Literature survey on improving type checker efficiency without altering the surface language

Martijn Staal[1]

Supervisor(s): Jesper Cockx[1]**, Bohdan Liesnikov**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

Abstract

Type checkers are invaluable tools which help programmers write correct programs. Fast and efficient type checkers are required to enable adoption of such tools in practice.

This study aims to provide an explorative overview of proposed efficiency improvements for type checkers. This provides language implementers insight in what approaches exist to increase the performance of their type checker.

Efficiency improvements are divided into three general approaches: incrementalization, parallelization, and algorithmic improvements. For each category, we discuss the implementation techniques and performance for several proposed approaches.

This study finds that a wide variety of approaches exist to increase the efficiency of type checkers. Based on reported benchmark results, incrementalization and parallelization are promising approaches to writing fast type checkers.

1 Introduction

Type checkers are invaluable tools which can help programmers write correct programs [9]. Creating practical type checkers is an essential step in getting the latest advancements in type system theory into daily programming practice. If these type checkers are too slow or require excessive computer resources, they will be adopted less by programmers. More specifically, if type checkers and new type checking functionality are to be implemented in integrated development environments (IDEs) to provide direct feedback to programmers regarding type correctness, they have to be fast enough to not hinder the programmer, even for large code bases [1; 25]. Also, with the rapid adoption of Continuous Integration (CI) tools, which may be used to automatically type check any new commit submitted to a version control system, efficient type checking is key to be able to host CI tools with as low as possible environmental impact and high cost-efficiency.

Despite the importance of having efficient implementations of new typing features to improve practical adoption, type system research is often mostly focused on theoretic properties of type systems and adding new features. Regardless of this common focus, there is still literature on improving the practical efficiency of type checkers, and some more theoretical research also generate efficiency improvements as a side-effect of their main focus. For example, multiple different approaches to parallelize type checking have been proposed in literature [17; 1], and are used in practice [11]. Besides parallelization, there is also research on incrementalizing type checking [14; 25; 10; 3; 1; 4; 5; 26]. Incrementalization may improve type checker performance significantly in specific contexts. One such context is in IDEs, where a type checker has to iteratively analyse source code to provide direct type-correctness feedback to programmers. Incrementalization can also be useful for continuous integration (CI) pipelines, since in a CI the same code is also type checked iteratively with often only small changes. And with regards to sound gradual typing, there is a lively academic debate on if there are actually effective enough implementations to make gradual typing practical [22; 16; 15].

There is however no clear overview of approaches that can be used to increase type checker performance. In order to help implementers choose the the best methods to improve the efficiency of their type checker, in this study we aim to provide an overview of approaches to implement and create efficient type checkers that have been proposed in literature or are currently in practical use.

To provide this overview, the main question of this study is: “What approaches exist in improving type checker efficiency, without altering the surface language, and how do implementation of these approaches compare to each other?”.

To answer this question in a structured manner, it is further divided into three subquestions:

1. What efficiency improvements of type checkers, that do not alter the surface language, have been proposed in literature?
2. What implementation techniques have been used or proposed for these efficiency improvements?
3. How do these implementation techniques compare to each other?

In the second section of this paper, we further discuss the used methods in this research. Then, in the responsible research section we reflect on the application of research integrity standards in this research. In Section 4, we discuss which efficiency improvements of type checkers have been proposed in literature or have been adopted in practice, and how these improvements can be implemented. In Section 5, this paper discusses the reported performance of the discussed efficiency improvements, and compare these with each other. In Section 6, the results from the comparison are discussed and concluded into an advice for language implementers. Finally, Section 7 concludes this paper with future work possibilities and conclusions.

2 Methodology

This study is meant as a first exploration in what efficiency improvements for type checkers have been proposed in literature. Our focus is to present and discuss a representative overview of the current academic literature regarding this topic, but not to be completely exhaustive.

