Subtyping and Type Inference

Software Technology Group, Utrecht University, March 2024

David Binder, University of Tübingen, 2024

Happy to be back!

Dependent Co/Data Types

Dependent Co/Data Types

- Symmetric dependent data and codata types.

Dependent Co/Data Types

- Symmetric dependent data and codata types.
- De/-Refunctionalize all the types!

Dependent Co/Data Types

- Symmetric dependent data and codata types.
- De/-Refunctionalize all the types!
- Accepted at OOPSLA '24

Dependent Co/Data Types

- Symmetric dependent data and codata types.
- De/-Refunctionalize all the types!
- Accepted at OOPSLA '24
- We are currently preparing the camera-ready version.

Dependent Co/Data Types

- Symmetric dependent data and codata types.
- De/-Refunctionalize all the types!
- Accepted at OOPSLA '24
- We are currently preparing the camera-ready version.

Deriving Dependently-Typed OOP from First Principles

DAVID BINDER, University of Tübingen, Germany
INGO SKUPIN, University of Tübingen, Germany
TIM SÜBERKRÜB, Aleph Alpha, Germany
KLAUS OSTERMANN, University of Tübingen, Germany

The *expression problem* describes how most types can easily be extended with new ways to *produce* the type or new ways to *consume* the type, but not both. When abstract syntax trees are defined as an algebraic data type, for example, they can easily be extended with new consumers, such as *print* or *eval*, but adding a new constructor requires the modification of all existing pattern matches. The expression problem is one way to elucidate the difference between functional or data-oriented programs (easily extendable by new consumers) and object-oriented programs (easily extendable by new producers). This difference between programs which are extensible by new producers or new consumers also exists for dependently typed programming, but with one core difference: Dependently-typed programming almost exclusively follows the functional programming model and not the object-oriented model, which leaves an interesting space in the programming language landscape unexplored. In this paper, we explore the field of dependently-typed object-oriented programming by *deriving it from first principles* using the principle of duality. That is, we do not extend an existing object-oriented formalism with dependent types in an ad-hoc fashion, but instead start from a familiar data-oriented language and derive its dual fragment by the systematic use of defunctionalization and refunctionalization. Our central contribution is a dependently typed calculus which contains two dual language fragments. We provide type- and semantics-preserving transformations between these two language fragments: defunctionalization and refunctionalization. We have implemented this language and these transformations and use this implementation to explain the various ways in which constructions in dependently typed programming can be explained as special instances of the general phenomenon of duality.

CCS Concepts: • **Theory of computation** → *Lambda calculus*; *Type theory*.

Additional Key Words and Phrases: Dependent Types, Expression Problem, Defunctionalization

ACM Reference Format:
David Binder, Ingo Skupin, Tim Süberkrüb, and Klaus Ostermann. 2020. Deriving Dependently-Typed OOP from First Principles. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 1 (January 2020), 46 pages.

