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:

```
4 let appInfo = ("My␣Application", 1.5)
5 let process (name, vers) = name ^ show_major (parse_version vers)
6 let main() = process appInfo
```

[1]We use OCaml syntax for all code examples because of its prevalence in error localization literature.

Authors' addresses: Ishan Bhanuka, HKUST, Hong Kong, China; Lionel Parreaux, HKUST, Hong Kong, China; David Binder, University of Tübingen, Tübingen, Germany; Jonathan Immanuel Brachthäuser, University of Tübingen, Tübingen, Germany.

2475-1421/2023/10-ART237
https://doi.org/10.1145/3622812

Here, we assume the following definitions imported from some library:

```
val show_major    : string -> string
val parse_version : string -> string
```

This program contains a type error. Below is the error reported by the OCaml compiler (v. 4.14.0):

```
OCaml
   1.6:  let main() = process appInfo
Error: This expression has type string * float
       but an expression was expected of type string * string
       Type float is not compatible with type string
```

It is not immediately clear, just from looking at this report, what caused the problem and how to go about fixing it, unless one is already familiar with the source code and has it fresh in their mind—note that the definitions in our little program above could be very far apart in a real-world scenario. The error seems to provide exactly as much information as the type inference engine had on hand at the time it encountered a problem and little to no contextual information is provided, which could have been helpful. Error reports produced by most other existing compilers for functional programming languages are not significantly different than this.

1.1 Flow-Based Error Messages

How come the wealth of previous ideas for improving ML type errors has not yet permeated modern compiler design practice? This could be for a number of reasons. Perhaps the previously-proposed approaches were too difficult to implement or to integrate into existing type systems; or they were too unreliable and their heuristics too difficult to tune; or perhaps the corresponding explanations were not actually helpful to real programmers.

In this paper, we set out to start addressing these questions by:

- proposing a *straightforward*, *heuristics-free* approach to recording and reproducing the information relevant to ML type errors in terms of *data flows*, a concept that we expect users can get used to quickly, because it relates to how programs evaluate; and by
- performing a randomized quasi-experimental study to evaluate whether our approach does help programmers understand the type errors found in actual ML programs. We compare the error messages we produce to those generated by `ocamlc` and Helium [Heeren et al. 2003][2].

The approach we propose, which we dub HM$^\ell$ (to be read as "*H-M-loc*"), produces the error message in Figure 1 when given the same program as above. This report adds several helpful bits of context to the error. Most importantly, instead of displaying a single erroneous location, it presents, in a logical order, each location involved in the erroneous *data flow*. This report illustrates the flow of data from the right-hand-side component of the `appInfo` pair into the `vers` parameter of function `process`, and the flow of this `vers` parameter into function `parse_version`, which is imported from some library. Our type error is caused by this specific erroneous data flow.

Note that if the abbreviated report above is still not enough for the user to resolve the issue at hand, a further detailed explanation can be obtained. HM$^\ell$ supports a verbose mode that lists the entire sequence of locations involved in a type mismatch, including those locations that go through nested type constructors. Verbose error messages can become very long and unwieldy. A promising future direction for our work will be to enable interactive type error debugging in

[2]Unfortunately, bit-rot seems to have gotten the better of the vast majority of historical type error reporting systems from academia, so we are only able to compare with these two.

```
HM^ℓ
[ERROR] Type `float` does not match `string`

● (float) comes from
|   - 1.4  let appInfo = ("My Application", 1.5)
|                                           ^^^
|   - 1.5  let process (name, vers) =
|                            ^^^^
|   - 1.9  name ^ show_major (parse_version vers)
▼                                           ^^^^
● (string) comes from
    - 1.2  val parse_version : string -> string
                               ^^^^^^
```

Fig. 1. Simple example of HM$^\ell$ error message.

integrated development environments (IDEs), whereby users will be able to explore the data flows involved in a type error interactively.

1.2 Complex Type Errors

The above program is simplistic and the erroneous data flow is easy to understand. In more complex scenarios, errors can arise from two types flowing *into* or *from* the same location, which we refer to as *confluence* errors. Data flows are further complicated when types flow *through constructors*. Later in the paper, we detail how we propose to handle these more advanced typing errors in our error reporting system (Section 4). As an early example, consider the following linear algebra program:

```
1 let move (x, y) = (x / 2, y / 2)
2 let dist (x, y) = x *. x +. y *. y
3 let move_closer pos =
4   if dist pos < 25.0 then pos else move pos
```

The OCaml compiler gives the following error for this program, which again lacks context:

```
OCaml
File line 3, characters 59-62:
3 |   let move_closer pos = if dist pos < 25.0 then pos else move pos
Error: This expression has type float * float
       but an expression was expected of type int * int
       Type float is not compatible with type int
```

On the other hand, Figure 2 shows HM$^\ell$'s verbose error report, where the flows of all values and how they pass through constructors is precisely described:

This error report introduces new syntax that we explain below.

- The indented parts are those corresponding to *nested constructor flows*—here, the `float` value does not immediately flow into an `int` position, but rather flows into the left-hand side of a pair, which itself has its own flow. This nested flow is important to understand the entire context of the error and is shown in the verbose error reports.
- The labels `?a`, `?b`, and `?c` are placeholders for types. `?b` is the label for the argument of the `move_closer` function and it flows into two use sites. It is used as the argument to the `dist` function which expects a pair. The left type argument of the pair is labelled `?a` which flows

HM$^{\ell}$

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

● (float) comes from
▲  - lib. let ( *. ): float -> float -> float
|                     ^^^^^
|  - l.2  let dist (x, y) = x *. x +. y *. y
|                           ^
|
● (?a) is assumed for
   - l.2  let dist (x, y) = x *. x +. y *. y
                  ^
  ● (?a * _) comes from
  ▲  - l.2  let dist (x, y) = x *. x +. y *. y
  |                  ^^^^^^
  |  - l.4      if dist pos < 25.0 then pos else move pos
  |                    ^^^
  |  - l.3  let movecloser pos =
  |                        ^^^
  |
  ● (?c) is assumed for
  |  - l.3  let movecloser pos =
  |                        ^^^
  |  - l.4      if dist pos < 25.0 then pos else move pos
  |                                             ^^^
  ▼
  ● (?b * _) comes from
     - l.1  let move (x, y) = (x / 2, y / 2)
                     ^^^^^^
● (?b) is assumed for
|  - l.1  let move (x, y) = (x / 2, y / 2)
|                  ^
|  - l.1  let move (x, y) = (x / 2, y / 2)
|                           ^
▼
● (int) comes from
   - lib. let ( / ): int -> int -> int
                     ^^^
```

Fig. 2. Complex example of HM$^{\ell}$ error message.

into a location expecting a `float`. Label `?b` is also used as the argument to the `move` function which also expects a pair with the left type argument labelled as `?c`. Type `?c` flows into a location expecting an `int`. The data flow, presented like this, shows how two incompatible types are being unified causing a type error.

This program represents a more realistic example where composing different functions together can lead to erroneous data flows.

1.3 Contributions

Our new approach is based on a theory of type provenance tracking. A key observation is that we have to treat type equality constraints $\tau_1 = \tau_2$ as *asymmetric*, since such type constraints are read as information about a value flowing from a source where it has been introduced with type τ_1 to a usage site where it will be used at type τ_2. This asymmetry suggest to look for inspiration from constraint-based inference algorithms for *subtyping* constraints of the form $\tau_1 <: \tau_2$, which are naturally asymmetric. In this paper, we use the algebraic subtyping approach and algorithms developed by Dolan [2017]; Dolan and Mycroft [2017] and simplified by Parreaux [2020], and combine it with the idea of Gast [2005] to use data flow for explaining error messages. While we use subtype inference to improve the quality of error messages, our approach targets the familiar type theory of Hindley, Damas and Milner as the user-facing type system.

Specifically, we make the following contributions:

- A *classification system for unification errors* based on data flow, where each unification error is assigned a numeric level (Section 2). The classification allows us to speak about Level-*n* errors, and to craft error messages specific for each level. We suggest that it is crucially important to use different textual explanations when explaining type errors of different levels.
- A *subtyping constraint solving algorithm* which reports data flow-based error messages for *Level-0 errors*, which is very close to the one used in algebraic subtyping but additionally tracks the provenances of types and flows in the program (Section 3). This demonstrates an observation of the algebraic subtyping community that it is easier to create helpful error messages from a subtype-inference-based system rather than from a unification-based one.
- An *equality-constraint solving algorithm* which reports data-flow-based error messages for *Level-n errors* (where $n \geq 1$). This algorithm is close to unification-based algorithms, but also tracks provenances of types and flows (Section 4).
- *A user study* to empirically evaluate our error messages and to help us guide further research into improving their quality (Section 5). The experiment compares the effects of HM^ℓ, `ocamlc`, and Helium on programmers' ability to understand and localize type errors. While the quantitative evaluation does not show that HM^ℓ provides any measurable improvement over the state-of-the-art, a qualitative analysis suggests the demand for flow-based errors in situations with complex type errors.

We provide an implementation of HM^ℓ as an extension of Simple-sub [Parreaux 2020]. Our system type checks a reasonable subset of OCaml features while providing high-quality error messages[3].

2 CLASSIFYING TYPE ERRORS

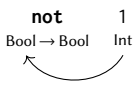
Not all unification errors are created equal. By treating them as equal, compiler engineers pass up an opportunity to improve the quality of the error messages that we can generate. Independent of the constraint algorithm we use, eventually the type checker might come to a point where it has discovered enough information to conclude that two incompatible types, such as `Bool` and `Int`, should be equal, at which point it emits an error. Following this line of thought, we might conclude there is only one essential kind of unification error, namely that two types are incompatible. Accordingly, most type checkers only use one textual template to display these errors to the user. The error messages might be enriched with additional information about the typing context in which the error arose, or about the source code region for which it was generated, but the underlying textual template often stays the same.

[3]Our implementation is permanently available on Zenodo [Bhanuka et al. 2023]. A web demo is available on the repository which hosts the latest version of our implementation. The repository is hosted at github.com/hkust-taco/hmloc.

In this section, we argue that this uniform view holds us back if we want to create great error messages for the user. To improve error messages that arise from type unification, as a first step, we realize that not all unification errors are the same and introduce a precise *classification of unification errors*. Based on this classification, as a second step, it is then possible to craft a textual error message for each kind of unification error, instead of using one fits-all template. As we will see, we classify different constraint solving errors using the *direction of data flow in the program*.

2.1 Flow of Types

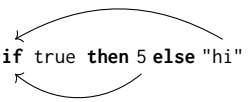
Let us assume that we typecheck the faulty expression `not 1`. Traditionally, one would generate a type constraint expressing that Int (the type of the literal `1`) has to be equal to Bool (the argument type of the `not` function). However, we can observe that this information is directed and closely corresponds to the data flow:



In analogy to the well-known concept of data flow, we argue that programmers can reason about the flow of types to understand faulty programs. In the above example, we say that the argument type Int *flows into* the parameter type Bool. It would be incorrect to generate a constraint for this expression which says that Bool flows into Int. Most standard unification algorithms discard this directionality information, since they make implicit use of the rule of symmetry to solve constraints. In these algorithms, the type equality constraint $\tau_1 = \tau_2$ is considered equivalent to the constraint $\tau_2 = \tau_1$. As a first technical insight, we thus recognize that we have to use non-symmetric constraints if we want to preserve directionality information during constraint solving. Luckily, there already is a ready-made notion for non-symmetric constraints: *subtyping constraints* $\tau_1 <: \tau_2$ which express that τ_1 has to be a subtype of τ_2. We will see that we can equivalently interpret these subtyping constraints as expressing that a value of type τ_1 flows into a context which expects a value of type τ_2, and that this reading is independent of whether we consider a system with subtyping or without.

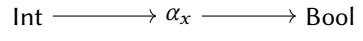
2.2 Change of Direction

The flow of types provides us with a different explanation model for type errors. In our example above, the flow was excessively short. In realistic programs, the distance between the point where a type is introduced (like type Int) and the point where it collides with a different expected type (type Bool in our example) can be arbitrarily large. Furthermore, type errors are not only introduced when one type flows directly into another incompatible one, but also if two incompatible types flow into a single location. This is the case in the following example, where both Int and Str flow into the result type of the conditional expression:



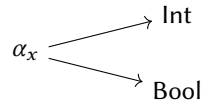
We refer to these type errors as *confluence* errors. When type checking a program like the one above, we would gather the two constraints Str <: α_{ret} and Int <: α_{ret}, where α_{ret} is a unification variable corresponding to the result of the conditional. While there is an obvious type error in the program, just given the constraints we cannot immediately derive an inconsistency. In order to do so, we would have to invoke the rule of symmetry. As a second technical insight, we observe that invoking symmetry corresponds to a change of direction in the flow of types: Str <: α_{ret} :> Int following the type flow from Str to Int, we can notice that it reverses direction once. Generalizing this observation, we present the following classification of type errors.

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

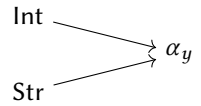


(a) Program with Level-0 error.

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

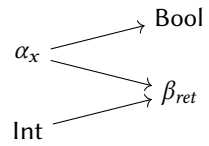