Searching for relevant literature was done in two phases. The first phase consisted of a search via general academic search engines and relevant publishers to discover what categories of improvements are discussed and proposed. Then, in the second phase, per category of improvements a more specific search was conducted to find papers that fit well in the scope of this study.

To be in scope for this study, papers should not only discuss their proposed efficiency improvements, but should also describe actual implementations for these improvements.

Preferably, this also includes a thorough discussion of evaluated performance of these improvements.

Comparing the different approaches to improving type checker efficiency is not straightforward, because it is a rather broad topic. This is further complicated by the fact that reported performance results are hard to compare across papers, and often benchmarks are not easily replicated. This study therefore did not attempt to recommend a single path towards implementing efficient type checkers.

Instead, it aims to provide a good overview of the discussed methods and their reported performance results. For all approaches, we discuss the implementation techniques used and the reported performance results. Specifically note is taken if the benchmarks are done based on synthetic programs, or real-world scenario's.

3 Responsible Research

As an explorative literature survey, the most important aspects regarding responsible research in this study are the reproducibility and the representativeness of the discussed literature.

With regards to reproducibility, it is for this literature survey important that findings are clearly cited, and it is obvious for the reader what efficiency-improvement is discussed by which paper. To enable this, this study also notes page numbers when a reference is used to point to a specific part of a discussed source. Another problem regarding the reproducibility is that the reported performance of papers could not be verified by running these benchmarks again. This is partly due to time constraints, but also because not all papers provide proper and clear artifacts to actually reproduce benchmark results.

To make sure that the study actually covers a representative range of efficiency improvements, we discuss multiple approaches for each category of improvements. Furthermore, in searching for literature to discuss, effort was taken to be as thorough as possible so that a representative set of approaches could be selected from the findings. However, it is still possible that some biases were still present.

4 Efficiency improvements of type checkers and their implementations

Several different approaches to improving the efficiency of type checkers have been proposed in academic literature. This study identifies three general categories of improvements: incrementalization, parallelization, and algorithmic improvements. In this section, we will discuss several approaches to each category of efficiency improvements, and implementation techniques for these approaches.

4.1 Incrementalization

Since programming is an iterative process, the same source code is often type checked multiple times after each other with only small changes. An incremental type checker is a type checker which may re-use the results from a previous run, and can then infer what the result would be if a clean run of the type checker would be done on the new version, based on the results of the previous run and the altered source code [1]. Ideally, this is then faster than running the type checker on the whole program source.

Incrementalization can help specifically when a type checker is integrated in an IDE, where the incremental type checker runs transparently in the background to provide the programmer with immediate feedback on the type correctness of their program [3, p. 1]. Incrementalization can also be useful in making CI's more efficient, since these will often have to type check big code-bases with only minor changes since the previous run. Furthermore, incrementalization can also be a step towards parallelization of a type checker, as is discussed in Section 4.2.

General framework of incrementalizating a type checker Multiple different approaches exist to creating an incremental type checker. These different approaches generally consist of two parts: a separation strategy and an incremental analysis strategy.

The separation strategy is an essential preparation to the actual incremental analysis. The separation strategy is how the incrementalized type checker splits up the source code into several chunks. This is required so the changed chunks can be isolated from the unchanged part of the source code, so that the type checker only has to analyze the chunks that have been affected by the changes in the source code [1].

The granularity of these chunks is an important aspect to how effective the incrementalization can be in increasing type checker efficiency [1, p. 25]. High granularity can ensure that the least amount of chunks will have to be re-analyzed, but this comes at a greater cost in actually splitting up the source code and isolating changes.

The incremental analysis strategy is how the isolated chunks are actually evaluated. Parts of this analysis strategy will be how the type checker determines if a chunk is affected by a change, and how to type check the affected chunks while re-using previous results for the unchanged chunks.