1 INTRODUCTION

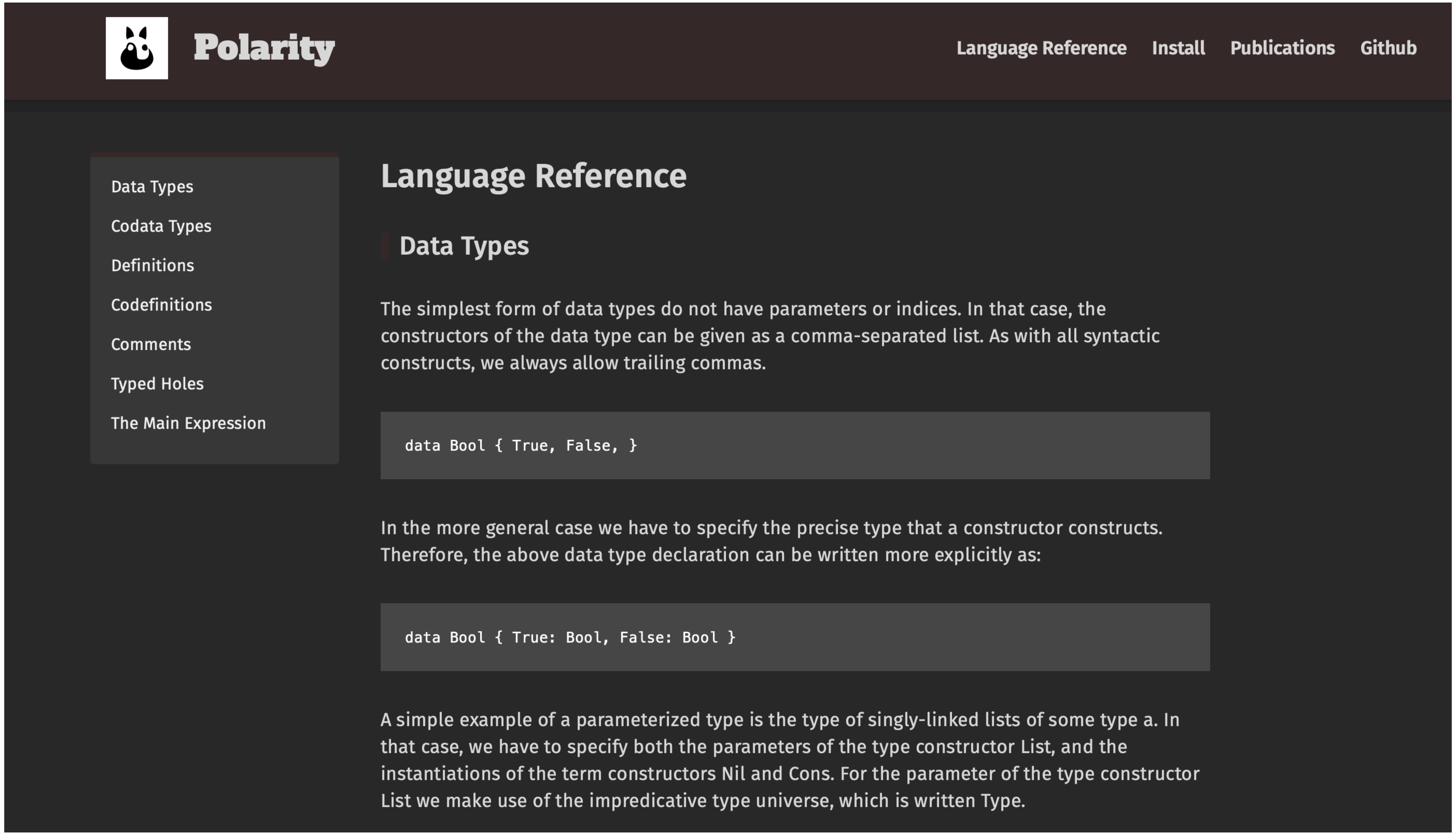
There are many programming paradigms, but dependently typed programming languages almost exclusively follow the functional programming model. In this paper, we show why dependently-typed programming languages should also include object-oriented principles, and how this can be done. One of the main reasons why object-oriented features should be included is a consequence of how the complexity of the domain is modeled in the functional and object-oriented paradigm. Functional programmers structure the domain using data types defined by their constructors, whereas object-oriented programmers structure the domain using classes and interfaces defined by methods. This choice has important implications for the extensibility properties of large programs, which are only more accentuated for dependently typed programs.

polarity-lang.github.io

We just made it public!

polarity-lang.github.io

We just made it public!



Polarity

Language Reference Install Publications Github

Data Types
Codata Types
Definitions
Codefinitions
Comments
Typed Holes
The Main Expression

Language Reference

Data Types

The simplest form of data types do not have parameters or indices. In that case, the constructors of the data type can be given as a comma-separated list. As with all syntactic constructs, we always allow trailing commas.

```
data Bool { True, False, }
```

In the more general case we have to specify the precise type that a constructor constructs. Therefore, the above data type declaration can be written more explicitly as:

```
data Bool { True: Bool, False: Bool }
```

A simple example of a parameterized type is the type of singly-linked lists of some type a. In that case, we have to specify both the parameters of the type constructor List, and the instantiations of the term constructors Nil and Cons. For the parameter of the type constructor List we make use of the impredicative type universe, which is written Type.

Subtyping and Type Inference

What are the types that we want to infer?

What do we want to achieve?

Example 1: When should we infer joins.

What do we want to achieve?

Example 1: When should we infer joins.

- What is the type of:

$$\lambda x\,.\,\textbf{if}\ x\ \textbf{then}\ \texttt{True}\ \textbf{else}\ 5$$

What do we want to achieve?

Example 1: When should we infer joins.

- What is the type of:

$$\lambda x \,.\, \mathbf{if}\ x\ \mathbf{then}\ \mathtt{True}\ \mathbf{else}\ 5$$

- We infer:

$$\mathbb{B} \to \mathbb{B} \sqcup \mathbb{N}$$

What do we want to achieve?

Example 1: When should we infer joins.

- What is the type of:

$$\lambda x\,.\,\mathbf{if}\ x\ \mathbf{then}\ \mathtt{True}\ \mathbf{else}\ 5$$

- We infer:

$$\mathbb{B} \to \mathbb{B} \sqcup \mathbb{N}$$

- Joins are for combining the types of multiple output paths.

What do we want to achieve?

Example 2: When should we infer meets?

What do we want to achieve?

Example 2: When should we infer meets?

- What is the type of:

$$\lambda x \ldots . (\texttt{not}\ x) \ldots (x + 1)$$

What do we want to achieve?

Example 2: When should we infer meets?

- What is the type of:

$$\lambda x \ldots . (\texttt{not}\ x) \ldots (x + 1)$$

- We infer:

$$\mathbb{B} \sqcap \mathbb{N} \rightarrow \ \ldots$$

What do we want to achieve?

Example 2: When should we infer meets?

- What is the type of:

$$\lambda x. \ldots (\texttt{not}\ x) \ldots (x + 1)$$

- We infer:

$$\mathbb{B} \sqcap \mathbb{N} \to \ldots$$

- Meets are for combining multiple requirements on inputs.