```
let x = 2
let y = if true then x else "x"
```



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

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



(c) Program with Level-2 error.

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

Definition 2.1. In a Level-n unification error, the derivation of the contradiction has the form of a chain of subtyping constraints τ <:> ... <:> τ' (with $\tau \neq \tau'$ and <:> denoting either <: or :>), where the direction of the subtyping constraints changes n times. Each change of direction corresponds to a reversal of the data flow which has to be explained to the user. While the rule of symmetry allows HM type inference algorithms to ignore this information about data flow, retaining it is important to properly explain the cause of the type error.

To see why this classification is useful, in the remainder of this section, we will consider concrete examples containing errors of various levels in Figure 3 and the corresponding error messages that we generate.

2.2.1 Level-0 Errors. The snippet in Figure 3a contains a Level-0 error. In this example, we have to introduce one unification variable α_x for the let-bound program variable x, and two constraints Int <: α_x and α_x <: Bool. The first constraint expresses that the type Int, introduced by the literal 2, flows into the variable x, while the second constraint expresses that the type of the variable x flows into the condition of the if-then-else expression which expects a boolean. These constraints are presented in the corresponding graph as arrows, with the direction of the arrow corresponding to the flow of data through the program. From these two constraints we can deduce the inconsistency Int <: Bool *without* having to reverse the data flow in the constraints. This means that we can directly explain the error as the flow from one type to the other as shown in Figure 4. Level-0 errors allow for a good textual explanation in error messages, since we can point to a location in the program where a value of a certain type was introduced, follow it through intermediate bindings and point to a position in the code where a different type was expected. Programs containing such Level-0 errors should always be rejected by a typechecker, since executing such programs would result in type mismatch errors at runtime (*i.e.,* non-value terms getting stuck and not reducing further). Such programs are therefore rejected by both systems which support subtyping and systems which do not.

2.2.2 Level-1 Errors. In Figure 3b we have two snippets which both exhibit Level-1 errors. In the first of these snippets we have an if-then-else expression with incompatible cases for the if and else

```
HM^ℓ
[ERROR] Type `int` does not match `bool`

● (int) comes from
|  - l.1  let x = 2;
▼                 ^
● (bool) comes from
   - l.2  let y x = if x then true else false
                    ^
```

Fig. 4. Level-0 error.

```
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

branch. During constraint generation we would generate a unification variable α_{ret} for the return type of the if-then-else expression, and two constraints: The constraint Int <: α_{ret} for the if branch and the constraint Str <: α_{ret} for the else branch. But taken together, these constraints are only contradictory if we reverse the data flow once in the chain Int <: α_{ret} :> Str. Figure 5 shows the error message.

In a system with subtyping and union and intersection types it would be possible to assign the type Int ⊔ Str to the expression. This shows that it is not strictly necessary to reject this expression, since the evaluation of this expression cannot lead to type unsoundness in itself, as long as its context can handle both an integer and a string.

The second example in Figure 3b exhibits a different Level-1 error. Here we have to generate a unification variable α_x for the lambda-bound variable x, and two constraints for the two different uses of x in either side of the tuple. The error is presented in Figure 6. A system with support for subtyping could assign the type (Int ⊓ Bool) ⟶ (Bool, Int) to the expression, *i.e.,* a function which can only be called with a value which can act as both an integer and a boolean.

2.2.3 Level-2 Errors. If we combine the two snippets from Figure 3b we obtain the example from Figure 3c which exhibits a Level-2 error. Here two unification variables, α_x and β_{ret} have to be generated, together with three constraints, and we have to change the direction of data flow twice

HM$^{\ell}$

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

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

● (int) comes from
▲  - lib. let (+): int -> int -> int
|                  ^^^
|  - 1.1  let f x = (not x , x + 1)
|                           ^
● (?a) is assumed here
|  - 1.1  let f x = (not x , x + 1)
|               ^
|  - 1.1  let f x = (not x , x + 1)
▼                       ^
● (bool) comes from
   - lib. let not: bool -> bool
                   ^^^^
```

Fig. 6. Level-1 error with divergent flows.

to obtain the inconsistency between Bool and Int. We conjecture that these kind of errors will already be quite rare in practice, even more so for errors of even higher levels, and even for a human it is no longer clear how the best error message should look like in this case. But the algorithm that we present is still able to provide an explanation, mentions all the essential information, and is much more informative than what other implementations provide. We show the error message for this example in Figure 7.

2.3 Reporting Different Levels

How do different type inference algorithms deal with these different errors?

Algebraic subtyping algorithms like MLsub usually only report Level-0 errors, and not any of the higher-level-errors ($n > 0$), as these are not considered errors in this typing discipline. This is why it has been remarked, in the subtyping literature, that it should be easier to generate good error messages for a system based on subtyping—the error messages only have to explain a linear and obviously problematic data flow of information through the program.

Standard unification algorithms, on the other hand, have to account for Level-n errors for arbitrary n, since symmetry is always valid for equality constraints. However, since these algorithms usually do not track the reversal of the direction of data flow in the constraint solving process, the same textual explanation is used for all unification errors, regardless of their level.

Our classification now allows us to design detailed and specific error messages for both systems with and without subtyping. If we have a system with subtyping, we only recognize Level-0 errors as proper errors, and display and explain them accordingly. We describe how to do this in Section 3. If, on the other hand, we are interested in a system which recognizes the same errors as a standard unification algorithm, then we have to recognize and explain errors for all levels. In Section 4, we extend the algorithm from Section 3 to emulate a standard unification algorithm, and recognize errors for all the levels. However unlike standard unification algorithm, we keep track of when the direction of data flow is reversed and report the full data flow for an error.

HM$^{\ell}$

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

        (bool) <--- (?a) ---> (?b) <--- (int)

● (bool) comes from
▲  - lib. let not: bool -> bool
|                  ^^^^
|  - l.1  let g x = ( not x
|                        ^
● (?a) is assumed here
|  - l.1  let g x = ( not x
|                ^
|  - l.2            , if true then x else 5)
▼                                 ^
● (?b) is assumed here
▲  - l.2            , if true then x else 5)
|                     ^^^^^^^^^^^^^^^^^^^^
● (int) comes from
   - l.2            , if true then x else 5)
                                       ^
```

Fig. 7. Level-2 error.

3 FORMALIZATION

In this section, we make the intuitions described in the previous sections formally precise. There are two important properties of type inference in ML-style languages we omit in this section: let-generalization and the occurs-check. We expect that both features will integrate well with the approach we have described in this paper, but leave the details for future work.

3.1 Terms and Locations

Figure 8 defines the syntax of terms e. The presentation is fairly standard, but since we want to track the flow of information through the program, we need a way to refer to the *locations* of subexpressions within the program. For this reason, every subexpression and every binding site of a variable is annotated with a program location ℓ. In this article, we do not commit to any actual representation for program locations, but in our implementation we choose one based on line and column number ranges. We also omit these locations in examples and explanations, or whenever they are not necessary.

Syntax of Terms. Terms themselves consist of variables x, the unit constructor `unit`, integer literals $\overline{n}$ and integer addition $e{+}e$. Booleans are constructed with literals `true` and `false`, and eliminated using the conditional **if** e **then** e **else** e. Functions are introduced using lambda abstraction $\lambda x.e$ and eliminated using function application $e\ e$. Pairs are constructed using the pairing constructor $[e, e]$ and deconstructed using projections $\pi_1(e)$ and $\pi_2(e)$. Sums are constructed using injections $\iota_1(e)$ and $\iota_2(e)$, and deconstructed using the pattern matching construct case e of $\{\, \iota_1(x) \Rightarrow e;\ \iota_2(x) \Rightarrow e \,\}$.

3.2 Types and Provenances

Where terms are annotated with locations, types are annotated with *provenances* (defined in Figure 8). These provenances p explain *why* a certain type is used at a specific point in the program,

and they are recorded and recombined during the type inference process. Provenances are also used to report errors; in that case they explain the flow of information through the program that led to the mismatch of two types. A provenance records a linear path through the program, so we have an operation $\cdot$ to concatenate two paths, and its unit ϵ standing for the empty path. Provenance concatenation is taken to be an associative operation where the *absent provenance* ϵ is taken to be the empty element. Therefore, for example, $(p_0 \cdot p_1) \cdot p_2$ is the same as $p_0 \cdot (p_1 \cdot p_2)$ and is simply written $p_0 \cdot p_1 \cdot p_2$. Similarly, $\epsilon \cdot p_0 \cdot \epsilon \cdot p_1 \cdot \epsilon$, is the same as $p_0 \cdot p_1$. We also use locations ℓ in provenances, to record specific points in the flow of information through the program. We will introduce and motivate the remaining syntactic forms of provenances in Section 3.3.3, where they are used in the constraint solving process.

Syntax of Types. The type forms are standard. We have type variables α, the unit type $\mathbf{1}$, and primitive types Int and Bool. We have three binary type constructors: the function type $\rightarrow$, the product type $\otimes$ and the sum type $\oplus$. As mentioned above, these are all annotated with provenances p. Just as with terms, we will sometimes omit these provenances in examples and explanations. In order to show how the different parts of an annotated type correspond to different parts of the information flow, we consider a very simple example.

Example 3.1. The inferred type of the term $[5^{\ell_1}, \texttt{unit}^{\ell_2}]^{\ell_3}$ is $\mathsf{Int}^{\ell_1} \otimes^{\ell_3} \mathbf{1}^{\ell_2}$.

