# Solved and Open Problems in Type Error Diagnosis [⋆]

Jurriaan Hage

Dept. of Information and Computing Sciences, Utrecht University The Netherlands
j.hage@uu.nl

**Abstract.** The purpose of this paper is to present a number of directions for future research in type error diagnosis. To be able to position these open problems, we first discuss accomplishments in the field without trying to be exhaustive.

**Keywords:** programming languages, type error diagnosis, functional programming paradigm, open problems

## 1 Introduction

Individual lives as well as entire economies now depend on the reliability of software. However, it is a major challenge to ensure that software is safe and correct: it is estimated that programmers make, on average, 15 to 50 errors per 1,000 lines of code [35], and these errors may lead to bugs, system failures, security leaks, or even major catastrophes [8, 15, 18, 45, 52]. There are many approaches to increase the reliability of software, ranging from manual code inspection, via testing, to formal methods such as program verification, model checking, and static typing. In this paper we only consider the light-weight formal method of static typing.

A static type-based approach integrates the verification of program properties tightly with the program source. A program type is effectively an automatically and independently verifiable certificate that the stated property holds for a specific program fragment, module, or the complete program. These properties can then be verified efficiently and effectively by a compiler using mathematically verified techniques, thereby improving the quality of software during its production. Compared to other formal methods it is lightweight, and can therefore be easily embedded into the software development loop. It is not surprising then that most of the recent developments in industrial cutting-edge programming languages draw heavily on types, e.g. Facebook's Hack [5], Google's Go [4], and Mozilla's Rust [7].

The guarantees that types provide are typically not complete: the type of the sorting function may guarantee that it returns a list, but not that the output list is indeed sorted, or even that the input and output list are of the same length. What can and cannot be verified depends on the expressiveness of the type system that is part of the programming language definition. Along this dimension dependently-typed languages such as Agda [1], Coq [2] and Idris [6] are considered to be the most expressive.

Expressiveness and power do come at a price. It seems that the more advanced type level features a language and its implementations support, the more incomprehensible type error messages become, decreasing programmer productivity and hindering the uptake of these features, features that were intended by their designers to increase the number and precision of the guarantees that the compiler can provide for a given application. Martin Odersky, in private communication, called this problem the "type wall", and not addressing the issue may well lead to developers turning their backs on statically typed languages, and flocking to dynamic languages instead.

In this paper, we want to propose a number of challenges within the field of type error diagnosis, embedding these challenges in discussions of the state of the art. We do not try to be comprehensive; there seems to be enough interesting work to be done. Note that the earlier sections consider concrete challenges in rather specific contexts, while later sections are more speculative.

Before we go on: in the literature, some researchers explicitly distinguish between two theoretical extremes: type checking (all types are given and we only need to check these) and type inference (no types are given, the process discovers all types on the go). In practical settings, the distinction is less extreme. Certainly, we do not expect in any programming language that the programmer annotates every single part of his/her program with types, so there will always be some amount of inference (note that inference goes beyond choosing the types of identifiers and also includes coming up with inferred types for all expressions and statements in the program). The extent to which types can be omitted depends intimately on the language and its implementation. Below, we do not distinguish between the two, and use type inferencing to refer to this combined process.

## 2 Type error diagnosis for functional languages

Although it is hard to say why, languages in the functional programming paradigm have received much more attention when it comes to type error diagnosis. Possibly this is because it is much easier to make mistakes when higher-order functions and parametric polymorphism are used on a daily basis. Some researchers in the field have published a manifest that details what properties one may want type error messages to have [58].

The earliest work on type error diagnosis aimed at improving the now classic algorithm $\mathcal{W}$ [16]. Researchers quickly discovered that any implementation that solves unifications during the traversal of the abstract syntax tree was bound to have some form of bias, derived from the particular traversal. For example, the folklore algorithm $\mathcal{M}$ was known to discover mistakes "sooner", that is, having looked at fewer unifications [32]. From an efficiency point of view, this may well be good, but not for type error diagnosis. The reason is that when you look at the type error diagnosis that can be provided, comparing algorithms $\mathcal{M}$ and $\mathcal{W}$, the latter can provide more context about the error, having seen more unifications.