What do we want to achieve?

Example 3: When should we infer the top type?

What do we want to achieve?

Example 3: When should we infer the top type?

- What is the type of:

$$\lambda x.5$$

What do we want to achieve?

Example 3: When should we infer the top type?

- What is the type of:

$$\lambda x.5$$

- We infer:

$$\top \rightarrow \mathbb{N}$$

What do we want to achieve?

Example 3: When should we infer the top type?

- What is the type of:

$$\lambda x.5$$

- We infer:

$$\top \rightarrow \mathbb{N}$$

- The top type is for inputs which are ignored.

What do we want to achieve?

Example 3: When should we infer the top type?

- What is the type of:

$$\lambda x.5$$

- We infer:

$$\top \to \mathbb{N}$$

- The top type is for inputs which are ignored.

- The type variable in $\forall \alpha \,.\, \alpha \to \mathbb{N}$ is not needed, because it doesn't relate an input with an output.

How does type inference work?

Solving Inequality Constraints

The breakthroughs of Pottier and Dolan

Solving Inequality Constraints

The breakthroughs of Pottier and Dolan

- In Hindley-Milner type inference we have to solve equality constraints:

$$\{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}$$

Solving Inequality Constraints

The breakthroughs of Pottier and Dolan

- In Hindley-Milner type inference we have to solve equality constraints:

$$\{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}$$

- For algebraic subtyping we have to solve inequality constraints:

$$\{\sigma_1 <: \tau_1, \ldots, \sigma_n <: \tau_n\}$$

Solving Inequality Constraints

The breakthroughs of Pottier and Dolan

- In Hindley-Milner type inference we have to solve equality constraints:

$$\{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}$$

- For algebraic subtyping we have to solve inequality constraints:

$$\{\sigma_1 <: \tau_1, \ldots, \sigma_n <: \tau_n\}$$

- The details were figured out by F. Pottier and S. Dolan.

High School Algebra

What is a solution?

High School Algebra

What is a solution?

- The solution of a system of equalities

$$\{y = 3 + x,\ x = 2,\ y = z\}$$

  is an assignment of values to variables:

$$x := 2,\ y := 5,\ z := 5$$

High School Algebra

What is a solution?

- The solution of a system of equalities

$$\{y = 3 + x,\ x = 2,\ y = z\}$$

is an assignment of values to variables:

$$x := 2,\ y := 5,\ z := 5$$

- The solution of a system of inequalities

$$\{x \leq 2,\ x \leq y,\ y \leq 1,\ -2 \leq x,\ 0 \leq y\}$$

is an assignment of bounds to variables:

$$-2 \leq x \leq 1,\ 0 \leq y \leq 1$$

Core Idea: Keep Track of Variable Bounds

Core Idea: Keep Track of Variable Bounds

- We keep track of upper and lower bounds:

$$\{\sigma_1, \ldots, \sigma_n\} <: \alpha <: \{\tau_1, \ldots, \tau_n\}$$

Core Idea: Keep Track of Variable Bounds

- We keep track of upper and lower bounds:

$$\{\sigma_1, \ldots, \sigma_n\} <: \alpha <: \{\tau_1, \ldots, \tau_n\}$$

- When we solve a constraint $\alpha <: \xi$ we add it to the upper bounds

$$\{\sigma_1, \ldots, \sigma_n\} <: \alpha <: \{\tau_1, \ldots, \tau_n, \xi\}$$

Core Idea: Keep Track of Variable Bounds

- We keep track of upper and lower bounds:

$$\{\sigma_1, \ldots, \sigma_n\} <: \alpha <: \{\tau_1, \ldots, \tau_n\}$$

- When we solve a constraint $\alpha <: \xi$ we add it to the upper bounds

$$\{\sigma_1, \ldots, \sigma_n\} <: \alpha <: \{\tau_1, \ldots, \tau_n, \xi\}$$

- We have to make sure it is consistent with lower bounds:

$$\sigma_1 <: \xi, \ldots, \sigma_n <: \xi$$

Using Subtyping Type Inference for Better Error Messages

Getting Into the Flow

Towards Better Type-Error Messages

Getting into the Flow

Towards Better Type Error Messages for Constraint-Based Type Inference

ISHAN BHANUKA, HKUST, Hong Kong, China
LIONEL PARREAUX, HKUST, Hong Kong, China
DAVID BINDER, University of Tübingen, Germany
JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany

Creating good type error messages for constraint-based type inference systems is difficult. Typical type error messages reflect implementation details of the underlying constraint-solving algorithms rather than the specific factors leading to type mismatches. We propose using subtyping constraints that capture data flow to classify and explain type errors. Our algorithm explains type errors as faulty data flows, which programmers are already used to reasoning about, and illustrates these data flows as sequences of relevant program locations. We show that our ideas and algorithm are not limited to languages with subtyping, as they can be readily integrated with Hindley-Milner type inference. In addition to these core contributions, we present the results of a user study to evaluate the quality of our messages compared to other implementations. While the quantitative evaluation does not show that flow-based messages improve the localization or understanding of the causes of type errors, the qualitative evaluation suggests a real need and demand for flow-based messages.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Theory of computation** → **Program analysis**; **Type theory**; • **Human-centered computing** → **Human computer interaction (HCI)**.