The above example shows that in the inferred type, the top-level provenance ℓ_3 only contains the information of the flow explaining the outermost type constructor _ $\otimes$ _. It does not contain the information about the provenance of its arguments. These provenances are annotated in the arguments to _ $\otimes$ _, namely Int^{ℓ_1} and $\mathbf{1}^{\ell_2}$.

$$
\begin{array}{rrl}
\textit{Location} & \ell & ::= \text{ program location} \\
\textit{Term} & e & ::= x^{\ell} \mid \texttt{unit}^{\ell} \mid \overline{n}^{\ell} \mid \texttt{true}^{\ell} \mid \texttt{false}^{\ell} \mid (\textbf{if } e \textbf{ then } e \textbf{ else } e)^{\ell} \mid e +^{\ell} e \mid (\lambda x^{\ell}.\, e)^{\ell} \mid (e\ e)^{\ell} \\
& & \quad \mid [e, e]^{\ell} \mid \pi_1(e)^{\ell} \mid \pi_2(e)^{\ell} \mid \iota_1(e)^{\ell} \mid \iota_2(e)^{\ell} \mid \mathsf{case}\ e\ \mathsf{of}\ \{\, \iota_1(x^{\ell}) \Rightarrow e;\ \iota_2(x^{\ell}) \Rightarrow e \,\}^{\ell} \\
\\
\textit{Provenance} & p & ::= p \cdot p \mid \epsilon \mid \ell \mid \langle p \rangle_L^{\rightarrow} \mid \langle p \rangle_R^{\rightarrow} \mid \langle p \rangle_L^{\oplus} \mid \langle p \rangle_R^{\oplus} \mid \langle p \rangle_L^{\otimes} \mid \langle p \rangle_R^{\otimes} \\
\textit{Type} & \tau, \delta & ::= \alpha^{p} \mid \mathbf{1}^{p} \mid \mathsf{Int}^{p} \mid \mathsf{Bool}^{p} \mid \tau \rightarrow^{p} \tau \mid \tau \oplus^{p} \tau \mid \tau \otimes^{p} \tau \\
\textit{Constraint} & Q & ::= \tau <: \tau \\
\\
\textit{Context} & \Gamma & ::= \epsilon \mid \Gamma \cdot (x : \alpha) \\
\textit{State} & \sigma & ::= \{\, \mathsf{bounds} : \overline{\overline{\tau} <: \alpha <: \overline{\tau}},\ \mathsf{errors} : \overline{p} \,\}
\end{array}
$$

Fig. 8. Syntax of terms and types.

3.3 Type Inference

Our type inference algorithm is an extension of a particular implementation of algebraic subtyping, due to Parreaux [2020]. For readers who want to grasp the full extent of the algebraic subtyping technique and the associated proofs of correctness, we suggest referring to the relevant literature [Dolan 2017; Dolan and Mycroft 2017]. Note that locations and provenances do not affect type inference; in the algorithm presented in the following sections, it is easy to see that erasing all mentions of locations and type provenance tracking from our algorithm results in the algorithm by Parreaux. Since algebraic subtyping accepts strictly more programs than Hindley-Milner type inference, it follows that all programs rejected by the algorithm in this section are also rejected by Hindley-Milner. However, programs which only exhibit level-n errors (with $n \geq 1$) are accepted

by the algorithm in this section, but rejected by Hindley-Milner type inference. Since the goal of our approach is to improve the quality of error messages, and not to change the set of accepted programs, we add an additional phase described in Section 4 so that the same programs are accepted by our algorithm and the standard unification based algorithms.

The type inference algorithm consists of three parts: The algorithmic inference rules, discussed in Section 3.3.1, the constraint solving algorithm, discussed in Section 3.3.2, and the computation of subconstraints, discussed in Section 3.3.3.

3.3.1 Algorithmic Inference Rules. The type inference judgement $\sigma \mid \Gamma \vdash e : \tau \mid \sigma'$ is specified in Figure 9. It takes a type inference state σ, a typing context Γ, and a term e, and returns a type τ along with a new type inference state σ'. This type inference state consists of the lower and upper bounds for each type variable, and a list of errors that were generated during type inference. To focus on the essential aspects of a rule, we fade out the state σ when it is only threaded through.

The top-level judgement $\emptyset \mid \epsilon \vdash e : \tau \mid \sigma$, also written $\vdash e : \tau \mid \sigma$, tells us whether a term e is well-typed: if there exists a p such that $\mathrm{err}\, p \in \sigma$, then we say that e is ill-typed and p is a type provenance chain highlighting one of its type collision errors; otherwise, we say that e is well-typed.

We rely on the usual informal notion of freshness for type variables—α is "fresh" if it does not appear anywhere in the previous values of σ, nor in the previous parts of the premise of a typing rule.

3.3.2 Constraint Solving Algorithm. The constraint solving algorithm is specified in Figure 10. The type constraining function $\sigma \mid \mathrm{cons}(Q)^H \mid \sigma'$ takes a type inference state σ, a constraint Q, and a set of current hypotheses H, and returns a new state σ'.

We introduce two helper functions 'add-lb' and 'add-ub' to add a type, respectively, to the upper and lower bounds of a type variable in the type inference state. For e.g. $\mathrm{add\text{-}ub}(\sigma_0, \alpha, p \cdot \tau^{p'}) = \sigma_1$ where $\sigma_1 = \sigma_0 \cup \{\alpha <: \tau^{p \cdot p'}\}$. Similarly, we introduce helper functions 'lb' and 'ub' to look up respectively the upper and lower bounds of a type variable in the type inference state.

C-Cache This rule allows to immediately solve a constraint if it has already been encountered in the constraint solving process, and is therefore contained in the set of hypotheses H. This is necessary to avoid divergence in the presence of recursive types.[4] We define $\mathrm{reset}(\tau_0, \tau_1)$ as the substitution, in (τ_0, τ_1), of all type provenances by ϵ. This is used to ensure that type provenances do not affect the memoization of the constraining function.

C-Refl A constraint between two equal types can be solved immediately. When we check for equality of types, we do not care for provenances. For this reason, we apply the reset function before we compare them.

C-Var-L (and similarly for C-Var-R) When we encounter a constraint $\alpha <: \tau$ between a unification variable α and a type τ (which is not a unification variable) we have to do two things. We first add τ to the set of upper bounds of α in the state σ. Then we generate and solve one additional constraint between τ and all existing lower bounds for α in σ.

C-Var-LR When we encounter a constraint $\alpha <: \alpha'$ between two unification variables, we add α to the lower bounds of α' and α' to the upper bounds of α before we generate the subconstraints to verify that the bounds are still consistent.

C-Sub When the constraint we have to solve is complex, i.e. neither of the two types is a unification variable, then we invoke the function sub in order to compute the subconstraints of the constraint. If this function returns with a set of new constraints, we solve them in turn.

[4]Whether the language is meant to support recursive types or not is an orthogonal concern, and it is easy to add a so-called "*occurs check*" to make sure recursive types are rejected, if needed.

$\boxed{\sigma \mid \Gamma \vdash e : \tau \mid \sigma}$

T-Unit
$$\frac{}{\sigma \mid \Gamma \vdash \mathtt{unit}^{\ell} : \mathbf{1}^{\ell} \mid \sigma}$$

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

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

T-Var
$$\frac{\sigma_0 \mid \Gamma(x) = \tau \mid \sigma_1}{\sigma_0 \mid \Gamma \vdash x^{\ell} : \tau^{\ell} \mid \sigma_1}\ x \in \mathsf{dom}(\Gamma)$$

T-Lam
$$\frac{\alpha \textit{ fresh} \qquad \sigma_0 \mid \Gamma \cdot (x : \alpha) \vdash e : \tau \mid \sigma_1}{\sigma_0 \mid \Gamma \vdash (\lambda x^{\ell_x}.\, e)^{\ell} : \alpha^{\ell_x} \to^{\ell} \tau \mid \sigma_1}$$

T-App
$$\frac{\alpha \textit{ fresh} \qquad \sigma_0 \mid \Gamma \vdash e_0 : \tau_0 \mid \sigma_1 \qquad \sigma_1 \mid \Gamma \vdash e_1 : \tau_1 \mid \sigma_2 \qquad \sigma_2 \mid \mathsf{cons}(\tau_0 <: \tau_1 \to \alpha^{\ell}) \mid \sigma_3}{\sigma_0 \mid \Gamma \vdash (e_0\ e_1)^{\ell} : \alpha^{\ell} \mid \sigma_3}$$

T-True
$$\frac{}{\sigma \mid \Gamma \vdash \mathtt{true}^{\ell} : \mathsf{Bool}^{\ell} \mid \sigma}$$

T-False
$$\frac{}{\sigma \mid \Gamma \vdash \mathtt{false}^{\ell} : \mathsf{Bool}^{\ell} \mid \sigma}$$

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

T-Prod
$$\frac{\sigma_0 \mid \Gamma \vdash e_0 : \tau_0 \mid \sigma_1 \qquad \sigma_1 \mid \Gamma \vdash e_1 : \tau_1 \mid \sigma_2}{\sigma_0 \mid \Gamma \vdash [e_0,\, e_1]^{\ell} : \tau_0 \otimes^{\ell} \tau_1 \mid \sigma_2}$$

T-LProj
$$\frac{\sigma_0 \mid \Gamma \vdash e : \tau \mid \sigma_1 \qquad \alpha, \beta \textit{ fresh} \qquad \sigma_1 \mid \mathsf{cons}(\tau <: \alpha^{\ell} \otimes^{\ell} \beta^{\epsilon}) \mid \sigma_2}{\sigma_0 \mid \Gamma \vdash \pi_1(e)^{\ell} : \alpha^{\ell} \mid \sigma_2}$$

T-RProj
$$\frac{\sigma_0 \mid \Gamma \vdash e : \tau \mid \sigma_1 \qquad \alpha, \beta \textit{ fresh} \qquad \sigma_1 \mid \mathsf{cons}(\tau <: \beta^{\epsilon} \otimes^{\ell} \alpha^{\ell}) \mid \sigma_2}{\sigma_0 \mid \Gamma \vdash \pi_2(e)^{\ell} : \alpha^{\ell} \mid \sigma_2}$$

T-LInj
$$\frac{\alpha \textit{ fresh} \qquad \sigma_0 \mid \Gamma \vdash e : \tau \mid \sigma_1}{\sigma_0 \mid \Gamma \vdash \iota_1(e)^{\ell} : \tau \oplus^{\ell} \alpha^{\epsilon} \mid \sigma_1}$$

T-RInj
$$\frac{\alpha \textit{ fresh} \qquad \sigma_0 \mid \Gamma \vdash e : \tau \mid \sigma_1}{\sigma_0 \mid \Gamma \vdash \iota_2(e)^{\ell} : \alpha^{\epsilon} \oplus^{\ell} \tau \mid \sigma_1}$$

T-Case
$$\frac{\begin{array}{c}\alpha, \beta, \gamma \textit{ fresh} \\ \sigma_0 \mid \Gamma \vdash e_0 : \tau_0 \mid \sigma_1 \qquad \sigma_1 \mid \Gamma \cdot (x^{\ell_x} : \alpha^{\ell_x}) \vdash e_1 : \tau_1 \mid \sigma_2 \qquad \sigma_2 \mid \Gamma \cdot (y^{\ell_y} : \beta^{\ell_y}) \vdash e_2 : \tau_2 \mid \sigma_3 \\ \sigma_3 \mid \mathsf{cons}(\tau_0 <: \alpha^{\ell_x} \oplus^{\ell} \beta^{\ell_y}) \mid \sigma_4 \qquad \sigma_4 \mid \mathsf{cons}(\tau_1 <: \gamma^{\ell}) \mid \sigma_5 \qquad \sigma_5 \mid \mathsf{cons}(\tau_2 <: \gamma^{\ell}) \mid \sigma_6\end{array}}{\sigma_0 \mid \Gamma \vdash \mathsf{case}\ e_0\ \mathsf{of}\ \{\, \iota_1(x^{\ell_x}) \Rightarrow e_1;\ \iota_2(y^{\ell_y}) \Rightarrow e_2 \,\}^{\ell} : \gamma^{\ell} \mid \sigma_6}$$

Fig. 9. Algorithmic type inference rules.