Approaches to incrementalization proposed in literature As discussed, multiple different approaches exist to create a type checker that can run in an incremental fashion. We will discuss the approaches used by Wachsmuth et al. [25], Erdweg et al. [7], Bosma [3], Aerts [1], and Zwaan et al. [26]. These respective approaches will be described in light of the general framework as given above, to make it explicit how these approaches differ. These specific papers are chosen because they represent a broad variety of approaches to incrementalization of type checkers, are recent, and all report performance evaluations.

This is by no means an exhaustive discussion of approaches to incrementalization of type checkers, as other approaches have also been proposed in literature. Some other examples of research regarding incrementalization of type checkers are those by Meertens [14], Kuci et al. [10], Busi et al. [4; 5], and Pacak et al. [18].

Wachsmuth et al. propose a general and language independent approach towards creating incremental type checkers [25]. The approach is based on two phases: a collection and evaluation phase [25, p. 2]. In the collection phase, name binding information is gathered from the source code. During this collection, deferred "analysis tasks" are created by the type checker [25, p. 2]. Each task is essentially one instruction, such as a lookup or cast, with respect to a specific name binding. Information about dependencies between tasks is also collected during the collection phase [25, p. 11]. In the evaluation phase, the deferred analysis tasks are executed to actually get the typing information.

The two phases in the approach by Wachsmuth et al. are not completely analogous to the two strategies as described in the general framework. The phases both partially implement the separation strategy. The collection phase is incremental on the file level [25, p. 2], and thus splits up the source code on the file level. The evaluation phase is incremental on the task level [25, p. 2], so the separation is done here on the task level. Both phases also partially implement the incremental analysis strategy. In an incremental run, the collection phase will only have to collect tasks and information from changed files. The evaluation phase then only will have to evaluate newly created tasks, and those tasks affected by the changes.

Erdweg et al. propose a structure for co-contextual type systems, which are naturally suited to implement incrementality [7]. Instead of starting at the root of the abstract syntax tree (AST) and passing the context down the tree in a top-down fashion, in co-contextual type checking this is replaced by bottom-up propagating of "context requirements" [7, p. 880]. So where in a contextual type checker contexts flow top-down and types flow bottom-up the AST, in a co-contextual type checker both contexts and types flow bottom-up the AST [7, p. 881]. Thanks to the bottom-up approach of co-contextual type checking, no coordination of contexts between sub-expressions is required, [7, p. 881] since context constraints are solved once they flow up the AST. If they cannot be solved, a type error is identified.

This structure makes incrementalization relatively straightforward. Instead of just running the co-contextual type checker again on the whole AST, the incremental co-contextual type checker re-checks only changed subexpressions from the bottom-up. In this bottom-up approach, type checking results are memoized to be used later up the flow of the type checker. Higher in the AST, the type checker will then use the memoized results regarding the changed subexpressions, but also the results regarding unchanged subexpressions from a previous run. This generates constraints in an incremental fashion, which can then be solved. To solve these constraints efficiently, Erdweg et al. propose to solve intermediate constraints in order to keep the size of the final constraint set manageable, and allows the type checker to also re-use intermediate solutions of generated constraints [7, p. 888].

In the approach of Erdweg et al., the two identified parts of the general framework of incrementalization - a separation and an incremental analysis strategy - are closely intertwined. The separation essentially flows from the structure of the type checker, which evaluates each subexpression from the bottom-up. The actual (incremental) analysis is then the way the type checker generates and solves the constraints.

Bosma uses the IncA domain specific language (DSL) to create an incremental type checker for Rust [3]. IncA is a domain specific language developed by Szabó et al. for "the definition of efficient incremental program analyses" [21]. As a separation strategy, IncA "represents computations as graph patterns on top of the abstract syntax tree (AST)", and incremental analysis is then done through graph pattern matching [21]. When the analysis is ran incrementally, this is communicated to the start of the graph and then propagated throughout the graph. Only the nodes in the graph that depend on the changed code will then be reanalyzed.