Additional Key Words and Phrases: type inference, error messages, subtyping, data flow, constraint solving

ACM Reference Format:
Ishan Bhanuka, Lionel Parreaux, David Binder, and Jonathan Immanuel Brachthäuser. 2023. Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 237 (October 2023), 29 pages. https://doi.org/10.1145/3622812

1 INTRODUCTION

Much academic research has gone into producing better type error messages for functional programming languages, dating back at least to Wand [1986]. Yet, one would be none the wiser by looking at the error messages produced by existing compilers, including those compilers designed specifically with learning in mind, such as Helium [Heeren et al. 2003]. For example, consider the following OCaml program[1], where operator `(^)` stands for string concatenation:

```
let appInfo = ("My_Application", 1.5)
let process (name, vers) = name ^ show_major (parse_version vers)
let main() = process appInfo
```

Getting Into the Flow

Towards Better Type-Error Messages

- Presented at OOPSLA '23

Getting into the Flow

Towards Better Type Error Messages for Constraint-Based Type Inference

ISHAN BHANUKA, HKUST, Hong Kong, China
LIONEL PARREAUX, HKUST, Hong Kong, China
DAVID BINDER, University of Tübingen, Germany
JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany

Creating good type error messages for constraint-based type inference systems is difficult. Typical type error messages reflect implementation details of the underlying constraint-solving algorithms rather than the specific factors leading to type mismatches. We propose using subtyping constraints that capture data flow to classify and explain type errors. Our algorithm explains type errors as faulty data flows, which programmers are already used to reasoning about, and illustrates these data flows as sequences of relevant program locations. We show that our ideas and algorithm are not limited to languages with subtyping, as they can be readily integrated with Hindley-Milner type inference. In addition to these core contributions, we present the results of a user study to evaluate the quality of our messages compared to other implementations. While the quantitative evaluation does not show that flow-based messages improve the localization or understanding of the causes of type errors, the qualitative evaluation suggests a real need and demand for flow-based messages.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Theory of computation** → **Program analysis**; **Type theory**; • **Human-centered computing** → **Human computer interaction (HCI)**.

Additional Key Words and Phrases: type inference, error messages, subtyping, data flow, constraint solving

ACM Reference Format:
Ishan Bhanuka, Lionel Parreaux, David Binder, and Jonathan Immanuel Brachthäuser. 2023. Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 237 (October 2023), 29 pages. https://doi.org/10.1145/3622812

1 INTRODUCTION

Much academic research has gone into producing better type error messages for functional programming languages, dating back at least to Wand [1986]. Yet, one would be none the wiser by looking at the error messages produced by existing compilers, including those compilers designed specifically with learning in mind, such as Helium [Heeren et al. 2003]. For example, consider the following OCaml program[1], where operator `(^)` stands for string concatenation:

```
let appInfo = ("My_Application", 1.5)
let process (name, vers) = name ^ show_major (parse_version vers)
let main() = process appInfo
```

Getting Into the Flow

Towards Better Type-Error Messages

- Presented at OOPSLA '23
- Error messages for HM type inference are usually bad.

Check for updates

Getting into the Flow

Towards Better Type Error Messages for Constraint-Based Type Inference

ISHAN BHANUKA, HKUST, Hong Kong, China
LIONEL PARREAUX, HKUST, Hong Kong, China
DAVID BINDER, University of Tübingen, Germany
JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany

Creating good type error messages for constraint-based type inference systems is difficult. Typical type error messages reflect implementation details of the underlying constraint-solving algorithms rather than the specific factors leading to type mismatches. We propose using subtyping constraints that capture data flow to classify and explain type errors. Our algorithm explains type errors as faulty data flows, which programmers are already used to reasoning about, and illustrates these data flows as sequences of relevant program locations. We show that our ideas and algorithm are not limited to languages with subtyping, as they can be readily integrated with Hindley-Milner type inference. In addition to these core contributions, we present the results of a user study to evaluate the quality of our messages compared to other implementations. While the quantitative evaluation does not show that flow-based messages improve the localization or understanding of the causes of type errors, the qualitative evaluation suggests a real need and demand for flow-based messages.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Theory of computation** → **Program analysis**; **Type theory**; • **Human-centered computing** → **Human computer interaction (HCI)**.

Additional Key Words and Phrases: type inference, error messages, subtyping, data flow, constraint solving

ACM Reference Format:
Ishan Bhanuka, Lionel Parreaux, David Binder, and Jonathan Immanuel Brachthäuser. 2023. Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 237 (October 2023), 29 pages. https://doi.org/10.1145/3622812

237

1 INTRODUCTION

Much academic research has gone into producing better type error messages for functional programming languages, dating back at least to Wand [1986]. Yet, one would be none the wiser by looking at the error messages produced by existing compilers, including those compilers designed specifically with learning in mind, such as Helium [Heeren et al. 2003]. For example, consider the following OCaml program[1], where operator `(^)` stands for string concatenation:

```
let appInfo = ("My_Application", 1.5)
let process (name, vers) = name ^ show_major (parse_version vers)
let main() = process appInfo
```

Getting Into the Flow

Towards Better Type-Error Messages

- Presented at OOPSLA '23
- Error messages for HM type inference are usually bad.
- Type Inference with Subtyping Constraints can do better!