C-Error If the function sub returns with an error, the returned σ' is populated with err p elements containing the provenance chains p corresponding to the error.

3.3.3 Computation of Subconstraints. The computation of subconstraints is defined in Figure 11. The function sub(Q) takes a constraint Q as input, and either computes a new list of constraints to be solved, or otherwise returns an error err p containing a provenance if the constraint cannot be

$\boxed{\sigma \vdash \mathsf{cons}(Q)^H \dashv \sigma}$

$$\textsc{C-Cache}\ \frac{\mathsf{reset}(Q) \in H}{\sigma \vdash \mathsf{cons}(Q)^H \dashv \sigma} \qquad \textsc{C-Refl}\ \frac{\mathsf{reset}(\tau_0) = \mathsf{reset}(\tau_1)}{\sigma \vdash \mathsf{cons}(\tau_0 <: \tau_1)^H \dashv \sigma}$$

$$\textsc{C-Var-LR}\ \frac{\mathsf{add\text{-}lb}(\sigma_1, \alpha', p_1 \cdot \alpha^{p_0}) = \sigma_2 \qquad \begin{array}{c}\mathsf{add\text{-}ub}(\sigma_0, \alpha, \alpha'^{p_0} \cdot p_1) = \sigma_1 \\ \sigma_2 \vdash \mathsf{cons}([\tau_\alpha <: \alpha'^{p_1} \mid \tau'_\alpha \in \mathsf{lb}(\sigma_0, \alpha)])^{H \cup \mathsf{reset}(\alpha <: \alpha')} \dashv \sigma_3\end{array}}{\sigma_0 \vdash \mathsf{cons}(\alpha^{p_0} <: \alpha'^{p_1})^H \dashv \sigma_3}$$

$$\textsc{C-Var-L}\ \frac{\mathsf{add\text{-}ub}(\sigma_0, \alpha, \tau^{p_0} \cdot p_1) = \sigma_1 \qquad \sigma_1 \vdash \mathsf{cons}([\tau' <: \tau^{p_1} \mid \tau' \in \mathsf{lb}(\sigma_0, \alpha)])^{H \cup \mathsf{reset}(\alpha <: \tau)} \dashv \sigma_2}{\sigma_0 \vdash \mathsf{cons}(\alpha^{p_0} <: \tau^{p_1})^H \dashv \sigma_2}$$

$$\textsc{C-Var-R}\ \frac{\mathsf{add\text{-}lb}(\sigma_0, \alpha, p_0 \cdot \tau^{p_1}) = \sigma_1 \qquad \sigma_1 \vdash \mathsf{cons}([\tau^{p_0} <: \tau' \mid \tau' \in \mathsf{ub}(\sigma_0, \alpha)])^{H \cup \mathsf{reset}(\tau <: \alpha)} \dashv \sigma_1}{\sigma_0 \vdash \mathsf{cons}(\tau^{p_0} <: \alpha^{p_1})^H \dashv \sigma_2}$$

$$\textsc{C-Sub}\ \frac{\mathsf{sub}(Q) = \overline{Q} \qquad \sigma_0 \vdash \mathsf{cons}(\overline{Q}) \dashv \sigma_1}{\sigma_0 \vdash \mathsf{cons}(Q) \dashv \sigma_1} \qquad \textsc{C-Error}\ \frac{\mathsf{sub}(Q) = \mathsf{err}\ p}{\sigma \vdash \mathsf{cons}(Q) \dashv \sigma_0}$$

$\boxed{\sigma \vdash \mathsf{cons}(\overline{Q})^H \dashv \sigma}$

$$\textsc{C-Nil}\ \frac{}{\sigma \vdash \mathsf{cons}([]) \dashv \sigma} \qquad \textsc{C-Cons}\ \frac{\sigma_0 \vdash \mathsf{cons}(Q) \dashv \sigma_1 \qquad \sigma_1 \vdash \mathsf{cons}(\overline{Q}) \dashv \sigma_2}{\sigma_0 \vdash \mathsf{cons}(Q :: \overline{Q}) \dashv \sigma_2}$$

Fig. 10. Constraint solving algorithm.

solved. We return new subconstraints if the types in the constraint are either both function types, both product types or both sum types. In that case, we also have to recombine the provenances of the types which are involved in the constraint, in order to track how a data flow can be tracked through a constructor. This is where the additional provenances which we introduced, but didn't explain, in Section 3.2 come into play. We write $\langle p \rangle_L^{\odot}$ and $\langle p \rangle_R^{\odot}$ where the L and R indicate if the provenance comes from the left or right hand side of a constraint on a constructor type $\odot$. We use the notations $\tau^{p_0} \cdot p_1$ and $p_0 \cdot \tau^{p_1}$ as shorthands for $\tau^{p_0 \cdot p_1}$. In every other case, that is, if the outermost types of the two sides of a subtyping constraint are not identical, the constraint is not solvable.

When we compute the subconstraints of two function types, we use the function $\mathsf{rev}(p)$ on provenances which yields a type provenance with the same contents, but in reverse order. Reversal applies recursively, meaning that it also reverses the order of provenances nested inside constructors like $\langle p \rangle_L^{\rightarrow}$. The use of this function spells out a curious phenomenon: passing through a function parameter *reverses* the direction of a type flow, switching from *flowing into* to *flowing from*, or *vice versa*. To illustrate this subtlety, consider the following program annotated with the relevant locations $\ell_{1..8}$:

```
let foo f^ℓ1 = let g^ℓ2 = f^ℓ3 in g^ℓ4 "hello"^ℓ5
foo (fun x^ℓ6 -> x^ℓ7 + 1)^ℓ8
```

The problematic flow here is '$\ell_5 \cdot \langle \ell_4 \cdot \ell_2 \cdot \ell_3 \cdot \ell_1 \cdot \ell_8 \rangle_L^{\rightarrow} \cdot \ell_6 \cdot \ell_7$'. Here is how to understand it:

> *The string literal at ℓ_5 has type* `string`*; it flows* **into** *the* parameter *of function* `g` *at ℓ_4 (hence the L, which denotes the* left-hand-side *of a function type, so we reverse the flow direction); where* `g` *itself flows* **from** *let-bound identifier* `g` *at ℓ_2;* **from** *parameter reference* `f` *at ℓ_3;* **from** *parameter* `f` *at ℓ_1;* **from** *the argument function at ℓ_8; and (leaving the function type and reverting back to a forward flow)* **into** *parameter* `x` *at ℓ_6;* **into** *reference* `x` *at ℓ_7; where type* `int` *is expected.*

Notice how this flow starts as a normal *forward* flows but reverses to a *backward* one upon entering the left-hand side of a function type before going back to a forward flow as the flow leaves the function type. Naturally, the *complete* flow information described above is far too verbose to report directly to users. We found that a good tradeoff (which we use in our tool) is to only report the *outer* flow '$\ell_5 \cdot \ldots \cdot \ell_6 \cdot \ell_7$' and reserve the full flow for the verbose mode and (in the future) for interactive type error exploration and IDE integration.

$$\begin{aligned}
\mathsf{sub}(\tau_{00} \otimes^{p_0} \tau_{01} <: \tau_{10} \otimes^{p_1} \tau_{11}) &= [\tau_{00} \cdot \langle p_0 \cdot p_1 \rangle^{\otimes}_{L} <: \tau_{10},\ \tau_{01} \cdot \langle p_0 \cdot p_1 \rangle^{\otimes}_{R} <: \tau_{11}] \\
\mathsf{sub}(\tau_{00} \oplus^{p_0} \tau_{01} <: \tau_{10} \oplus^{p_1} \tau_{11}) &= [\tau_{00} \cdot \langle p_0 \cdot p_1 \rangle^{\oplus}_{L} <: \tau_{10},\ \tau_{01} \cdot \langle p_0 \cdot p_1 \rangle^{\oplus}_{R} <: \tau_{11}] \\
\mathsf{sub}(\tau_{00} \to^{p_0} \tau_{01} <: \tau_{10} \to^{p_1} \tau_{11}) &= [\tau_{10} \cdot \langle p_1 \cdot \mathsf{rev}(p_0) \rangle^{\to}_{L} <: \tau_{00},\ \tau_{01} \cdot \langle p_0 \cdot p_1 \rangle^{\to}_{R} <: \tau_{11}] \\
\mathsf{sub}(\tau_1^{p_0} <: \tau_2^{p_1}) &= \mathsf{err}\, p_0 \cdot p_1
\end{aligned}$$

Fig. 11. Subconstraining rules.

Finally, notice that all the type constructors used in this section are *variant*: product and sum types are covariant in their components and functions are contravariant in their parameters and covariant in their results. In a system with subtyping, it is always possible to separate the covariant and contravariant uses of type parameters, so that this pervasive variance is generally feasible. However, in ML languages like OCaml, some type constructors are defined as invariant, such as mutable references. To handle these, we need a notion of *non-directional unification*, which is studied in the next section.

4 TYPE CONFLUENCE ERRORS

The algorithm presented in Section 3 only recognizes and reports Level-0 errors. Now, we extend it to also report Level-n errors for $n \geq 1$, by tracking data flows described by constraints, as explained in Section 2.1.

$$\begin{array}{rl}
\textit{Provenance} & p ::= \ \ldots \mid Z \\
\textit{Relation} & \bullet ::= <:^{p} \mid :>^{p} \mid \sim^{\langle Z \rangle^{\otimes \mid \oplus \mid \to}_{L \mid R}} \\
\textit{Data flow} & Z ::= \ \tau \bullet \tau \mid Z \bullet \tau \mid \tau \bullet Z
\end{array}$$

Fig. 12. Extended syntax with unification.

Data flows. We extend the syntax from Figure 8 with the notion of data flows. A data flow Z is a sequence of types related by either $<:^{p}$, $:>^{p}$, or $\sim^{\langle Z \rangle^{\otimes \mid \oplus \mid \to}_{L \mid R}}$, and we abstract over these relations with the symbol $\bullet$. We already discussed in Section 2.1 that constraints represent data flows; we can therefore embed constraints into data flows: constraint $\tau^{p_0} <: \tau^{p_1}$, for example, corresponds to data flow $\tau <:^{p_0 \cdot p_1} \tau$.

If a data flow has the form $\tau_1 \bullet \ldots \bullet \tau_2$, then we use the shorthand $\tau_1 \_ \tau_2$ if we are only interested in the two outermost types of the data flow. Unifying a data flow $Z = \tau_1 \_ \tau_2$ says that τ_1 and τ_2 must be equal, and the data flow for this unification is explained by the components of Z. If the

types τ_1 and τ_2 are not equal, then we have to generate a detailed unification error from Z, which we will explain in Section 4.2.

We introduce $\sim$ to relate arguments of constructor types. We also introduce nested data flows $\langle Z \rangle_{L|R}^{\otimes|\oplus|\rightarrow}$ for arguments of constructor types. Consider the data flow $\tau_{00} \sim^{\langle Z \rangle_L^{\otimes}} \tau_{10}$, which has the nested data flow $Z = \tau_{00} \otimes \tau_{01} <: \alpha <: \tau_{10} \otimes \tau_{11}$. We say that the types τ_{00} and τ_{10} are the left arguments of product types at the terminal ends of data flow Z. A key intuition is that the types don't flow directly in Z but are carried by the product types. Similarly, sum and function type arguments have nested data flows too.