As it happens, when all unifications are equality constraints, the order of solving is irrelevant, and we are in fact free to choose the order in which constraints are solved. This is why algorithm $\mathcal{G}$ was defined [32], providing external control of the various orderings it supports. However, whatever the chosen ordering, there will always be programs for which that ordering leads to a type error message that has some bias. If only the solving order could in some ways be dictated by the program itself.

A different way of implementing the type inference process was proposed by various authors [27, 28, 43]: as a constraint solving problem. This implies that type inference consists of an abstract syntax tree traversal that maps the program to some kind of constraint program (some use a constraint set, others a constraint tree, again others a single constraint in a more elaborate constraint language), which is later "interpreted" by a solver. If the solver succeeds, it returns a substitution as evidence that it did, and otherwise it typically returns a (set of) constraint(s), the solver deems responsible for the error. This separation of concerns has been followed by many others and may now be considered the standard approach [1]. In the case of the Helium compiler [21,22,25,28], constraints were used to decouple the type system from the solving order by mapping the program to a constraint tree. That constraint tree could then be flattened into a list of constraints that were then fed to a solver. The benefits were that when you were unsatisfied with the type error message, you could (1) choose a different ordering, and/or (2) choose a different solver. Various orders were predefined (including the orders of $\mathcal{W}$ and $\mathcal{M}$) as well as various solvers. Typically, the compiler uses a fast greedy solver for equality constraints as long as no error occurs, but when a type error does occur, it starts a more complicated solving process for the binding group where the type inconsistency arises. Such a binding group is typically small compared to the size of the program, and using a more complicated solver did not harm performance very much. In the case of Helium, heuristics were developed to provide tailormade error messages. Helium does this by means of a special datastructure called the *type graph* that can model inconsistent constraint sets over which heuristics, looking for clues, have been defined [21]. For example, we may define a heuristic that depending on how a literal is used, will provide a suggestion to use a different literal (examples taken from [11]).

Given the following simple Haskell program

$$\begin{array}{l} \mathit{shout} :: \mathit{Show}\ a \Rightarrow a \rightarrow \mathit{String} \\ \mathit{shout}\ x = \mathit{show}\ x \mathbin{+\!\!+} \text{'!'} \end{array}$$

a recent version of Helium that is based on OutsideIn(X) [55] returns the following message

[1] The approach is also followed in implementations of realistic functional languages such as Haskell [3,20], in the case of GHC, a constrain-based formulation also became necessary to be able to deal with complexity of the language and its many variations.

```
(2,21): Type error in literal
 expression                : '!'
   type                    : Char
   expected type           : String
 probable fix              : use a string literal instead
```

In this case, the character literal should be a string literal since it is passed to the $(+\!\!+)$ append operator that expects two lists (in Haskell lists of characters are synonymous with strings).

Another example is a heuristic that addresses faulty function applications by suggesting to rearrange, insert or delete arguments. For example,

$g :: [\mathit{Int}] \to [\mathit{Int}]$
$g\ \mathit{xs} = \mathit{map}\ (+1)$

leads to the following type error message:

```
(2,8): Type error in application
 expression       : map (+ 1)
 term             : map
   type           : (Int -> Int) -> [Int]
   does not match : (a   -> b  ) -> [a] -> [b]
 probable fix     : insert a second argument
```

In this case, the type signature provides evidence that *xs* has type $[\mathit{Int}]$ and that the type of the right-hand side should also be $[\mathit{Int}]$. Alas, *map* takes two arguments, a function and a list, of which only one is provided. Therefore the type of the right-hand side is $[\mathit{Int}] \to [\mathit{Int}]$. The heuristics detects this, and suggest to add a second argument. It does not suggest to use *xs* for that argument, however. Another way to fix the issue is to remove *xs* as an argument, so writing $g = \mathit{map}\ (+1)$, but that would assume that *xs* is in fact the missing argument to *map*. This is not unlikely in this particular case, but what if we add another argument of type $[\mathit{Int}]$?