Check for updates

Getting into the Flow

Towards Better Type Error Messages for Constraint-Based Type Inference

ISHAN BHANUKA, HKUST, Hong Kong, China
LIONEL PARREAUX, HKUST, Hong Kong, China
DAVID BINDER, University of Tübingen, Germany
JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany

Creating good type error messages for constraint-based type inference systems is difficult. Typical type error messages reflect implementation details of the underlying constraint-solving algorithms rather than the specific factors leading to type mismatches. We propose using subtyping constraints that capture data flow to classify and explain type errors. Our algorithm explains type errors as faulty data flows, which programmers are already used to reasoning about, and illustrates these data flows as sequences of relevant program locations. We show that our ideas and algorithm are not limited to languages with subtyping, as they can be readily integrated with Hindley-Milner type inference. In addition to these core contributions, we present the results of a user study to evaluate the quality of our messages compared to other implementations. While the quantitative evaluation does not show that flow-based messages improve the localization or understanding of the causes of type errors, the qualitative evaluation suggests a real need and demand for flow-based messages.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Theory of computation** → **Program analysis**; **Type theory**; • **Human-centered computing** → **Human computer interaction (HCI)**.

Additional Key Words and Phrases: type inference, error messages, subtyping, data flow, constraint solving

ACM Reference Format:
Ishan Bhanuka, Lionel Parreaux, David Binder, and Jonathan Immanuel Brachthäuser. 2023. Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 237 (October 2023), 29 pages. https://doi.org/10.1145/3622812

237

1 INTRODUCTION

Much academic research has gone into producing better type error messages for functional programming languages, dating back at least to Wand [1986]. Yet, one would be none the wiser by looking at the error messages produced by existing compilers, including those compilers designed specifically with learning in mind, such as Helium [Heeren et al. 2003]. For example, consider the following OCaml program[1], where operator (^) stands for string concatenation:

```
let appInfo = ("My_Application", 1.5)
let process (name, vers) = name ^ show_major (parse_version vers)
let main() = process appInfo
```

One central idea!

We read $\sigma <: \tau$ as:

"A value of type σ flows into a position where a τ is expected"

Classify Constraint Solving Errors

Level-0 Error

```
let x = 2;
let y = if x then true else false;
```

Level-0 Error

```
let x = 2;
let y = if x then true else false;
```

$$\mathrm{Int} \longrightarrow \alpha_x \longrightarrow \mathrm{Bool}$$

Level-1 Error

```
let f x = (not x, x + 1);
```

Level-1 Error

```
let f x = (not x, x + 1);
```

$$\begin{tikzcd} & \mathrm{Int} \\ \alpha_x \arrow[ur] \arrow[dr] & \\ & \mathrm{Bool} \end{tikzcd}$$

Level-1 Error

```
let x = 2
let y = if true then x else "x"
```

Level-1 Error

```
let x = 2
let y = if true then x else "x"
```

$$
\begin{tikzcd}
\mathrm{Int} \arrow[dr] & \\
& \alpha_y \\
\mathrm{Str} \arrow[ur] &
\end{tikzcd}
$$

Level-2 Error

```
let g x = ( not x
          , if true then x else 5)
```

Level-2 Error

```
let g x = ( not x
          , if true then x else 5)
```

$$
\begin{tikzcd}
& \mathsf{Bool} \\
\alpha_x \arrow[ur] \arrow[dr] & \\
& \beta_{ret} \\
\mathsf{Int} \arrow[ur] &
\end{tikzcd}
$$

Level-n Errors

```
let x = 2;
let y = if x then true else false;
```

$$\text{Int} \longrightarrow \alpha_x \longrightarrow \text{Bool}$$

(a) Program with Level-0 error.

```
let f x = (not x, x + 1);
```

$$\begin{tikzcd}[column sep=large] & \text{Int} \\ \alpha_x \arrow[ur] \arrow[dr] & \\ & \text{Bool} \end{tikzcd}$$

```
let x = 2
let y = if true then x else "x"
```

$$\begin{tikzcd}[column sep=large] \text{Int} \arrow[dr] & \\ & \alpha_y \\ \text{Str} \arrow[ur] & \end{tikzcd}$$

(b) Two programs with different Level-1 errors.

```
let g x = ( not x
          , if true then x else 5)
```

$$\begin{tikzcd}[column sep=large] & \text{Bool} \\ \alpha_x \arrow[ur] \arrow[dr] & \\ & \beta_{ret} \\ \text{Int} \arrow[ur] & \end{tikzcd}$$

(c) Program with Level-2 error.

