args} : \mathsf{String} \to \mathsf{Set} \\
&\mathsf{Args}\ \texttt{"\%s"} :: \mathsf{s} = \mathsf{String} \to \mathsf{Args}\ \mathsf{s} \\
&\mathsf{Args}\ \texttt{"\%d"} :: \mathsf{s} = \mathsf{Int} \to \mathsf{Args}\ \mathsf{s} \\
&\mathsf{Args}\ \mathsf{c} :: \mathsf{s} = \mathsf{Args}\ \mathsf{s} \\
&\mathsf{Args}\ \texttt{""} = \mathsf{IO}\ \top
\end{aligned}
$$

Pre-Newtonian dependent types

Objection

Using data kinds we can represent types like this in Haskell.

- Yes, but only if the format string is static.
- What if it is part of a scheme that is computed internally?

The scheme is part of the data

```
Scheme : Set
Data : Scheme → Set
record DataBase : Set where
  field
    scheme : Scheme
    content : Data scheme
```

The power of Σ.

Eliminating runtime errors

$$\mathsf{Matr} : \mathbb{N} \to \mathbb{N} \to \mathsf{Set}$$
$$\mathsf{Matr}\ \mathsf{m}\ \mathsf{n} = \mathsf{Vec}\ \mathsf{m}\ (\mathsf{Vec}\ \mathsf{n}\ \mathbb{R})$$
$$\_\times\_ : \{\mathsf{i}\ \mathsf{j}\ \mathsf{k} : \mathbb{N}\} \to \mathsf{Matr}\ \mathsf{i}\ \mathsf{j} \to \mathsf{Matr}\ \mathsf{j}\ \mathsf{k} \to \mathsf{Matr}\ \mathsf{i}\ \mathsf{k}$$

Gradual Typing

```
U : Set
El : U → Set
record Dynamic where
  field
    type : U
    value : El type
```

Summary : use cases for dependent types

- Giving types to all sensible programs
- Schemes are part of the data
- Eliminating run time errors
- Capturing gradual typing

If dependent types are so great
why isn't everybody using them?

Some myths and facts about dependently typed programming (DTP)

Myth 1 : Programming in DTP languages is more difficult

- DTP languages contain Haskell types as a subset.
- Hence you can always not use dependent types.
- If you use dependent types you may have to work harder to get your programs through the type checker.
- Pay as you go.

Myth 2 : You have to prove that your program is terminating

- Good programs are total programs.
- It can be hard to convince a compiler that your program is total.
- You can declare that your program is total without proving it.
  I am a doctor.
- In some cases knowing that your program is total can make the program faster without compromising safety.

Knowing that your program terminates makes it faster

```
have : Vec A m
? : Vec A n
thm : m ≡ n
transport (Vec A) thm have : Vec A n
```