Other research approaches to type error diagnosis are possible too, however, and it makes sense to discuss these at this point before we move on.

The idea of type error slicing is inspired by that of program slicing [57]. In program slicing, a slicing criterion is chosen, e.g., a variable used at a particular line in the program, and then the (backward) slicing removes parts of the program that cannot affect the value of the variable at that program point. This can be used to simplify programs while debugging. In the case of type error slicing, programmers are provided with a program in which all the parts that may contribute to a type error are highlighted; all other parts of the program can be ignored when trying to resolve the inconsistency. Skalpel [44] (a continuation of [19]) implements type error slicers for Standard ML, supporting advanced SML features like modules, which are somewhat related to GADTs in Haskell. [46] adapts this idea to Haskell 98. The advantage of slicing is that the actual location that causes the problem is highlighted, a disadvantage is that many other locations are highlighted as well.

Because type error slices can be large, many researchers prefer to blame one or maybe a few constraints. For example, SHErrLoc [59] uses a graph-based structure to encode the solving process, and then ranks the likeness of a constraint being to blame using a Bayesian model. Their work considers type error reporting for modern Haskell, including local hypotheses. [12] explains type errors in Haskell programs using counter-factual typing, a version of variational typing in which they keep track of the different types that an expression may take. Although computationally somewhat costly, they can propagate type inconsistencies from one binding group to another. [38] achieves something similar by using an iterative deepening approach, in which the body of a binding is inlined in its usage site if a conflict is detected between both. This allows the inferencer to blame a location in the body of a (type correct) function if an application of that function is type incorrect, at the expense of repeatedly calling an SM solver with a growing set of constraints. These papers only address the problem of *error localization*, and they can do little to explain what went wrong, and how to fix the error (by suggesting changes to the source code). This information is available when type graphs are used. Work, beyond Helium, that uses type graphs includes [39] and [56]. In the former type graphs were extended with type classes and row types in the setting of Elm; in the latter heuristics were defined to explain confidentality errors. Type graphs were extended recently to OutsideIn(X), so that existential types and GADTs could be modeled [11].

Some authors use a more complicated structure to diagnose type errors: [40] and [53] expose the trace of the type checker to the programmer (for Scala and OCaml, respectively), and [13] defines an explanation graph for Hindley-Miler type systems, which summarizes the information involved in type checking. LiquidHaskell [54] uses SMT solving as part of type checking. In those cases, *reverse interpolation* [36] can be used to derive a simpler explanation.

From a didactic point of view, some researchers thought it a good idea to let the type inference process, for a type incorrect program, to be best modeled as an interaction between programmer and compiler. The most important research line in this direction is that of Chameleon, although it seems work has been at a standstill for quite some time [50, 51].

For the case that we have no control over the compiler infrastructure, [33] presents an approach in which the compiler is iteratively queried for the well-typedness of modified versions of the programmer input, which are then ranked to present a solution to the type error. A big advantage of this approach is it treats the type inference process as a black box.

Now that the reader should have some idea of the work that has been done in this area, we can now list some challenges within this (narrow) field, including some reflections on the problems at hand.

**Open Problem: advanced type level features and their interactions** The Haskell implementation GHC contains many type system features that have never been considered from the type error diagnosis perspective. The only exception seems to be GADTs and existential types [11]. Although others have made implementations that can work in the presence of some of Haskell's advanced features, as far as this author knows, the benchmarks they employed did not include any programs that used such features. It has not yet been established that good solutions to error diagnosis for these advanced features exist, and even more so what happens when a given programmer selects a particular set of extensions to use within a single application. Does this change the error diagnosis at all? Does this depend strongly on the particular combination of extensions? This is a wide open area with many unanswered questions.