Fig. 3. Examples of faulty programs and their corresponding constraint graphs.

Explaining Type Errors With Data Flow

HM^{ℓ}

```
[ERROR] Type `int` does not match `string`

        (int) ---> (?a) <--- (string)

● (int) comes from
|  - l.1  let x = 2
|                 ^
|  - l.2  let y = if true then x else "x"
▼                              ^
● (?a) is assumed here
▲  - l.2  let y = if true then x else "x"
|                 ^^^^^^^^^^^^^^^^^^^^^^^
● (string) comes from
   - l.2  let y = if true then x else "x"
                                     ^^^
```

Fig. 5. Level-1 “confluence” error with convergent flows

Keeping Track of Data Flow in Constraints

Terms & Types

Annotate terms with locations

Location	$\ell ::= \text{program location}$
Term	$e ::= x^{\ell} \mid \texttt{unit}^{\ell} \mid \overline{n}^{\,\ell} \mid \texttt{true}^{\ell} \mid \texttt{false}^{\ell} \mid (\textbf{if } e \textbf{ then } e \textbf{ else } e)^{\ell} \mid e +^{\ell} e \mid (\lambda x^{\ell}.\ e)^{\ell} \mid (e\ e)^{\ell}$
	$\mid [e, e]^{\ell} \mid \pi_1(e)^{\ell} \mid \pi_2(e)^{\ell} \mid \iota_1(e)^{\ell} \mid \iota_2(e)^{\ell} \mid \text{case } e \text{ of } \{\, \iota_1(x^{\ell}) \Rightarrow e;\ \iota_2(x^{\ell}) \Rightarrow e \,\}^{\ell}$

Ordinary Term Language

Terms & Types

Annotate terms with locations

Annotated with locations

$$
\begin{array}{rrcl}
\textit{Location} & \ell & ::= & \text{program location} \\
\textit{Term} & e & ::= & x^{\ell} \mid \texttt{unit}^{\ell} \mid \overline{n}^{\,\ell} \mid \texttt{true}^{\ell} \mid \texttt{false}^{\ell} \mid (\textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e)^{\ell} \mid e +^{\ell} e \mid (\lambda x^{\ell}.\ e)^{\ell} \mid (e\ e)^{\ell} \\
 & & & \mid [e, e]^{\ell} \mid \pi_1(e)^{\ell} \mid \pi_2(e)^{\ell} \mid \iota_1(e)^{\ell} \mid \iota_2(e)^{\ell} \mid \text{case}\ e\ \text{of}\ \{\ \iota_1(x^{\ell}) \Rightarrow e;\ \iota_2(x^{\ell}) \Rightarrow e\ \}^{\ell}
\end{array}
$$

Ordinary Term Language

Terms & Types

Annotate types with provenances

$$\begin{array}{lrcl} \textit{Provenance} & p & ::= & p \cdot p \mid \epsilon \mid \ell \mid [p]_L^{\rightarrow} \mid [p]_R^{\rightarrow} \mid [p]_L^{\oplus} \mid [p]_R^{\oplus} \mid [p]_L^{\otimes} \mid [p]_R^{\otimes} \\ \textit{Type} & \tau, \delta & ::= & \alpha^p \mid \mathbf{1}^p \mid \mathsf{Int}^p \mid \mathsf{Bool}^p \mid \tau \rightarrow^p \tau \mid \tau \oplus^p \tau \mid \tau \otimes^p \tau \end{array}$$

Ordinary Types

Terms & Types

Annotate types with provenances

$$
\begin{array}{lrl}
\textit{Provenance} & p & ::= \; p \cdot p \mid \epsilon \mid \ell \mid [p]_L^{\rightarrow} \mid [p]_R^{\rightarrow} \mid [p]_L^{\oplus} \mid [p]_R^{\oplus} \mid [p]_L^{\otimes} \mid [p]_R^{\otimes} \\
\textit{Type} & \tau, \delta & ::= \; \alpha^p \mid \mathbf{1}^p \mid \mathsf{Int}^p \mid \mathsf{Bool}^p \mid \tau \rightarrow^p \tau \mid \tau \oplus^p \tau \mid \tau \otimes^p \tau
\end{array}
$$

Ordinary Types **Annotated with Provenances**

Generating and Solving Constraints

Two Judgement Forms

$$\boxed{\sigma \mid \Gamma \vdash e : \tau \mid \sigma} \quad \boxed{\sigma \mid \mathsf{cons}(Q)^{H} \mid \sigma}$$

$$\begin{array}{rrcl}
\textit{Constraint} & Q & ::= & \tau <: \tau \\
\textit{Context} & \Gamma & ::= & \epsilon \mid \Gamma \cdot (x : \alpha) \\
\textit{State} & \sigma & ::= & \{\, \mathsf{bounds} : \overline{\overline{\tau} <: \alpha <: \overline{\tau}},\ \mathsf{errors} : \overline{p} \,\}
\end{array}$$