IncA is not specifically made for type checking, but Bosma shows that it can be used to write incremental type checkers. A benefit of using IncA for such incremental analysis, is that the incrementality is provided by the language itself and thus transparent for the developers of the type checker. However, existing type checkers will generally not already have been written in IncA, so that it does not seem to be a good general strategy for adding incrementality, since it will probably require a complete rewrite of the type checker.

Aerts also employs a two phase strategy to incrementalize type checking, in this case for Statix [1]. First, as a separation strategy, Aerts defines a method to separate the source code into a set of modules. What specifically a module is, depends in the Statix type checker on what is treated as a module in the language of the source code that is being type checked [1, p. 25]. Since Statix uses a constraint-solver-based type checking method, part of the separation strategy in Aerts incrementalization effort, is splitting up the single constraint solver into multiple solvers that each, cooperatively, solve a module [1, p. 31]. Coordination of the multiple solvers is done by the "Solver Coordinator", which "ensures that each solver that is able to make progress, is allocated some time to actually make that progress" [1, p. 31].

As the incremental analysis strategy, Aerts constructs a model for incremental analysis that allows for different algorithms to perform the incremental analysis [1, p. 45-46]. This model consists of four main features. The first feature is that the model is aware of the modules created in the separation strategy. The second feature is that the model detects dependencies between these modules. Furthermore, to detect the impact of changes in a module, the model supports scope graph comparisons. The last core feature is that the model allows for different strategies (algorithms for incremental analysis) to manage the solving process [1, p. 62].

Zwaan et al. propose a general technique to incrementalize type checkers that are based on scope graphs, and implement it for type-system specifications written in Statix [26]. Zwaan et al. build their incrementality on previous work by Van Antwerpen and Visser which add the *hierarchical compilation unit model*, in which the program is divided in *compilation units* with each their own local scope graph, and a technique to type check these units concurrently [24; 26]. In this approach for incrementalization, the hierarchical compilation unit model can be seen as the separation strategy. The model of hierarchical compilational units provides an abstraction for the concrete compilational structure of a specific target language and allows for more generic approach to discussing, for example, type checking a program in some separated chunks. The downside of this is however, that for each target language an implementation of compilational units is still required.

Zwaan et al. use this separation and parallelization to add incrementality to the type checker, which "relies on the observation that a type checker result is determined completely by the AST of the compilation unit, and the result of external name lookups." [26, p. 140:7]. This observation means that if the AST and external name lookups of a compilation unit stay the same, the type checking result for that compilation unit can be re-used. Since, in this approach, name lookups are modeled as queries in scope graphs [26, p. 140:2], these queries can be used to resolve dependencies between compilation units [26, p. 140:2]. Only if query results are not the same, a unit has to be re-analyzed because the typing information can be changed. Instead of re-computing all queries, Zwaan et al. propose a technique of *query confirmation* which verify if a query result is the same. This strategy is more efficient than re-computing queries, since only incoming queries of units that need to be re-analyzed have to be confirmed. This query confirmation algorithm is the core of the incremental analysis strategy of this approach to incrementalization, since it resolves both dependencies and evaluates what results can be re-used, and which units have to be re-analyzed.

4.2 Parallelization

With the widespread availability and adoption of multiprocessor computers, parallelization is an obvious way to increase the speed of a program. It is however not always evident how non-trivial programs, such as type checkers, can be adjusted to use multiprocessor computers to their full advantage.

In literature and in practice, multiple different approaches are used to parallelize existing type checkers. This study will discuss the approach presented by Newton et al. which uses LVars [17], and the approach based on incrementalization as used by Aerts [1].

Parallelization using LVars

Newton et al. propose a method for parallelizing the Typed Racket type checker using "Saturating LVars". LVars are *monotonic data structures* that allow for deterministic parallel programming in functional programs [17, p. 1]. With LVars, multiple parallel threads can write to the same LVar, while still getting determinstic results. This determinism is guaranteed because the `put` operation used to write to these LVars always commutes [17, p. 2]. Newton et al. show that LVars can be used to parallelize type inference, by performing unification of constraints concurrently [17, p. 4].