**Open Problem: proof assistance** In dependently typed languages, such as Agda, Idris, and Coq, type correctness corresponds to functional correctness. These languages are usually not used to implement applications, but as proof tools. This means that "error diagnosis" can take on two forms: the usual type error messages we also see come up in everyday functional programming (e.g., you pass the arguments to this function in the wrong order), and the problem of proof assistance (e.g., you need to strenghten this lemma for the theorem to go through). In the former case we can look at research on plain functional languages, in the latter case we find a pretty completely open field of investigation. Only a few works have addressed the problem of error diagnosis at all in this setting [14, 17].

**Open Problem: error explanation and repair** As the above historical overview suggests, most work in this area is on error localization. The advantage of work in that direction is that it is relatively easily to validate: the output is a simple location, not the way a type error message is formulated.

But localization also means we can only point at a location, and not diagnose the mistake, or make suggestions on how to fix the problem. This does not mean localization is altogether useless, and in fact such localizers can also be used in a setting that employs heuristics. Consider a compiler that has any number of implementations of localizing errors. In addition, we implement a heuristic approach in which many different heuristics are asked to come up with both a location and a diagnosis or fix based on that location. A problem when using heuristics is you may have multiple heuristics, each choosing a particular location and explanation. Which heuristic is the right one? Well, maybe that is simply the one that agrees best with the localizers. In other words, the localizers provide more confidence when choosing the right message (or, if we can have multiple messages, which one to put at the top).

**Open Problem: benchmarks** A major problem in all type error diagnosis endeavors is the absence of suitable benchmarks. The author of this paper has collected a "large" number (around 50,000) of programs written by students, that have also been used by others. However, these were collected from second year undergraduate students and they do not employ any advanced features, just plain old Haskell 98. There are not many places where students, in sufficiently large numbers, program using the more advanced features we now will want to be considering, and for publishing about research on these features, benchmarks are indispensable.

**Open Problem: aspects of domain-specific type error diagnosis** There is a strand of work by the author that addresses the problem of type error diagnosis for embedded domain-specific languages, again within the context of Haskell, and mostly within Helium [24,26,48]; a PhD has been published that has similar aims but works on Scala [41]. In these works, domain-specific type error diagnosis is implemented as an external DSL that uses type error specifications to control the solving order and the delivered diagnosis. The specifications are automatically checked for soundness with respect to type system.

Some of the ideas of these works have also been implemented in GHC [47]. In this case, we are limited by what can be grafted onto GHC's type inference engine. But this approach also has significant advantages: changes to the compiler were minimal, and soundness of the type rules is verified by the compiler as part of its type system. Moreover, since the rules are written using the type level programming facilities of Haskell, the methods of abstraction that Haskell provides there are all available to us. A disadvantage is that the diagnosis of type level programming itself is not a solved problem (see earlier), and the control we can exert is less than reported in other work. It is unclear at this time how far the idea can be taken without having to completely redesign the type inference engine of GHC.

In a setting with external DSLs, it may well be possible to approach the problem of type error diagnosis as we do in defining the intrinsic type system of a general purpose language. In [49], an architecture for such an implementation is defined and motivated. The full architecture is not something that has been exhaustively tested, but generally it does seem to be the case that decoupling the specification of the type system in terms of constraints from the solving process is a good idea for two reasons: it simplifies the type system implementation, and it simplifies diagnosis.

## 3 Beyond type error diagnosis in functional languages

The type system that comes with statically typed languages as part of their definition is not the only one of its kind. One can implement various extensions to such type systems. Typical examples include dimension analysis [29,30], type based security analysis [42,56] and pattern match analysis [31,37]. In the latter case, we want the compiler to verify that pattern match failures cannot crash the program, in dimension analysis type are refined by having "floats that represent meters" and "floats that represent kilograms" and preventing the programmer from, say, adding two such floats together. Such types have been implemented in F#. Since we now want to reject more programs, the problem of type error diagnosis will come up sooner or later, although it will depend somewhat on the richness of the dimension types.