A data flow is *valid* for a given a type inference state if all the individual relations in it are valid. A sub-typing relation is valid if it is contained in the state σ, and the relation $\sim^{\langle Z \rangle_{L|R}^{\odot}}$ is valid if the nested data flow is valid and terminated by the correct constructor types.

$$
\begin{aligned}
\mathsf{valid}(Z)_\sigma &= \mathsf{valid}(\tau \bullet \tau')_\sigma \ \forall \tau, \tau'.\ (\ldots\ \tau \bullet \tau'\ \ldots) \in Z \\
\mathsf{valid}(\tau \sim^{\langle Z \rangle_L^{\odot}} \tau')_\sigma &= \mathsf{valid}(Z)_\sigma \text{ where } Z = (\tau \odot \_)\ \_\ (\tau' \odot \_) \\
\mathsf{valid}(\tau \sim^{\langle Z \rangle_R^{\odot}} \tau')_\sigma &= \mathsf{valid}(Z)_\sigma \text{ where } Z = (\_ \odot \tau)\ \_\ (\_ \odot \tau') \\
\mathsf{valid}(\tau <:^p \tau')_\sigma &= \tau <:^p \tau' \in \sigma \\
\mathsf{valid}(\tau :>^p \tau')_\sigma &= \tau' <:^{\mathsf{rev}(p)} \tau \in \sigma
\end{aligned}
$$

4.1 Unification Algorithm

The unification algorithm is specified in Figure 13. We start with function $\mathsf{uni}(\sigma)^H$, which takes a type inference state σ and recurses over its bounds through the unification function $\mathsf{uni}(Z)_\sigma^H$, where Z is a data flow and H is the current set of hypotheses. This function equates the first and last types of a data flow and terminates with an error if they are not equal. A piece of global state could be threaded through the inference rules describing this function to collect all incorrect data flows; however, we omit this to keep the algorithm's specification concise. It is enough to see that, given a derivation of this function, we can gather all unification errors by collecting all uses of the U-Error rule.

We say that type inference state σ is *saturated* when for all α and α' we have $\alpha \in \mathsf{lb}(\sigma, \alpha') \iff \alpha' \in \mathsf{ub}(\sigma, \alpha)$. Helper function 'saturate' saturates its input state in the obvious way.

U-state This rule is the entry point for unification. We first saturate the type inference state. Then, for each type variable α in state σ, we unify it with its upper and lower bound types τ.

U-Cache A unification is solved trivially if it is already cached. We use 'reset' to erase the provenance of the types before looking up the cache, where $\mathsf{reset}(\tau, \tau') = (\mathsf{reset}(\tau), \mathsf{reset}(\tau'))$. This cache is necessary for the same reason as the cache in Section 3.3.2, *i.e.,* to prevent divergence in the case of cyclic bounds. However, since unifying two types is a symmetric operation, we now look up both subtyping directions in the cache (*i.e.,* both $\mathsf{reset}(\tau, \tau')$ and $\mathsf{reset}(\tau', \tau)$).

U-Refl A unification between two types that are equal modulo their provenance is solved immediately. We use reset to erase the provenance information before comparing the types.

U-Var-L This rule unifies type variable α with τ. It produces a set of data flows from the the upper and lower bounds of α to τ. Since α is on the left of the relation, the new relations are concatenated to the left of the existing data flow Z. This preserves the left to right continuity of the data flow.

U-Var-R This is similar to U-Var-L except the type variable α is on the right of the relation. So the new relation gets concatenated on the right.

U-Sub When the unification involves constructed types (product, sum and function types), we invoke ctor-uni to compute sub-unifications for their type arguments. If the function cannot equate the constructor types it returns an error.

U-Error If U-Sub returns an error for an incorrect data flow, this rule terminates the algorithm. The actual algorithm can be implemented by threading through a state and collecting all such errors before reporting them.

$$\boxed{\mathsf{uni}(\sigma)^H}$$

$$\text{U-State}\quad \frac{\sigma' = \mathsf{saturate}(\sigma) \quad \mathsf{uni}([\tau <:^p \alpha \mid \forall \tau <:^p \alpha \in \mathsf{lb}(\sigma', \alpha)])^H_{\sigma'} \quad \mathsf{uni}([\alpha <:^p \tau \mid \forall \alpha <:^p \tau \in \mathsf{ub}(\sigma', \alpha)])^H_{\sigma'}}{\mathsf{uni}(\sigma)^H}$$

$$\boxed{\mathsf{uni}(\overline{Z})^H_\sigma}$$

$$\text{U-Nil}\quad \frac{}{\mathsf{uni}([])^H_\sigma} \qquad \text{U-Cons}\quad \frac{\mathsf{uni}(Z)^H_\sigma \qquad \mathsf{uni}(\overline{Z})^H_\sigma}{\mathsf{uni}(Z :: \overline{Z})^H_\sigma}$$

$$\boxed{\mathsf{uni}(Z)^H_\sigma}$$

$$\text{U-Cache}\quad \frac{\{\mathsf{reset}(\tau, \tau'), \mathsf{reset}(\tau', \tau)\} \cap H \neq \emptyset}{\mathsf{uni}(\tau \_ \tau')^H_\sigma} \qquad \text{U-Refl}\quad \frac{\mathsf{reset}(\tau) = \mathsf{reset}(\tau')}{\mathsf{uni}(\tau \_ \tau')^H_\sigma}$$

$$\text{U-Var-L}\quad \frac{\mathsf{uni}([\tau' <:^p \alpha \_ \tau \mid \tau'^p \in \mathsf{lb}(\sigma, \alpha)])^{H \cup \mathsf{reset}(\alpha,\tau)}_\sigma \qquad \mathsf{uni}([\tau' :>^{\mathsf{rev}(p)} \alpha \_ \tau \mid \tau'^p \in \mathsf{ub}(\sigma, \alpha)])^{H \cup \mathsf{reset}(\alpha,\tau)}_\sigma}{\mathsf{uni}(\alpha \_ \tau)^H_\sigma}$$

$$\text{U-Var-R}\quad \frac{\mathsf{uni}([\tau \_ \alpha :>^{\mathsf{rev}(p)} \tau' \mid \tau'^p \in \mathsf{lb}(\sigma, \alpha)])^{H \cup \mathsf{reset}(\alpha,\tau)}_\sigma \qquad \mathsf{uni}([\tau \_ \alpha <:^p \tau' \mid \tau'^p \in \mathsf{ub}(\sigma, \alpha)])^{H \cup \mathsf{reset}(\alpha,\tau)}_\sigma}{\mathsf{uni}(\tau \_ \alpha)^H_\sigma}$$

$$\text{U-Sub}\quad \frac{\mathsf{ctor\text{-}uni}(\tau \_ \tau')^H_\sigma = \overline{Z} \qquad \mathsf{uni}(\overline{Z})^{H \cup \mathsf{reset}(\tau,\tau')}_\sigma}{\mathsf{uni}(\tau \_ \tau')^H_\sigma} \qquad \text{U-Error}\quad \frac{\mathsf{ctor\text{-}uni}(Z)^H_\sigma = \mathbf{err}\ Z}{\mathsf{uni}(Z)^H_\sigma}$$

$$\mathsf{ctor\text{-}uni}(Z)^H_\sigma = [\tau_{00} \sim^{\langle Z \rangle^\otimes_L} \tau_{10},\ \tau_{01} \sim^{\langle Z \rangle^\otimes_R} \tau_{11}] \quad \textit{where} \quad Z = \tau_{00} \otimes \tau_{01} \_ \tau_{10} \otimes \tau_{11}$$

$$\mathsf{ctor\text{-}uni}(Z)^H_\sigma = [\tau_{00} \sim^{\langle Z \rangle^\oplus_L} \tau_{10},\ \tau_{01} \sim^{\langle Z \rangle^\oplus_R} \tau_{11}] \quad \textit{where} \quad Z = \tau_{00} \oplus \tau_{01} \_ \tau_{10} \oplus \tau_{11}$$

$$\mathsf{ctor\text{-}uni}(Z)^H_\sigma = [\tau_{00} \sim^{\langle Z \rangle^\rightarrow_L} \tau_{10},\ \tau_{01} \sim^{\langle Z \rangle^\rightarrow_R} \tau_{11}] \quad \textit{where} \quad Z = \tau_{00} \rightarrow \tau_{01} \_ \tau_{10} \rightarrow \tau_{11}$$

$$\mathsf{ctor\text{-}uni}(Z)^H_\sigma = \mathbf{err}\ Z \quad \textit{where} \quad Z = \tau \_ \tau'$$

Fig. 13. Data-flow-tracking unification.

Notice that because we look up *both* pairs (τ, τ') and (τ', τ) in the cache, we will only ever traverse a given data flow between two variables in a single direction only, instead of potentially traversing the data flow once in each direction, which could otherwise happen in the presence of type variable cycles, such as $\alpha <: \beta, \beta <: \alpha$. Long cyclic chains could lead to a potentially exponential amount of erroneous data flows, which we avoid this way. Moreover, our concrete implementation performs a breadth-first search while following the algorithm of Figure 13 in order to find the shortest erroneous paths, and then stops the search without traversing further constraints. From our own observations, this seems to make inference fast in practice.[5] We only report one erroneous path to the programmer to explain a given error, even when many possible paths (including several shortest ones) are available. This choice can be considered *arbitrary*, as there is no reason to consider that one particular path should be better at explaining the error than the others. In the future, different approaches to report such equivalent paths could be studied: We could report all paths,

[5]This can be observed in the web demo, which updates the output quickly upon every keystroke, even for larger programs.

explain cyclic data flows separately to programmers, or use heuristics to determine which path will be more understandable.

4.2 Detailed Error Reports

We create error reports from the data flows of failed unifications. Each data flow is transformed into a sequence of source locations separated by alternating data flow directions, encoding the back-and-forth nature of higher-level error, and possibly containing nested flows when the problematic unification is indirect and goes through type constructor arguments. Additional type information in the data flow is used to add helpful details to the report. The data flow information can be used for interactive error reporting, interactive code exploration, code hints in IDEs to name a few. Our current implementation only reports errors as we explain below.

Example 4.1 (Level-0 Data Flow). We can visualize a data flow as a type flowing through a sequence of program locations. A constraint is the most basic data flow. We can translate the subtyping constraint $\alpha^{p_0} <: \tau^{p_1}$ into the data flow $\alpha <:^{p_0 \cdot p_1} \tau$. After unification, τ and α must be the same type, so we can say that the data represented by the type flows through the locations p_0 and p_1. This can be visualized by the following diagram:

$$\alpha \xrightarrow{\quad p_0 \quad p_1 \quad} \tau$$

Example 4.2 (Level-1 Data Flow). We annotate the second program from Figure 3b, which yields the following annotated program:

```
let x^ℓ0 = 2^ℓ1
let y^ℓ2 = (if true then x^ℓ4 else "x"^ℓ5)^ℓ3
```

For this program we generate the constraints $\mathsf{Int}^{\ell_4 \cdot \ell_0 \cdot \ell_1} <: \alpha_y^{\ell_3 \cdot \ell_2}$ and $\mathsf{Str}^{\ell_5} <: \alpha_y^{\ell_3 \cdot \ell_2}$. We get a unification error for $\mathsf{Int} <:^{p_0} \alpha_y :>^{p_1} \mathsf{Str}$ where $p_0 = \ell_1 \cdot \ell_0 \cdot \ell_4 \cdot \ell_3 \cdot \ell_2$ and $p_1 = \ell_2 \cdot \ell_3 \cdot \ell_5$. Notice that invoking symmetry for Str relation to creates a linear data flow. This flow looks like this:

$$\mathsf{Int} \xrightarrow{\quad \ell_1 \quad \ell_0 \quad \ell_4 \quad \ell_3 \quad} \alpha_y \xleftarrow{\quad \ell_3 \quad \ell_5 \quad} \mathsf{Str}$$