Generating and Solving Constraints

Two Judgement Forms

$$\boxed{\sigma \mid \Gamma \vdash e : \tau \mid \sigma} \qquad \boxed{\sigma \mid \mathsf{cons}(Q)^{H} \mid \sigma}$$

Input:

$$
\begin{array}{rrcl}
\textit{Constraint} & Q & ::= & \tau <: \tau \\
\textit{Context} & \Gamma & ::= & \epsilon \mid \Gamma \cdot (x : \alpha) \\
\textit{State} & \sigma & ::= & \{\ \mathsf{bounds} : \overline{\overline{\tau} <: \alpha <: \overline{\tau}},\ \mathsf{errors} : \overline{p}\ \}
\end{array}
$$

Generating and Solving Constraints

Two Judgement Forms

$$\boxed{\sigma \mid \Gamma \vdash e : \tau \mid \sigma} \qquad \boxed{\sigma \mid \mathsf{cons}(Q)^{H} \mid \sigma}$$

Input:

Output:

$$
\begin{array}{rrcl}
\textit{Constraint} & Q & ::= & \tau <: \tau \\
\textit{Context} & \Gamma & ::= & \epsilon \mid \Gamma \cdot (x : \alpha) \\
\textit{State} & \sigma & ::= & \{\ \mathsf{bounds} : \overline{\overline{\tau} <: \alpha <: \overline{\tau}},\ \mathsf{errors} : \overline{p}\ \}
\end{array}
$$

Generating and Solving Constraints

Two Judgement Forms

$\boxed{\sigma \mid \Gamma \vdash e : \tau \mid \sigma}$ $\boxed{\sigma \mid \mathsf{cons}(Q)^{H} \mid \sigma}$

Input:

Output:

$$\begin{aligned}
\textit{Constraint} \quad & Q ::= \tau <: \tau \\
\textit{Context} \quad & \Gamma ::= \epsilon \mid \Gamma \cdot (x : \alpha) \\
\textit{State} \quad & \sigma ::= \{\, \mathsf{bounds} : \overline{\overline{\tau} <: \alpha <: \overline{\tau}},\ \mathsf{errors} : \overline{p} \,\}
\end{aligned}$$

Collect bounds for unification variables

Tracking Provenance

Dataflows begin in introduction forms

$$\frac{\textsc{T-Lit}}{\sigma \mid \Gamma \vdash \overline{n}^{\ell} : \mathsf{Int}^{\ell} \mid \sigma}$$

Tracking Provenance

Dataflows begin in introduction forms

$$\frac{\textsc{T-Lit}}{\sigma \mid \Gamma \vdash \overline{n}^{\ell} : \mathsf{Int}^{\ell} \mid \sigma}$$

Dataflow starts at integer literal

Tracking Provenance

Dataflows end in elimination forms

T-Plus

$$\frac{\sigma_0 \mid \Gamma \vdash e_0 : \tau_0 \mid \sigma_1 \qquad \sigma_1 \mid \mathrm{cons}(\tau_0 <: \mathsf{Int}^{\ell}) \mid \sigma_2 \qquad \sigma_2 \mid \Gamma \vdash e_1 : \tau_1 \mid \sigma_3 \qquad \sigma_3 \mid \mathrm{cons}(\tau_1 <: \mathsf{Int}^{\ell}) \mid \sigma_4}{\sigma_0 \mid \Gamma \vdash e_0 +^{\ell} e_1 : \mathsf{Int}^{\ell} \mid \sigma_4}$$

Tracking Provenance

Dataflows end in elimination forms

Dataflow ends at addition

T-Plus

$$\frac{\sigma_0 \mid \Gamma \vdash e_0 : \tau_0 \mid \sigma_1 \qquad \sigma_1 \mid \mathsf{cons}(\tau_0 <: \mathsf{Int}^{\ell}) \mid \sigma_2 \\ \sigma_2 \mid \Gamma \vdash e_1 : \tau_1 \mid \sigma_3 \qquad \sigma_3 \mid \mathsf{cons}(\tau_1 <: \mathsf{Int}^{\ell}) \mid \sigma_4}{\sigma_0 \mid \Gamma \vdash e_0 +^{\ell} e_1 : \mathsf{Int}^{\ell} \mid \sigma_4}$$

Tracking Provenance

Provenance passes through some constructs

T-IfThenElse

$$\frac{\begin{array}{c}\alpha \textit{ fresh} \\ \sigma_0 \mid \Gamma \vdash e_1 : \tau_1 \mid \sigma_1 \qquad \sigma_1 \mid \Gamma \vdash e_2 : \tau_2 \mid \sigma_2 \qquad \sigma_2 \mid \Gamma \vdash e_3 : \tau_3 \mid \sigma_3 \\ \sigma_3 \mid \mathsf{cons}(\tau_1 <: \mathsf{Bool}^{\ell}) \mid \sigma_4 \qquad \sigma_4 \mid \mathsf{cons}(\tau_2 <: \alpha^{\ell}) \mid \sigma_5 \qquad \sigma_5 \mid \mathsf{cons}(\tau_3 <: \alpha^{\ell}) \mid \sigma_6\end{array}}{\sigma_0 \mid \Gamma \vdash (\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)^{\ell} : \alpha^{\ell} \mid \sigma_6}$$