In the case of security type systems, the type system prevents values of high confidentiality to end up in variables of low confidentiality. As far as this author knows, this is the only validating analysis for which type error diagnosis has been considered [56].

Much more speculatively, optimising analyses for functional languages are often designed as annotated type systems [34]. Examples are too numerous to mention, and include control-flow analysis, sharing analysis, escape analysis, binding-time analysis. In such a system, annotations, that describe additional properties of the type, are attached to types, and the type system implementation will try to infer the best properties. The above validating analyses actually use the same approach.

The difference is that an optimising analysis typically will accept exactly all programs accepted by the underlying intrinsic type system. At this point, error diagnosis does not play a role. But what if we want to allow the programmer to express certain properties about his/her programs to communicate to the programmer that certain optimisations should be made, and to have the compiler verify that these properties make sense? Then, if the analysis fails, we may want the compiler to explain to the programmer why this is not the case. This is currently a wide open field of investigation.

## 4 Type error diagnosis, elsewhere

What is most striking about type error diagnosis beyond the functional programming languages is that there is almost nothing to report on. A few papers exist that deal with generic method invocations in Java [9,10], and a PhD thesis within the context of Scala [41] that tries to achieve something like the specialized type rules of Helium [26], but with much more control over the process by providing the ability to inspect type inference traces. This increased control does come at the price of more complexity.

Type error diagnosis in this setting may well be harder, because the type systems themselves typically combine higher-order functions, parametric polymorphism and some form of subtyping. Therefore, type inference is algorithmically already quite hard, and diagnosis will typically not make it any easier. Another aspect is that multi-paradigm languages like Scala tend to draw people from different paradigms, and their needs are likely to be different. For example, a Haskell programmer may be surprised that types do not propagate as easily under local type inference in Scala, while Java programmers will not always be that familiar with type variables. Another complication is that a language like Java started from an object-oriented core and had various extensions added to it while having to remain backwards compatible. These grafts on top the language tend to be quite ad hoc, complicating type error diagnosis. To be fair, Haskell seems to have a similar problem as compared to full dependently typed languages.

## 5 What about SLE?

The talk that led to this paper was presented at OOPSLE, which may lead to the question: how is this topic relevant to SLE? Clearly, the problem is related to the implementation of programming languages.

In [49], an architecture is presented for a (Haskell) compiler that has type error diagnosis as part of its design, not as an afterthought. Although it remains to be seen whether the architecture is sufficiently rich to be used in all situations, and whether its features are relevant in every setting, it does tell us that things aren't so simple as they may seem at first. The relevance to SLE is that the work tells us how to organize/structure a compiler to deal with various kinds of tasks in type error diagnosis.

### 5.1 Open Problem: analysis of compilation behavior

A paper published at the first SLE in Toulouse presented Neon [23], a library for analyzing interactions with the compiler in the context of Helium. Helium has built-in log facilities that could communicate compiled programs to a server. Programs, collected during class hours while students were working on their practical assignments, have been used by various authors in the field. The first attempt with Helium to extract information from these compiles is hampered by the fact that the programs were collected in vivo. For example, when someone fixes a mistake after 10 minutes is that because they went for coffee and checked their phones for messages, or were they working on solving the problem? And when after a type error the program compiles correctly is that because a teaching assistant told them how to fix it, did they fix it themselves, or did they delete the definition altgother? And if we perform these experiments with students, either in vivo or in vitro, does that tell us anything about professional developers? Every answer seems to raise only more questions.

### 5.2 Open Problem: everyone is different

Type error diagnosis is to some extent a matter of taste. The thing is that depending on where you come from, your level of expertise, your programming style even, you will be wanting different things from error diagnosis. Maybe, all a seasoned programmer needs is to get the approximate location of a mistake, but what if these seasoned programmers start to uses type system extensions they are not yet familiar with? Do their needs change over time, as they do become more familiar? Everyone seems to have different needs, and at this time we are not yet at the level that we can satisfy the needs themselves. The problem to decide what someone needs and when is likely to be even harder. Some process in which programming becomes an interaction with a compilation system, so that that system can accrue some kind of operational profile is likely to be essential in that case.