Example 4.3 (Level-2 Data Flow). Similarly, we annotate the locations for the program in Figure 3c which yields the following annotated program:

```
let g x^ℓ1 = (not x^ℓ2, (if true then x^ℓ4 else 5^ℓ5)^ℓ3)
```

We generate the following constraints and unification error for α_x and α_{ite}[6]. The constraints are $\mathsf{Int}^{\ell_5} <: \alpha_{ite}^{\ell_3}$, $\alpha_x^{\ell_4 \cdot \ell_1} <: \alpha_{ite}^{\ell_3}$ and $\alpha_x^{\ell_1} <: \mathsf{Bool}^{\ell_2}$. We visualized the unification error $\mathsf{Int} <:^{\ell_5 \cdot \ell_3} \alpha_{ite} :>^{\ell_3 \cdot \ell_4 \cdot \ell_1} \alpha_x <:^{\ell_1 \cdot \ell_2} \mathsf{Bool}$ in the following way:

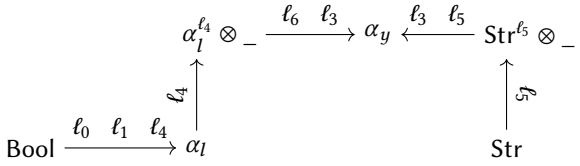
$$\mathsf{Int} \xrightarrow{\quad \ell_5 \quad \ell_3 \quad} \alpha_{ite} \xleftarrow{\quad \ell_3 \quad \ell_4 \quad \ell_1 \quad} \alpha_x \xrightarrow{\quad \ell_1 \quad \ell_2 \quad} \mathsf{Bool}$$

Example 4.4 (Constructor Data Flow). We introduce the following program to demonstrate a nested data flow:

```
let x^ℓ0 = true^ℓ1
let y^ℓ2 = (if true then (x^ℓ4, "true")^ℓ6 else ("false"^ℓ5,"false")^ℓ7)^ℓ3
```

We consider constraints $(\alpha_l^{\ell_4} \otimes^{\ell_6} \_) <: \alpha_y^{\ell_3 \cdot \ell_2}$, $(\mathsf{Str}^{\ell_5} \otimes^{\ell_7} \_) <: \alpha_y^{\ell_3 \cdot \ell_2}$, $\mathsf{Bool}^{\ell_1 \cdot \ell_0} <: \alpha_l^{\ell_4}$. Unification gives an error for $\mathsf{Bool} <:^{\ell_1 \cdot \ell_0 \cdot \ell_4} \alpha_l \sim^{\langle Z \rangle_L^{\otimes}} \mathsf{Str}$ where $Z = (\alpha_l^{\ell_4} \otimes \_) <:^{\ell_6 \cdot \ell_3 \cdot \ell_2} \alpha_y :>^{\ell_2 \cdot \ell_3 \cdot \ell_7} (\mathsf{Str}^{\ell_5} \otimes \_) <: \alpha_y$. We elide the right type argument and irrelevant provenances for clarity. It is visualized in the following ~~diagram:~~

[6] α_{ite} is the type for the if-then-else expression.



Converting this representation into a textual error message is straightforward. The sequence of program locations are shown vertically betwe en lines describing intermediate types. The constructor data flow is shown with a horizontal offset corresponding to the height of the nested data flow in the diagram. Our layout is not prescriptive and future implementations can experiment with other layouts.

Notice that in this section, we now unify constructor arguments using the non-directional symbol ~ instead of the directional <:. This is appropriate because some type constructors can be invariant in OCaml, and even variant constructors use unification semantics during type inference anyway. But since product, sum, function, and other types can still be considered variant even in ML (because their type parameters are used only at one polarity in their definitions), this can still be used to construct properly directional flows in type error explanations. For instance, notice that in the diagram above, we use directional arrows in all the edges of the graph, because $\otimes$ is covariant in its first argument. We would use a non-directional edge if the type constructor was invariant, for example, if it had been a mutable reference. We can reconstruct the directionality of these variant flows by inspecting the nature of the type constructor in the nested $\sim^{\langle Z \rangle_{L|R}^{\otimes|\oplus|\rightarrow}}$ unification forms.

We also include a one-line flow summary (or *outline*) in the survey error messages. It was omitted from the introductory examples as it is not essential. At a glance, it shows the user a high-level overview of the erroneous flow by stripping away the location info and using ASCII symbols to show the flow direction. Figure 14 shows flow summaries for the examples discussed above.

```
(?a) ---> (τ)
```

(a) Example 4.1

```
(int) ---> (?a) <--- (str)
```

(b) Example 4.2

```
(int) ---> (?a) <--- (?b) <--- (str)
```

(c) Example 4.3

```
(bool) ---> (?a) ~~~> (?a * _) ---> (?b) <--- (str * _) <~~~ (str)
```

(d) Example 4.4

Fig. 14. Flow summary for data flows shown in Section 4.2

5 EXPERIMENTAL EVALUATION

In this paper, we hypothesize that flow-based type error messages are more effective than location-based type error messages in helping programmers understand errors. Traditional error messages only provide *one* of the possible locations related to each error, which can be useful to understand *where* a program goes wrong but is often insufficient to understand the full context of the error. In contrast, our proposed system presents the flow of types, which intends to help understand *how* a program is goes wrong. Traditional systems thus focus on locality, while we attempt to facilitate causal reasoning, which we believe is essential to effectively understand and repair errors.

We conduct a randomized quasi-experimental study to validate our hypothesis. The experiment compares the understanding of programmers using flow-based type error messages with those using traditional location-based error messages.

In the user study, we ask participants to understand and describe program errors, measuring *a)* the perceived satisfaction of the participants with the provided error messages and *b)* whether the participant sufficiently understood the error.

5.1 Experiment Design

The experiment has been conducted in form of an online survey (using lab.js [Henninger et al. 2021] and hosted on Open Lab [Shevchenko 2022]). After a demographic questionaire and a short introduction to OCaml, participants were presented with a series of errornous OCaml programs. Provided with the program and an error message (side-to-side), participants were invited to understand the error using the provided error message followed by a series of questions:

Q1 "In your own words: what is the problem in the program above?"
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"

The first question (Q1) was asked as free form text asking participants to keep the answer short. The remaining two questions (Q2 and Q3) were answered on a five-point Likert scale ranging from "Not helpful" to "Very helpful". The survey ended with a single optional free form text field asking users "Is there anything you want to tell us?".

Conditions. When starting the survey, each participant was randomly assigned (drawn without replacement) to one of three conditions:

(A) *HM$^\ell$* – our implementation of HM$^\ell$, extending SystemSub with flow-based type errors.
(B) *OCaml* – as a first control group, we compare against the standard OCaml compiler.
(C) *Helium* – as a second control group, we compare against Helium [Heeren et al. 2003].

Helium is a compiler for Haskell, not OCaml, but the subset of programs we consider can be easily translated from OCaml to Haskell. We therefore translated the OCaml examples to Haskell by hand, used Helium to generate an error message, and translated the types contained in the resulting error message back to use OCaml Syntax.

Selection of programs. We prepared ten different ill-typed example programs that we manually ranked as "easy" (3), "medium" (4), or "hard" (3). Programs labeled as "easy" were constructed specifically for the study, while examples labeled "medium" or "hard" were selected from the datasets shared by [Seidel et al. 2017]. From each of the three categories, each participant was presented with two randomly sampled example programs; that is, six programs in total. While the order of examples within each category was random; the categories itself have always been presented in ascending complexity.

5.2 Participants

We shared the online survey with professionals and researchers by posting it on relevant online platforms (such as Reddit and Twitter). Participants were asked to self-estimate their experience in programming in general, functional programming, statically typed programming, and programming in OCaml on a 5-point Likert scale [Feigenspan et al. 2012]. From a total of 455 participants, 318 started and 119 concluded the survey (40 HM$^\ell$, 39 OCaml, 40 Helium). We manually excluded two participants (both in the OCaml condition) since they visibly did not invest enough effort to answer the questions. Of the 117 non-excluded participants, 70 assigned themselves an expertise ≥ 4 for

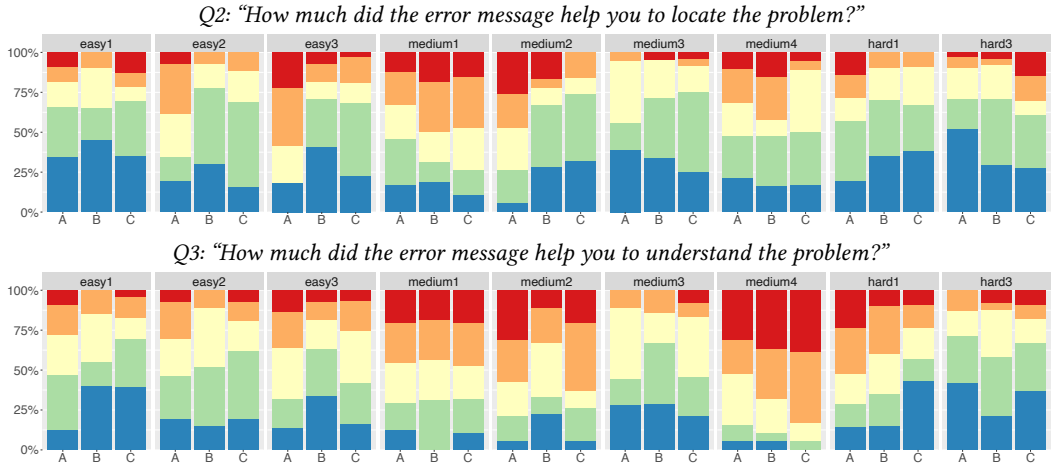


Fig. 15. Participants answering the respective question on a five-point Likert scale from "Not helpful" (top, red) to "Very helpful" (bottom, blue). We compare conditions HM$^{\ell}$ (A), OCaml (B), and Helium (C).

"functional programming" or "OCaml". Only 14 participants did assign themselves an expertise ≤ 3 for all categories.

5.3 Evaluation

The study sets out to analyse whether the error-message condition (HM$^{\ell}$, OCaml, or Helium) significantly influences *a)* the perceived usefulness of the error messages, as well as *b)* the understanding of the programming error. We first discuss the evaluation of perceived usefulness (Q2 and Q3) before discussing the evaluation of the open question (Q1).

After data collection, we realized a mistake made while hand-translating Helium error message for `hard2` back to OCaml types. It left a Haskell formatted list type, which might have been confusing to participants and led to poor responses. Thus we exclude example `hard2` from our evaluation.

5.3.1 Perceived Usefulness (Q2 and Q3). For each of the ten individual example programs, and each of the three conditions (A), (B), and (C), Figure 15 presents the results for the perceived usefulness reported by participants. The five-point Likert-scaled data is visualized as stacked bar charts, where the lowest (dark-blue) component corresponds to the answer "Very helpful", and the highest (dark-red) component corresponds to the answer "Not helpful".

Locating the problem (Q2). Performing a Kruskal-Wallis rank sum test [Kruskal and Wallis 1952] for each of the ten tasks, we can find significant differences for `easy2` ($p = 0.0119$, $\chi^2 = 8.87$), `easy3` ($p = 0.0025$, $\chi^2 = 11.96$), `medium2` ($p = 0.0067$, $\chi^2 = 10.015$). Table 1 lists the full results for all performed Kruskal-Wallis rank sum tests. All participants correctly described the error for program `easy1`.