Saturating LVars are an addition to these LVars where errors are "trapped" in a `Saturated` state in the LVar, so that error handling can be avoided. This is important, because in Haskell error handling must be done via the `IO` monad which might introduce nondeterminism [17, p. 5]. Using Saturating LVars, it is thus possible to simply return that type checking failed instead of handling errors.

Newton et al. first describe their approach by implementing their parallelization approach for simple Hindley-Milner type inference, and then add it to the type checker for Typed Racket. They use the Haskell "LVish" library as starting point for implementing Saturated LVars [17, p. 1]. Their implementation support both and-parallelism and or-parallelism, which can independently be enabled or disabled.

Parallelization through incrementalization

In his efforts to incrementalize type checking in Statix, **Aerts**, as described above, also split up the constraint-solving in Statix. Each module is solved by a separate solver, which create a natural avenue for parallelization [1, p. 59]. It is then only the Solver Coordinator, which oversees the solvers for all separate modules, that has to be adjusted to allow for parallel execution. Caution has to be taken to ensure that the parallel execution is safe, as with any parallel program. Aerts solves this problem by leveraging existing features in Statix, which allows constraints to be solved non-deterministically [1, p. 61]. This makes it possible that when a solver queries information from a different module, it will either receive a complete answer, or waits for one.

Parallelization through incrementalization is also sometimes proposed as possible future work after incrementalization efforts, such as by Erdweg et al. [7].

Other approaches to parallelization of type checkers

As discussed above, Zwaan et al. based their incrementalization on an addition to the use of scope graphs for type checking proposed by **Van Antwerpen and Visser** [24; 26], which includes parallelization. The technique proposed by Van Antwerpen and Visser leverage their hierarchical compilation unit model to implement a parallel type checker.

The **Flix programming language** has a type checker that is parallel with function-level granularity [11; 12]. Although there is no thorough discussion of the parallelization of the Flix type checker, the key to the implementation is that Flix requires all functions to have type signatures. Since the type checker verifies if the type of a function actually matches the given type signature, the type checker can assume that the type of all other functions is actually as defined in the type signature [13]. With this assumption, all functions can be type checked independently and in parallel.

The approach used in Flix is straightforward and simple, but can not be leveraged in all programming languages without altering the surface language, since type signatures of functions are essential to this approach.

4.3 Algorithmic improvements

Where incrementalization and parallelization are in essence running the same algorithms but in a more efficient way, the last category is that of algorithmic improvements: using different and more efficient algorithms, or more efficient implementations of algorithms, but still get the same outcome.

This category of improvements is by far the most heterogenous category, as they depend on what aspect of an existing type checker they improve. They can also be optimized for a specific use-case, propose a completely new algorithm to achieve the same result, or instead improve existing algorithms. Furthermore, these improvements also depend on the featureset that is to be implemented in the type checker.

Since this category of improvements is so diverse, we will only discuss two examples of improvements that fit in this category to illustrate the possibilities. We do not intend to provide an exhaustive or even representative sample of possible algorithmic improvements, because that would be out of scope for the more generalistic perspective of this study.

Also, because the discussed papers are more illustrative examples and will not be thoroughly compared due to the diversity of improvements in this category, the discussion of each example will be less thorough than the discussion of improvements of the other categories.

These examples are chosen because they report benchmark results, and represent two different approaches to algorithmic improvements: creating a new algorithm optimized for a different use-case (Bellamy et al. [2]), and using a different but existing algorithm (Rajendrakumar and Bieniusa [19]).

The first example of a more efficient algorithm designed for a specific case is that proposed by **Bellamy et al.** [2], which was specifically designed with type inference of local variables in Java bytecode. Bellamy et al. propose a new algorithm for this type of type inference, which was a result of optimizing not for the worst-case, but for the general case.