Of course, there is a field in computer science that excels at modeling the ad-hocness of human nature: machine learning. But that will only work if we have enough data to learn from. And as we explained above, that data is at this time simply not available.

## 6 Conclusion

The conclusion is simple: there is still an immense amount of work to do.

## References

1. The Agda programming language. http://wiki.portal.chalmers.se/agda/pmwiki.php, accessed: 2020-03-20
2. The Coq proof assistant. https://coq.inria.fr/, accessed: 2020-09-11
3. The Glasgow Haskell Compiler, GHC. https://www.haskell.org/ghc/, accessed: 2020-03-27
4. The Go programming language. https://golang.org/, accessed: 2020-03-27
5. The Hack programming language. https://hacklang.org/, accessed: 2020-03-27
6. The Idris programming language. https://www.idris-lang.org/, accessed: 2020-09-11
7. The Rust programming language. https://www.rust-lang.org/, accessed: 2020-03-27
8. When code can kill or care, tech. Quarterly (Economist), Q2.2012
9. el Boustani, N., Hage, J.: Corrective hints for type incorrect Generic Java programs. In: Gallagher, J., Voigtländer, J. (eds.) Proceedings of the ACM SIGPLAN 2010 Workshop on Partial Evaluation and Program Manipulation (PEPM '10). pp. 5–14. ACM Press (2010)
10. el Boustani, N., Hage, J.: Improving type error messages for generic java. Higher-Order and Symbolic Computation **24**(1), 3–39 (2012), http://dx.doi.org/10.1007/s10990-011-9070-3, 10.1007/s10990-011-9070-3
11. Burgers, J.: Type error diagnosis for OutsideIn(X) in Helium (2019), https://dspace.library.uu.nl/handle/1874/382127
12. Chen, S., Erwig, M.: Counter-factual Typing for Debugging Type Errors. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 583–594. POPL '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2535838.2535863, http://doi.acm.org/10.1145/2535838.2535863
13. Chitil, O.: Compositional explanation of types and algorithmic debugging of type errors. In: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming. pp. 193–204. ICFP '01, ACM, New York, NY, USA (2001). https://doi.org/10.1145/507635.507659, http://doi.acm.org/10.1145/507635.507659
14. Christiansen, D.R.: Reflect on your mistakes ! lightweight domain-specific error messages (2014)
15. Cipra, B.: How number theory got the best of the pentium chip, science, 267:5195, pp.170-175, 1995
16. Damas, L., Milner, R.: Principal Type-schemes for Functional Programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 207–212. POPL '82, ACM, New York, NY, USA (1982). https://doi.org/10.1145/582153.582176, http://doi.acm.org/10.1145/582153.582176
17. Eremondi, J., Swierstra, W., Hage, J.: A framework for improving error messages in dependently-typed languages. Open Comput. Sci. **9**(1), 1–32 (2019). https://doi.org/10.1515/comp-2019-0001, https://doi.org/10.1515/comp-2019-0001