Tracking Provenance

Provenance passes through some constructs

Dataflow passes through if-then-else

T-IfThenElse

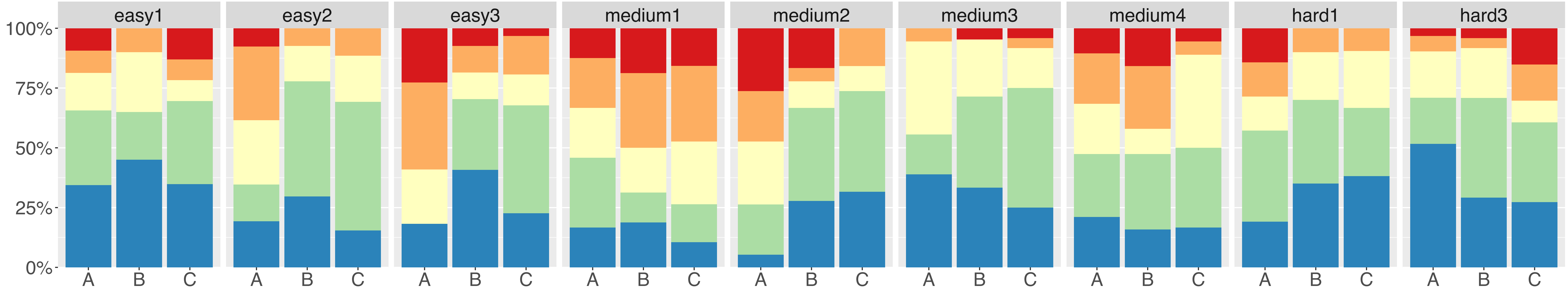
$$\frac{\alpha \textit{ fresh} \qquad \begin{array}{ccc} \sigma_0 \mid \Gamma \vdash e_1 : \tau_1 \mid \sigma_1 & \sigma_1 \mid \Gamma \vdash e_2 : \tau_2 \mid \sigma_2 & \sigma_2 \mid \Gamma \vdash e_3 : \tau_3 \mid \sigma_3 \\ \sigma_3 \mid \mathsf{cons}(\tau_1 <: \mathsf{Bool}^{\ell}) \mid \sigma_4 & \sigma_4 \mid \mathsf{cons}(\tau_2 <: \alpha^{\ell}) \mid \sigma_5 & \sigma_5 \mid \mathsf{cons}(\tau_3 <: \alpha^{\ell}) \mid \sigma_6 \end{array}}{\sigma_0 \mid \Gamma \vdash (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)^{\ell} : \alpha^{\ell} \mid \sigma_6}$$

Empirical Evaluation

Empirical Results

Location and Understandability

Q2: “How much did the error message help you to locate the problem?”



Q3: “How much did the error message help you to understand the problem?”

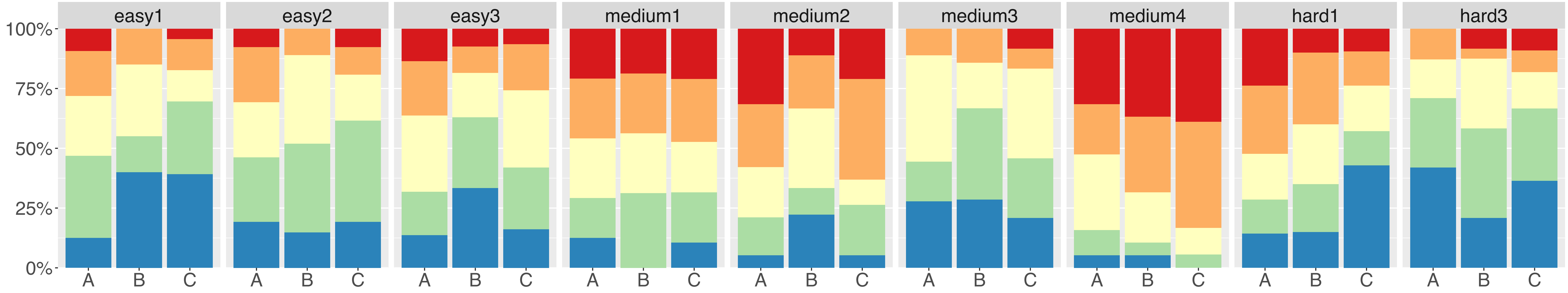


Fig. 9. Participants answering the respective question on a five-point Likert scale from “Not helpful” (top, red) to “Very helpful” (bottom, blue). We compare conditions HM^{ℓ} (A), OCaml (B), and Helium (C).

Ongoing and Future Work

Using Data Flow as an Explanatory Device

Useful for more than just typechecking?

- Our hypothesis is that (functional) programmers reason about programs using data flow.
- If that is the case, then data flow is a good explanatory device when explaining errors.
- We showed how to do it for type inference, but what about: Effect systems, type classes, linear type systems, region based memory management.

Time for your questions!