In our second example, **Rajendrakumar and Bieniusa** propose and implement a prototype for a bidirectional type checker for Erlang [19], which they compare with the Hindley-Milner-based Erlang Type Checker (ETC) by Valliappan and Hughes [23]. The bidirectional approach by Rajendrakumar and Bieniusa has several improvements over the ETC, such as better error locality [19, p. 55], and their prototype already works on more benchmarks without having to alter the target source code. Their type checker however does require adding extra type specifications in some cases [19, p. 62].

Many other algorithmic improvements exist. For example, specifically within the field of gradual typing, approaches to efficient gradual typing are discussed by Rastogi et al. [20], Muehlboeck and Tate [16], Castagna et al. [6], and by Moy et al. [15].

5 Reported efficiency improvements and comparison

In this section, we will discuss the reported performance improvements of the discussed approaches and then compare these approaches.

5.1 Reported performance improvements

First, we discuss the reported performance improvements of the approaches we discussed in Section 4. An overview of these reported performance improvements is given in Table 1.

Incrementalization

For the type checker improvements that implement incrementalization, we define "clean analysis time" as the time it takes to type check a target without the use of previous results, and "incremental analysis time" as the type it takes to type check a target with the use of previous results.

Wachsmuth et al. report a significant improvement in analysis times when comparing incremental analysis times to clean analyses. On average, the incremental analysis is about 10 times faster than clean analysis, with a worst case of about 2 times as fast than the clean analysis [25, p. 15-16]. Their reported results are based on analyzing existing open source WebDSL applications. For evaluating incremental analysis time, they used the actual revisions of these applications.

Erdweg et al. evaluate the performance of their incrementalized type checker using synthesized input programs [7, p. 890]. They evaluate four different versions of their incremental type checker, each with more optimizations. These four versions are then compared to a standard non-incremental, top-down and contextual constraint-based type checker. In general, they report a significant decrease in incremental analysis time in comparison with the standard type checker, with a speed-up of up to 10 times when the incremental changes are small. Overall performance measured in nodes per millisecond evaluated saw an improvement of up to 24.56x [7, p. 891-892].

The reported non-incremental performance of the co-contextual type checkers is only slightly worse than the compared standard type checker. This is on average however, and in some situations the co-contextual type checkers may perform better or worse than the standard type checker that they are compared to [7, p. 890-891].

Bosma reports quite a bit slower clean analysis times than the reference implementation of the Rust type checker, with 47.395 seconds for the incremental IncA-based type checker, and 4.156 seconds for the reference type checker. The clean analysis thus has a slowdown of 11.4 times. Incremental analysis times are however a great deal faster, with an average of 33.6 ms and a maximum of 434 ms [3, p. 54]. So the IncA-based incremental implementation by Bosma is thus on average about 123 times faster, and in the worst case 9.5 times faster than the reference Rust type checker, when running incrementally.

The incremental analysis times are however based on synthetically generated changes in the code base, which may not necessarily reflect real-world scenario's. Bosma does not discuss any impact on storage or memory usage of the incremental type checker in comparison with the reference implementation.

Aerts performed benchmarks on a small and large software project. The incremental analysis times reported by Aerts are based on synthetically generated changes in the code base, and not based on real-world change history [1, p. 78].

Incremental analysis times are consistently better than clean analysis times for small changes, and not worse for big changes [1, p. 81-83]. Clean analysis times where up to 10% slower using the incremental solver than the original solver [1, p. 76]. Aerts also reports an increase of 447% of the size of the analysis results when comparing the incremental type checker with the original one [1, p. 77]. This underlines that incrementalization is a trade-off, which might or might not be worthwile in every situation.

Zwaan et al. evaluated their incrementalization efforts using both synthetic and real-world benchmarks, and using Statix type-system declarations for two languages: Java and WebDSL [26, p. 140:17].