To determine which groups are different, we perform a post-hoc Dunn test [Dunn 1964] for each of the significant example programs. A Bonferonni adjustment of the p-value is used to account for the error rate introduced by the pair-wise comparison. For `easy2`, we have significant differences for $A-B$ with -2.934363 ($p = 0.0050$). That is, participants reported messages of HM$^{\ell}$ to help less with locating than those of OCaml. For `easy3`, we have significant differences for $A-B$ with -3.2893 ($p = 0.0015$) and $A-C$ with -2.7449 ($p = 0.0091$). That is, participants reported messages of HM$^{\ell}$ to help less with locating than those of OCaml or Helium. For `medium2`, we have significant

Table 1. Results of all Kruskal-Wallis tests per program.

Program	Q1 Describe p	Q1 Describe χ^2	Q2 Locate p	Q2 Locate χ^2	Q3 Understand p	Q3 Understand χ^2
`easy1`	N/A	N/A	0.7748	0.51	0.0805	5.04
`easy2`	0.8808	0.25	0.0119	8.87	0.6259	0.94
`easy3`	0.3015	2.40	0.0025	11.95	0.0860	4.91
`medium1`	0.8943	0.22	0.5009	1.38	0.9988	0.00
`medium2`	0.7996	0.45	0.0067	10.02	0.1727	3.51
`medium3`	0.4535	1.58	0.9386	0.13	0.5235	1.29
`medium4`	0.9042	0.20	0.6442	0.88	0.3943	1.86
`hard1`	0.8190	0.40	0.2410	2.85	0.0632	5.52
`hard3`	0.4928	1.42	0.1257	4.15	0.3650	2.02

Q1: "In you own words: what is the problem in the program above?"

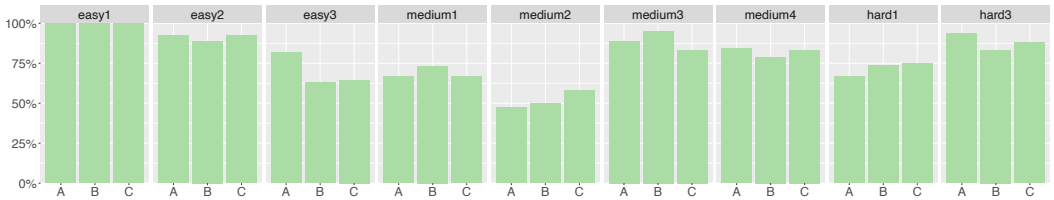


Fig. 16. Percentage of participants that correctly described the problem. We compare conditions HM$^{\ell}$ (A), OCaml (B), and Helium (C).

differences $A-C$ with -3.018982 ($p = 0.0038$). That is, participants reported messages of HM$^{\ell}$ to help less with locating than those of Helium.

Understanding the problem (Q3). Similar to Q2, we perform a Kruskal-Wallis test, however there is no significant difference between the results of the three systems.

5.3.2 Understanding Errors (Q1). To evaluate the open question Q1, we manually assigned a binary grade to the provided textual answers, judging whether the participant understood the underlying problem or not. Emphasis was given on participants' understanding of erroneous program expressions and the fixes they suggested. To avoid biases, the manual coding was performed blind—that is, the condition of a participant during evaluation was hidden to the grader. The same grader graded the responses from all participants. Figure 16 reports the results for the three conditions, again grouped per example program.

5.4 Interpretation

Our experiment yielded little statistically significant difference between respondents using different tools to locate and understand errors. In general, the study cannot show that HM$^{\ell}$ improves understanding or localization of problems.

In a few cases, other systems even performed significantly better than HM$^{\ell}$. For Q2, which measured the perceived value of the error message in locating the error, `easy2`, `easy3`, and `medium2` showed significant results. In all three cases, participants reported that HM$^{\ell}$ error messages helped less in locating the problem. All three programs (especially `easy2` and `easy3`) are comparatively small. Constructing (and consuming) a detailed data flow explanation that is longer than the programs might not pay off in complexity. For Q3, which measured the perceived value of the error message in understanding the error, no results were significant.

A qualitative analysis of the freeform feedback provided by participants reveals interesting insight into why HM^ℓ might have performed worse than OCaml and Helium in *locating* errors.

Errors are too verbose or unnecessary for small programs. Many participants reported the HM^ℓ errors to be excessively verbose, which was detrimental to understanding errors for *small* programs.

> *The earliest examples were so trivial I could've figured them out without error messages.*

However, some participants also reported that they were indeed useful for longer problems.

> *I felt that the error messages were more helpful on the longer problems, only because I didn't need to look at them on the shorter problems.*

The problem of errors that are too verbose could be remedied by designing better layouts or using heuristics to hide some of the extra locations for simple programs. Furthering the previous point, in many of the simple cases (in *easy* and *medium* programs), the respondents were frustrated by the comparatively large number of locations reported by HM^ℓ and instead just wanted to see *"the" location where the error happened.* We conjecture that it would be helpful to combine our work with previous research on identifying single error locations and provide an "*incipit*" of sorts before the precise data flow—reporting a single location, which is often enough to understand simple errors. This way, programmers would only look at the precise data flow when they need to obtain more context and understanding for the root causes of the error. In an interactive setting the additional information provided by flow errors could be presented on request of the user. While the participant's concerns are important and it would be interesting to investigate more compact representations, real-world programs are almost always larger than the programs surveyed in our study.

Error message layout and notation. Respondents also complained about the error message layout as well as other aspects of the used notation, including the data-flow arrows, line number formatting, ellipsis for large code and so on.

> *Many of the error messages I was shown seemed upside-down, in that they showed what I believe to be the source of the error at the bottom, with a stack of more-removed sources of incompatible constraints extending upwards. This might look better coming from a CLI, but in this format it was weird and unhelpful.*

However, other partipants appreciated the structure of HM^ℓ's error messages:

> *In general, I really like the detail and consistency in the error messages. This really helps with solving more subtle errors, but it also adds a lot of noise for more simple issues. I feel like this isn't a real problem since even a little experience will allow you to immediately identify the issues at a glance of the error message by looking at the right things. One possible improvement could be to mention (and preview) the offending statement with the relevant parts marked before going in depth on the breakdown of the type interpretation. As it is now, one part of the breakdown hides the other conflicting part of the statement*

There is ample opportunity to improve all aspects of the presentation of error messages, including the data-flow summaries, the textual explanation, and the graphical layout. For each of the components, the usefulness is still unclear, the concrete design can be improved, and the interplay between the different aspects needs to be studied. More research and testing is needed to develop effective error message layouts for data-flow style reasoning.

One factor that might have contributed to the confusion of participants is that many of our respondents were experienced practitioners who were accustomed to message layout from standard tools. We also conjecture that experienced programmers already developed a deep understanding for *how* type-checking in OCaml proceeds and thus learnt to infer information from existing error

messages. It naturally requires some time to adapt to the new format of error messages, as also one participant described:

> *The error message has all the information, it takes some time to get used to it though.*

Similarly, many respondents found the unification variable shown in HM$^\ell$ messages unhelpful. The unexplained "`?a`" notation used for unification variables combined with the new concept of data flows was confusing to these participants. We conjecture that a gentle introduction to HM$^\ell$'s notation, as well as familiarity built over time, could potentially remedy these issues.

Flow-based reasoning. The work presented in this paper builds upon the assumption that the underlying mental model (also called "notional machine" by du Boulay et al. [1981]) of flow-based reasoning is natural and can help programmers to understand and locate error messages. A possible interpretation of the below user feedback is that this assumption is false.

> *I honestly find the '`int --> ?a <-- bool`' notation quite confusing. It is useful in some cases where there is no obvious expected or actual type, but in cases where the unification variable is unnecessary, it adds quite a bit of unnecessary mental overhead.*

Maybe understanding how values (and types) flow through a program does not contribute to the understanding of type errors. Maybe guiding users along the data flow is not helpful afterall, since they could also follow the data flow themselves without the overhead of processing verbose error messages with positions marked by unification variables that are not part of the original program text. Our work aims to *support* users in this process, which only makes sense if the process itself is practically useful.

No specialized messages. OCaml and Helium had specialized messages for certain errors. Messages like "function applied to too many arguments" (`hard1`) and "expected type …" seem more helpful than just data flow information. Of course, HM$^\ell$ could be specialized to also add such helpful text to error messages, which are orthogonal to the idea of presenting data flow.

5.4.1 The Need for Detailed Explanations. While some participants remarked that the messages of HM$^\ell$ are too verbose, those in the control groups often remarked the opposite about messages generated by OCaml and Helium.

> *I think in many of the examples it would be helpful for the errors to explain where the constraint was introduced that we are hitting [...].*

Another participant recognized the difficulty of extensive inference:

> *Presumably some of the harder ones were caused by extensive inference throughout the code. It might help to show and look at multiple errors in such cases. Perhaps there's a way to group them, but this is hypothetical.*

Yet another participant would like to see more type information of the different components that constitute a problematic call.

> *The error messages are not at all clear about what the expected and what the found type is. Also, it is not clear why it believes the found type is the found type. Maybe it could show the different elements that are applied to the function separately as well?*

Participants were also aware of the difficulty to trade-off concise error messages and sufficiently detailed information.

> *Almost every type error in a program is an accumulation of multiple (smaller) erroneous parts. Any such error that states "something happened exactly here and here" is incomplete, because there's far more context that should be included to fully understand the error. Some languages show the entire chain of type unification that led to the error, but that's*

rather verbose. I hope one can find a solution that shows just enough context to be perfectly helpful.

The above quotes are only a few examples of the feedback that we received by the control groups. In many cases, respondents want to know source expressions for the two conflicting types, instead of the singular location where the error was triggered. They also want to see surrounding source code in the error message to gain context.

HM^{ℓ}'s error messages address exactly these two concerns. We interpret this as support our hypothesis that data-flow-style error messages could be useful for programmers.

Overall, the quantitative analysis cannot show that HM^{ℓ} improves over `ocamlc` or Helium. The qualitative analysis suggests that this might be due to the error message layout and notations or the verbosity. However, some respondents found the verbosity and context helpful as well, particularly for large programs and subtle errors. HM^{ℓ}'s contribution is the detailed data flow information, not the specific error message layout. However, HM^{ℓ} in its current form is certainly not perfect and respondents point out a few shortcomings, such as the exact textual representation of errors. We believe, in future work can well utilize it to design better error message layouts, code exploration tools, and IDE intellisense features.

5.5 Threats to Validity

Internal validity. For most of the tasks, the results were insignificant. Potentially, programs in the used corpus were not complex enough to measure differences between the systems. The study was conducted remotely with no control over the context. Participants might have been distracted while answering, spent different amount of time (mean = 32min, sd = 22min), and have used different devices. We used the browser's *user agent* to identify mobile devices. We were initially concerned that mobile users may have a harder time taking the survey and may subsequently provide a worse quality of responses, but all open-text answers provided by the 14 participants on mobile devices were of a high quality, so we decided to keep them. Participants might not know enough OCaml to understand the errors, which we tried to address by presenting each participant with a one-page introduction into the relevant OCaml concepts. Participants might recognize the different styles of error messages from different systems. We tried to limit the influence of this bias by fixing one participant to one specific condition. However, experienced OCaml programmers reportedly recognized the errors generated by Helium and our system to be non-standard. Participants might be biased in favour of our system, since it is socially desirable: firstly, it is always interesting to see a new tool that could turn out an improvement and secondly, because participants might be fellow researchers and practitioners that want to support research in the field. Participants were not trained on how to read HM^{ℓ} error messages. Unknown notation and unconventional layout of error messages might have confused participants.