18. Gleick, J.: A bug and a crash (1996), `http://www.around.com/ariane.html`
19. Haack, C., Wells, J.B.: Type Error Slicing in Implicitly Typed Higher-order Languages. Sci. Comput. Program. **50**(1-3), 189–224 (Mar 2004). https://doi.org/10.1016/j.scico.2004.01.004, `http://dx.doi.org/10.1016/j.scico.2004.01.004`
20. Hage, J.: The Helium homepage (2020), `https://github.com/Helium4Haskell/`
21. Hage, J., Heeren, B.: Heuristics for type error discovery and recovery. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) Implementation of Functional Languages – IFL 2006. vol. 4449, pp. 199 – 216. Springer Verlag, Heidelberg (2007)
22. Hage, J., Heeren, B.: Strategies for solving constraints in type and effect systems. Electronic Notes in Theoretical Computer Science **236**, 163 – 183 (2009), proceedings of the 3rd International Workshop on Views On Designing Complex Architectures (VODCA 2008)
23. Hage, J., van Keeken, P.: Neon: A library for language usage analysis. In: Gasevic, D., Lämmel, R., Wyk, E.V. (eds.) Proceedings of the 1st Conference on Software Language Engineering (SLE '08). Lecture Notes in Computer Science, vol. 5452, pp. 35 – 53. Springer (2009), revised selected papers
24. Heeren, B., Hage, J.: Type class directives. In: Seventh International Symposium on Practical Aspects of Declarative Languages. pp. 253 – 267. Springer Verlag, Berlin (2005)
25. Heeren, B., Hage, J., Swierstra, S.D.: Constraint based type inferencing in Helium. In: Silaghi, M.C., Zanker, M. (eds.) Workshop Proceedings of Immediate Applications of Constraint Programming. pp. 59 – 80. Cork (September 2003)
26. Heeren, B., Hage, J., Swierstra, S.D.: Scripting the type inference process. In: Eighth International Conference on Functional Programming. pp. 3 – 13. ACM Press, New York (2003)
27. Heeren, B., Hage, J., Swierstra, S.D.: Constraint based type inferencing in Helium. In: Silaghi, M.C., Zanker, M. (eds.) Workshop Proceedings of Immediate Applications of Constraint Programming. pp. 59 – 80. Cork (September 2003)
28. Heeren, B.J.: Top Quality Type Error Messages. Ph.D. thesis, Universiteit Utrecht, The Netherlands (Sep 2005)
29. Kennedy, A.: Types for units-of-measure: Theory and practice. In: Horváth, Z., Plasmeijer, R., Zsók, V. (eds.) Central European Functional Programming School (CEFP), Lecture Notes in Computer Science, vol. 6299, pp. 268 – 305. Springer Verlag (2010)
30. Kennedy, A.J.: Programming Languages and Dimensions. Ph.D. thesis, Computer Laboratory, University of Cambridge (1995), available as Technical Report No. 391
31. Koot, R., Hage, J.: Type-based exception analysis for non-strict higher-order functional languages with imprecise exception semantics. In: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation. pp. 127–138. PEPM '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2678015.2682542, `http://doi.acm.org/10.1145/2678015.2682542`
32. Lee, O., Yi, K.: A generalization of hybrid let-polymorphic type inference algorithms. In: Proceedings of the First Asian Workshop on Programming Languages and Systems. pp. 79–88. National university of Singapore, Singapore (December 2000)
33. Lerner, B.S., Flower, M., Grossman, D., Chambers, C.: Searching for Type-error Messages. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 425–434. PLDI '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1250734.1250783, `http://doi.acm.org/10.1145/1250734.1250783`

34. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 47–57. ACM, New York, NY, USA (1988). https://doi.org/http://doi.acm.org/10.1145/73560.73564
35. McConnell, S.: Code Complete. A Practical Handbook of Software Construction. 2nd edn. (2015)
36. McMillan, K.L.: An interpolating theorem prover. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 16–30. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
37. Mitchell, N., Runciman, C.: Not all patterns, but enough: an automatic verifier for partial but sufficient pattern matching. In: Proceedings of the first ACM SIGPLAN symposium on Haskell. pp. 49–60. Haskell '08, ACM, New York, NY, USA (2008)
38. Pavlinovic, Z., King, T., Wies, T.: Practical SMT-based Type Error Localization. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. pp. 412–423. ICFP 2015, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2784731.2784765, http://doi.acm.org/10.1145/2784731.2784765
39. Peijnenburg, F., Hage, J., Serrano, A.: Type directives and type graphs in elm. In: Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages, IFL 2016, Leuven, Belgium, August 31 - September 2, 2016. pp. 2:1–2:12 (2016). https://doi.org/10.1145/3064899.3064907, https://doi.org/10.1145/3064899.3064907
40. Plociniczak, H.: Scalad: An Interactive Type-level Debugger. In: Proceedings of the 4th Workshop on Scala. pp. 8:1–8:4. SCALA '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2489837.2489845, http://doi.acm.org/10.1145/2489837.2489845
41. Plociniczak, H.: Decrypting Local Type Inference. Ph.D. thesis, IC, Lausanne (2016). https://doi.org/10.5075/epfl-thesis-6741
42. Pottier, F., Simonet, V.: Information flow inference for ml. In: POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 319–330. ACM, New York, NY, USA (2002). https://doi.org/http://doi.acm.org/10.1145/503272.503302
43. Pottier, F., Rémy, D.: he Essence of ML Type Inference. In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages, chap. 10, pp. 389–489. MIT Press (2005), http://cristal.inria.fr/attapl/
44. Rahli, V., Wells, J., Pirie, J., Kamareddine, F.: Skalpel: A constraint-based type error slicer for Standard ML. J. Symb. Comput. **80**(P1), 164–208 (May 2017). https://doi.org/10.1016/j.jsc.2016.07.013, https://doi.org/10.1016/j.jsc.2016.07.013
45. Reuters: Toyota to recall 436,000 hybrids globally, feb. 2010
46. Schilling, T.: Constraint-free Type Error Slicing. In: Proceedings of the 12th International Conference on Trends in Functional Programming. pp. 1–16. TFP'11, Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32037-8_1, http://dx.doi.org/10.1007/978-3-642-32037-8_1
47. Serrano, A., Hage, J.: Type error customization in ghc: Controlling expression-level type errors by type-level programming. In: Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages. pp. 2:1–2:15. IFL 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3205368.3205370, http://doi.acm.org/10.1145/3205368.3205370

48. Serrano, A., Hage, J.: Type error diagnosis for embedded dsls by two-stage specialized type rules. In: Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Proceedings. pp. 672–698 (2016)
49. Serrano, A., Hage, J.: A compiler architecture for domain-specific type error diagnosis. Open Comput. Sci. **9**(1), 33–51 (2019). https://doi.org/10.1515/comp-2019-0002, `https://doi.org/10.1515/comp-2019-0002`
50. Stuckey, P.J., Sulzmann, M., Wazny, J.: Interactive type debugging in Haskell. pp. 72–83. New York (2003)
51. Stuckey, P.J., Sulzmann, M., Wazny, J.: Improving type error diagnosis. pp. 80–91 (2004)
52. Technica, A.: Airbus confirms software configuration error caused plane crash, 2015
53. Tsushima, K., Asai, K.: An embedded type debugger. In: Hinze, R. (ed.) Implementation and Application of Functional Languages. pp. 190–206. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
54. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton Jones, S.: Refinement Types for Haskell. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. pp. 269–282. ICFP '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2628136.2628161, `http://doi.acm.org/10.1145/2628136.2628161`
55. Vytiniotis, D., Peyton Jones, S., Schrijvers, T., Sulzmann, M.: OutsideIn(x): Modular Type Inference with Local Assumptions. J. Funct. Program. **21**(4-5), 333–412 (Sep 2011). https://doi.org/10.1017/S0956796811000098, `http://dx.doi.org/10.1017/S0956796811000098`
56. Weijers, J., Hage, J., Holdermans, S.: Security type error diagnosis for higher-order, polymorphic languages. Science of Computer Programming **95**, 200 – 218 (2014). https://doi.org/https://doi.org/10.1016/j.scico.2014.03.011, `http://www.sciencedirect.com/science/article/pii/S0167642314001518`, selected and extended papers from Partial Evaluation and Program Manipulation 2013
57. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering. pp. 439–449. ICSE '81, IEEE Press (1981)
58. Yang, J., Michaelson, G., Trinder, P.: Explaining polymorphic types. The Computer Journal **45**(4), 436–452 (2002)
59. Zhang, D., Myers, A.C., Vytiniotis, D., Peyton Jones, S.: Diagnosing Type Errors with Class. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 12–21. PLDI '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2738009, `http://doi.acm.org/10.1145/2737924.2738009`