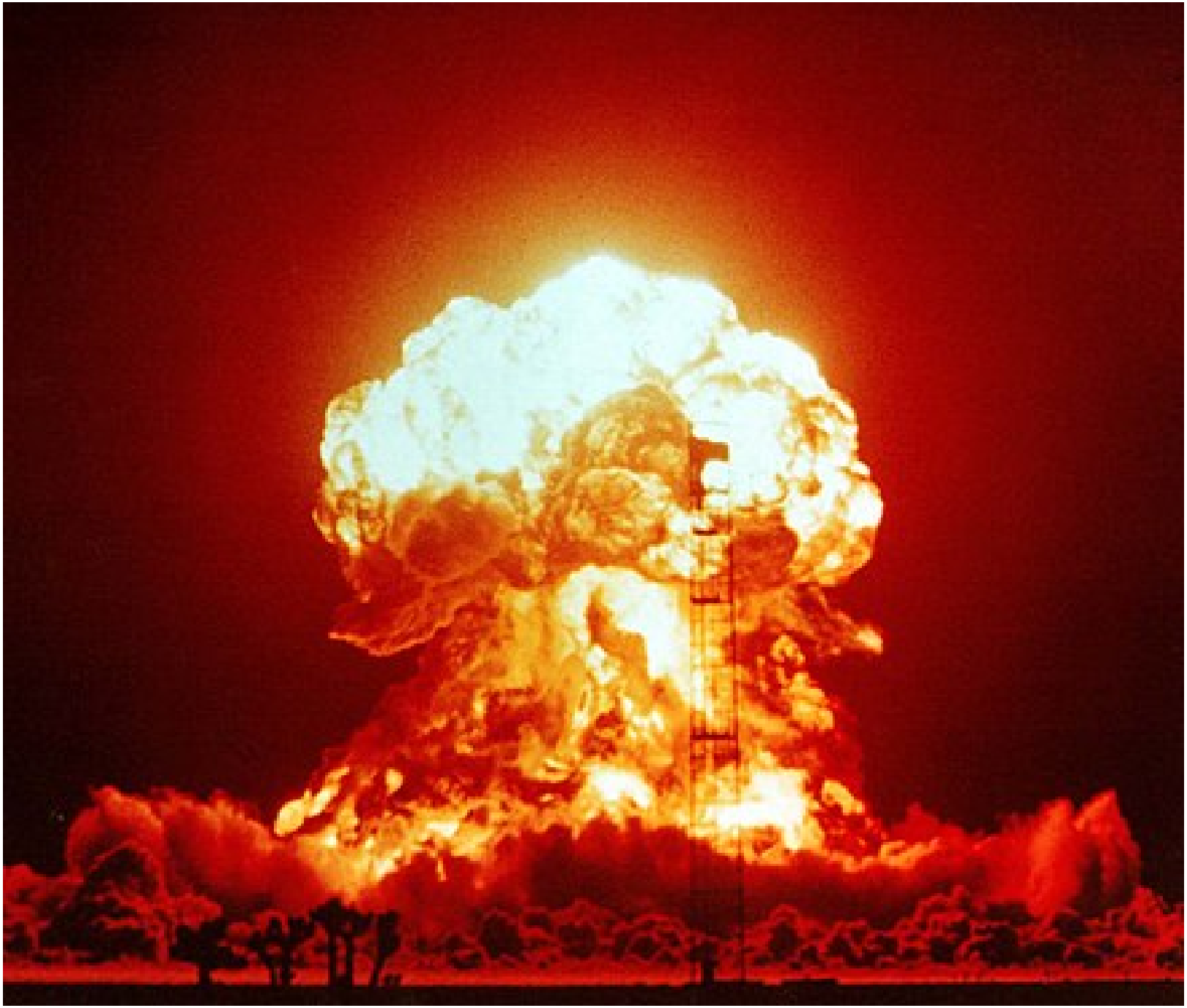
- Do we need to run thm at runtime?
- No, if we know that it is total.
- Yes, if we don't.

Myth 3 : DTP languages are not Turing complete

- If we insist that all programs are total we can't have all programs.
- Partiality is a monad (see our paper).
  TA, Nils Anders Danielsson, Nicolai Kraus
  Partiality, Revisited: The Partiality Monad as a Quotient Inductive-Inductive Type
- Using the partiality monad we can write and run all partially recursive functions in a DTP language.

Myth 4 : DTP doesn't work with effects

- What happens if an effectful computation appears in a type?
- Does the compiler execute the effect?
- E.g. we implement a missile control system and startMissiles : IO $\top$ appears in a dependent type ...



Beauty in the Beast

- Wouter Swierstra and I made a proposal how to integrate effects in DTP in 2007.
- At compiletime we use a mathematical semantics of effects, a functional program.
- At runtime we just execute the effect.
- Example: Partiality

Fact 1: DTP makes big demands on the IDE

- Need special support to effectively develop DTP software.
- Interactive and incremental typing
- Automatic case splitting
- Moving between implicit and explicit syntax

Fact 2: Compile time can be an issue

- The problem is not that you run functions at compile time.
- Solving unification problems with existential variables may blow up compilation time.
- Cause: loss of sharing
- Solutions have been suggested (e.g. by Andras Kovacs) but still an issue in current implementations.

Fact 3: More precise types can hamper reusability

- Eg we have lists, vectors, sorted lists, ...
- In many cases this can be addressed by identifying new abstractions.
- Cf Conor McBride's paper:
  Ornamental Algebras, Algebraic Ornaments

Fact 4: DTP toolchains are not yet ready for professional developers

This guy may disagree:

Edwin Brady, developer of Idris
and author of *Type driven development with Idris*

Summary: Myths vs facts

Myths

1. Programming in DTP languages is more difficult
2. You have to prove that your program is terminating
3. DTP languages are not Turing complete
4. DTP doesn't work with effects

Facts

1. DTP makes big demands on the IDE
2. Compile time can be an issue
3. More precise types can hamper reusability
4. DTP toolchains are not yet ready for professional developers

Claims

- A strong static type discipline requires dependent types.
- Dependent types are not an additional feature to be added in the end.
- Full dependent types are feasible and extremely useful for programming.