External validity. Our example programs were sourced from university students solving programming assignments [Seidel et al. 2017]. These may not be indicative of the broader range of programming practices prevalent at large. The results of the study might not carry over to languages other than OCaml. Our approach can be used with all languages that implement Hindley-Milner-style type inference. However, we chose OCaml because of extensive empirical analysis already done on OCaml error localization by other researchers [Geng et al. 2022; Zhang and Myers 2014] and because of the relative simplicity of the language's base constructs. For instance, compared to Haskell, the lack of type classes in OCaml makes it easier to quickly introduce the language to beginner and novice programmer before starting the survey. Conveniently, previous research made available large datasets of ill-typed OCaml programs ranging from low to high complexity.

6 RELATED WORK

Algebraic subtyping. Type inference for systems with subtyping and parametric polymorphism is a known hard problem. We build upon the algebraic subtyping approach developed in [Dolan 2017] and [Dolan and Mycroft 2017]. More precisely, we build upon the more recent publications [Parreaux 2020] and [Parreaux and Chau 2022].

Explaining type errors with data flow. The approach to explaining type errors using the data flow of the program is very reminiscent of the similar approach by Gast [2005]. It describes an algebra based on subtyping constraints and defines a "consistency" relation between types. This is similar to how we unify the bounds of type variables to find flow errors where incompatible types flow into or flow from the same type variable. Neubauer and Thiemann [2003] use sum types to encode all the types an expression can be and flows sets to track the locations each type can flow through. Our work builds on this by formally categorizing different kinds of data flows and describing a systematic approach to display error reports for them. We also provide an implementation of our algorithm that integrates with existing type systems and supports let polymorphism.

Algorithmic error localization. Previous work on type errors focus on finding the program expression most likely causing the error. Zhang and Myers [2014] demonstrate a constraint based system to identify expressions that create unsatisfiable constraints. Using heuristics they pick the simplest explanation for the error. Loncaric et al. [2016] also demonstrate a constraint based system that can integrate with existing type systems to produce error reports efficiently. Our system directly integrates provenance tracking with constraint solving allowing us to track detailed information. Heuristics based error localization can complement detailed data flow errors.

Data driven error localization. More recent work by Geng et al. [2022]; Seidel et al. [2017] leverages language models and supervised learning techniques to localize errors. They use large datasets with pairs of ill-typed and fixed programs to train models, which can then predict the likely location for the fix with high accuracy. However these techniques are limited to identifying a program expression and cannot create error messages which explain the flow that causes the error.

Improving compiler error messages. There's been considerable research on type errors messages [Heeren 2005] and their role in programmer experience. Becker et al. [2019] mention that type error messages play an important role in helping the programmer fix the error. Marceau et al. [2011a] argue that reporting all type errors and mapping error messages back to source code are crucial for effective error messages. Furthermore, Marceau et al. [2011b] recommends not to highlight specific fixes as they may be incorrect. Techniques from Wrenn and Krishnamurthi [2017] can be used to evaluate and improve data flow style error messages. Finally, Kochhar et al. [2016] surveyed software engineering practitioners to find that respondents prefer general solutions that can integrate with existing tooling and IDEs, furthermore they should scale to large codebases. Our system addresses the key challenge of mapping an error back to source code locations and existing tools and IDEs can be instrumented with the detailed data provenance information for interactive debugging.

7 CONCLUSION AND FUTURE WORK

We now conclude and suggest directions for future work.

7.1 Conclusion

If we want powerful type inference techniques to become broadly accepted in mainstream programming languages, we have to generate excellent error messages when type inference goes wrong. In

this article, we laid some foundations to improve one important class of error messages: type error messages arising from constraint solving for both subtyping and equality constraints. Our main insight is that these constraints contain information about the data flow that led to the error, and that we can use this information to generate more informative error messages. We carried out a user study and compared our error messages to those of `ocamlc` and Helium. The study suggests that the additional information can potentially be overwhelming, so we have to carefully consider under what circumstances we use it and and how much of it we present to the user. While the empirical part of the study could not quantitatively show that HM^ℓ improved over the state of the art, we also received encouraging feedback by participants which suggest that the general approach of flow-based error messages is worthwhile.

7.2 Future Work

We see two important directions of future work related to the extension of our method to model additional features of common type systems.

Let polymorphism. The formalism that we presented does not include let bindings which are polymorphically generalized, even though this is a standard feature of both the Hindley-Damas-Milner algorithms and algebraic subtyping. Our implementation supports both *top-level* and *local* let-polymorphism. We are yet to investigate its formalization, although we do not expect any particular difficulty. However, authors such as Vytiniotis et al. [2010] argue that local let bindings usually need not be generalized.

Occurs check. We have not implemented the *occurs check* in our prototype yet. The occurs check is a standard feature of Hindley-Milner type inference that catches cycles in constraint graphs. One of the most significant results in our study is that occurs-check failures are much harder to understand for users than unification failures, and so there is much space for improvement. We hope to significantly improve these error messages using our approach based on data flows. Running the occurs-check separately also has algorithmic complexity advantages [Rémy 1992].

More advanced type system features. We would like to investigate how flow-based reasoning scales to more advanced type system features, where type error messages can often become even more confusing than traditional unification error. For example, we are particularly interested in studying support for higher-rank and first-class polymorphism [Jones et al. 2007], especially since these approaches could benefit from subtyping [Le Botlan and Rémy 2014]. Other important and tricky type system features include generalized algebraic data types, constrained types, modalities, and linear types.

ACKNOWLEDGMENTS

We want to thank the anonymous reviewers for their comments and for their help in improving the paper. We would also like to thank Volker Franz for feedback on the study design and Marlen Brachthäuser for help with the study design and the empirical evaluation. Jiří Beneš contributed the idea to replicate the flow overview in a gutter on the left of our error messages.

DATA-AVAILABILITY STATEMENT

The implementation of the system HM^ℓ described in this paper is permanently available on Zenodo [Bhanuka et al. 2023]. The latest implementation, which may contain slight changes and features not described in this paper, can be found at github.com/hkust-taco/hmloc. An offline version of the user survey, the raw collected data, as well as the processing scripts are available on request.

REFERENCES

Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 177–210. https://doi.org/10.1145/3344429.3372508 ↪ page 26

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. https://doi.org/10.5281/zenodo.8332129 Implementation of the system described in the paper.. ↪ pages 5 and 27

Stephen Dolan. 2017. *Algebraic Subtyping: Distinguished Dissertation 2017*. BCS, Swindon, GBR. https://dl.acm.org/doi/book/10.5555/3180976 ↪ pages 5, 11, and 26

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 60–72. https://doi.org/10.1145/3009837.3009882 ↪ pages 5, 11, and 26

Benedict du Boulay, Tim O'Shea, and John Monk. 1981. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies* 14, 3 (1981), 237–249. https://doi.org/10.1016/S0020-7373(81)80056-9 ↪ page 24

Olive Jean Dunn. 1964. Multiple comparisons using rank sums. *Technometrics* 6, 3 (1964), 241–252. https://doi.org/10.1080/00401706.1964.10490181 ↪ page 21

Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2012. Measuring programming experience. In *2012 20th IEEE international conference on program comprehension (ICPC)*. IEEE, 73–82. https://doi.org/10.1109/ICPC.2012.6240511 ↪ page 20

Holger Gast. 2005. Explaining ML Type Errors by Data Flows. In *Implementation and Application of Functional Languages*, Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 72–89. https://doi.org/10.1007/11431664_5 ↪ pages 5 and 26

Chuqin Geng, Haolin Ye, Yixuan Li, Tianyu Han, Brigitte Pientka, and Xujie Si. 2022. Novice Type Error Diagnosis with Natural Language Models. In *Programming Languages and Systems*, Ilya Sergey (Ed.). Springer Nature Switzerland, 196–214. https://doi.org/10.1007/978-3-031-21037-2_10 ↪ pages 25 and 26

Bastiaan Heeren. 2005. *Top quality type error messages*. Ph. D. Dissertation. Utrecht University. ↪ page 26

Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003. Helium, for Learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell* (Uppsala, Sweden) *(Haskell '03)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/871895.871902 ↪ pages 1, 2, and 20

Felix Henninger, Yury Shevchenko, Ulf K Mertens, Pascal J Kieslich, and Benjamin E Hilbig. 2021. lab.js: A free, open, online study builder. *Behavior Research Methods* (2021), 1–18. https://doi.org/10.3758/s13428-019-01283-5 ↪ page 20

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming* 17, 1 (2007), 1–82. https://doi.org/10.1017/S0956796806006034 ↪ page 27

Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/2931037.2931051 ↪ page 26

William H. Kruskal and W. Allen Wallis. 1952. Use of Ranks in One-Criterion Variance Analysis. *J. Amer. Statist. Assoc.* 47, 260 (1952), 583–621. https://doi.org/10.1080/01621459.1952.10483441 ↪ page 21

Didier Le Botlan and Didier Rémy. 2014. MLF: Raising ML to the Power of System F. *SIGPLAN Not.* 49, 4S (jul 2014), 52–63. https://doi.org/10.1145/2641638.2641653 ↪ page 27

Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. 2016. A Practical Framework for Type Inference Error Explanation. *SIGPLAN Not.* 51, 10 (oct 2016), 781–799. https://doi.org/10.1145/3022671.2983994 ↪ page 26

Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011a. Measuring the Effectiveness of Error Messages Designed for Novice Programmers. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (Dallas, TX, USA) *(SIGCSE '11)*. Association for Computing Machinery, New York, NY, USA, 499–504. https://doi.org/10.1145/1953163.1953308 ↪ page 26

Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011b. Mind Your Language: On Novices' Interactions with Error Messages *(Onward! 2011)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/2048237.2048241 ↪ page 26

Matthias Neubauer and Peter Thiemann. 2003. Discriminative Sum Types Locate the Source of Type Errors. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (Uppsala, Sweden) *(ICFP '03)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/944705.944708 ↪ page 26

Lionel Parreaux. 2020. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 124 (Aug. 2020), 28 pages. https://doi.org/10.1145/3409006 ↪ pages 5, 11, and 26

Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 141 (oct 2022), 30 pages. https://doi.org/10.1145/3563304 ↪ page 26

Didier Rémy. 1992. *Extension of ML type system with a sorted equation theory on types.* Research Report RR-1766. INRIA. https://hal.inria.fr/inria-00077006 Projet FORMEL. ↪ page 27

Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. Learning to Blame: Localizing Novice Type Errors with Data-Driven Diagnosis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 60 (oct 2017), 27 pages. https://doi.org/10.1145/3138818 ↪ pages 20, 25, and 26

Yury Shevchenko. 2022. Open Lab: A web application for running and sharing online experiments. *Behavior Research Methods* 54, 6 (2022), 3118–3125. https://doi.org/10.3758/s13428-021-01776-2 ↪ page 20

Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. 2010. Let should not be generalized. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation.* 39–50. https://doi.org/10.1145/1708016.1708023 ↪ page 27

Mitchell Wand. 1986. Finding the Source of Type Errors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida) *(POPL '86).* Association for Computing Machinery, New York, NY, USA, 38–43. https://doi.org/10.1145/512644.512648 ↪ page 1

John Wrenn and Shriram Krishnamurthi. 2017. Error Messages Are Classifiers: A Process to Design and Evaluate Error Messages *(Onward! 2017).* Association for Computing Machinery, New York, NY, USA, 134–147. https://doi.org/10.1145/3133850.3133862 ↪ page 26

Danfeng Zhang and Andrew C. Myers. 2014. Toward General Diagnosis of Static Errors. *SIGPLAN Not.* 49, 1 (jan 2014), 569–581. https://doi.org/10.1145/2578855.2535870 ↪ pages 25 and 26

Received 2023-04-14; accepted 2023-08-27