In synthetic benchmarks, their implementation realized a speedup of up to 147 times relative to non-incremental analysis. This speedup is biggest when a large portion of the synthetic code-base can be re-used and the type checker runs with only a single CPU core. With increased CPU cores available to the type checker, the speedup relative to the non-incremental type checker is decreased [26, p. 140:18-

19]. For example, the relative speedup in the large synthetic benchmark is up to 147 times with only 1 CPU core, and about 50 times with 16 CPU cores. Not unexpectedly, relative speedups are lower when the type checked code-base is smaller relative to the size of the changes, since there is less analysis that is re-used relative to the non-incremental type checker.

For real-world benchmarks, Zwaan et al. ran their incremental type checker on 3 commits each from five open source software projects, three in Java and 2 in WebDSL. Just as in the synthetic benchmarks, for large projects incrementalization can provide substantial performance increase, although the speedups are not as high as in the synthetic benchmarks. For one of bigger the Java projects, Commons-Lang, an average relative speedup of 18 times is reported for single-core benchmarks. As in the synthetic benchmarks, reported relative speedups with a higher number of CPU cores allocated, are lower. The same is the case for the real-world benchmarks in small projects, such as Commons-CSV, which only realized an average 1.5x speedup in single-core benchmarks, and even lower when more CPU cores were available.

Parallelization

Newton et al. evaluate their parallelization of the Typed Racket type checker using two synthetic benchmark cases, which should represent the most important obstacles in the performance of Typed Racket [17, p. 9]. The first case, dubbed “Bigcall” by the authors, concerns a combination of polymorphism and overloading. The second case, named “Treecall”, concerns very large constant data.

With regards to the Bigcall case, Newton et al. report a relative speedup of up to 80 times. This speedup also manages to scale well with increased number of available threads, at least as far as tested by Newton et al. In the Treecall case, they report relative speedups of up to 7.68 times with 8 threads and up to 3.17 times with 4 threads.

Aerts reports an increase the clean analysis time using the incrementalized type checker in a non-concurrent fashion. However, once any concurrency is introduced, even clean analysis times are more than 10% lower than the initial type checker when using two threads, and up to 45% faster when using 8 threads [1, p. 76]. Since these benchmarks were done on a computer with only 4 physical CPU cores, but due to multithreading technology has 8 logical threads available, it is very well possible that the concurrency speedup is even higher for 8 threads on computers with 8 actual physical CPU cores.

Algorithmic improvements

The algorithm proposed by **Bellamy et al.** realizes significant speedups across all benchmarks in their benchmark suite, with relative speedups of between 4.01 times and 428.17 times, and 24.72 times on average [2, p. 483]. The speedups are relative to the previous best solution to their specific problem, by Gagnon et al. [8]. The benchmarks were done on real-world Java codebases.

Rajendrakumar and Bieniusa evaluate their prototype by running it on three modules from the Erlang standard library [19, p. 62]. In one case, their prototype is 6% faster than the Hindley-Milner-based Erlang Type Checker (ETC). In one other case their prototype is about 3,3% slower. In the third and last case, no speed comparison can be made, since that module could not be type checked by ETC.

Paper	Type of improvement	Reported speedup
Wachsmuth et al. [25]	Incrementalization	Relative speedup of up to 10x in real-world benchmarks in incremental runs
Erdweg et al. [7]	Incrementalization	Relative speedup of up to 10x in synthetic benchmarks in incremental runs
Bosma [3]	Incrementalization	Average relative speedup of 123x, with worst case of 9.5x in synthetic benchmarks in incremental runs
Aerts [1]	Incrementalization and parallelization	Relative speedup of up to 45% in real-world benchmarks in parallel clean runs, up to 6x in incremental runs
Zwaan et al. [26]	Incrementalization	Relative speedup of up to 147x in synthetic benchmarks and up to 18x in real-world benchmarks for incremental runs
Newton et al. [17]	Parallelization	Relative speedup of up to 80x in synthetic benchmark
Bellamy et al. [2]	Algorithmic improvement	Relative speedup of 24.72x on average, up to 428.17x in real-world benchmarks
Rajendrakumar and Bieniusa [19]	Algorithmic improvement	Between 3,3% slower and 6% faster in real-world benchmarks

Table 1: Overview of reported efficiency improvements

5.2 Comparison

Before starting the comparison, it is important to note that these reported results cannot easily be compared, since they all used different benchmarks and environments for their benchmarks. These results will therefore only be used as rough indication of the potential of the proposed improvement.

Based on the previous discussion of the implementation techniques and reported performance of the several approaches, it is clear that there is no silver bullet for making a type checker faster. Incrementalization and parallelization are clearly very promising as general techniques, but there are still many very different approaches to implement them. For example, the approaches to incrementalization by Erdweg et al. [7], Bosma [3], and Zwaan [26] all depend on some existing structure of the type checker. If these approaches are to be implemented in type checkers that do not have that existing structure, this would probably require a complete rewrite of the type checker. The approach by Aerts [1] is an insightful view on how to add incrementalization and parallelization to an existing type checker. It shows that it often requires an approach specific to the existing structure of the type checker.

Algorithmic improvements are nearly by definition dependent on the existing type checker. The discussed approaches can however serve as inspiration and examples. For example, it might be useful to think about for which cases an algorithm has to be optimized to be useful in the real-world.

6 Discussion

As can be seen in the overview in Table 1, the discussed efficiency improvements can yield significant performance gains, both in synthetic and real-world benchmarks. Specifically incrementalization and parallelization are generic approaches which can realize significant speed improvements. These performance improvements do come with trade-offs however.

First of all, these performance improvements are not always straightforward to implement, and certainly not in existing type checkers. Some approaches to a specific efficiency improvement might be fundamentally incompatible with the structure of an existing type checker, or not compatible with other requirements for a type checker. For example, the approach by Bosma requires the use of a specific incrementalization-focused language for writing the type checker [3].

Another trade-off is that improvements in run-time might require the use of more other resources, such as storage space. For example, Aerts reports a 447% increase in the size of the analysis results [1, p. 77].

For incrementalization specifically, the clean analysis time is nearly always worse than in a non-incremental type checker. Although this is in general a small price to pay for the incremental performance improvements, this is probably not always the best performing strategy.

6.1 Advice for language implementers

Based on the discussed implementation techniques in Section 4 and the reported performance discussed in Section 5, this study has the following advice for language implementers:

- Incrementalization is quickly becoming a *must have* in order to incorporate a type checker in IDE's and CI's. If it is still possible, try to incorporate it at a fundamental level in your type checker, since adding it later will probably be a lot of work.

- Approaches exist to implement parallelization based on the incrementalization of a type checker, and vice versa. Implementing either of these thus often allows for easier implementation of the other, potentially realizing significant overall performance improvements.

- A lot of different approaches exist to implement incrementalization and parallelization in a type checker. Research these approaches carefully and see which fits your use case best.

With regards to algorithmic improvements of type checkers, it is hard to give general advice, since this depends a lot on what kind of language has to be type checked and the existing structure of the type checker.

7 Conclusions and Future Work

We have identified three general categories of efficiency improvements of type checkers which do not require altering the surface language: incrementalization, parallelization and algorithmic improvements. A type checker is incremental when it can re-use previous results, so that unchanged code does not need to be type checked again. Incrementalization can in some cases be a step towards making a type checker run efficiently in parallel, but other approaches to parallelization exist as well. The last category, algorithmic improvements, are those improvements where more efficient algorithms or more efficient implementations for algorithms are used in a type checker.

For each category of improvements, several different approaches and corresponding implementation techniques have been proposed in literature. We have discussed an explorative sample of these approaches. Based on this sample, specifically incrementalization and parallelization are promising approaches to speeding up a type checker. Incrementalization is useful in IDE's, where it enables real-time feedback on the type correctness of a program.

7.1 Future work

Several possibilities for future work exist. First of all, most papers that propose incrementalization as a means for improving type checker efficiency only mention IDE-integration as an important practical motive for adding incrementalization to a type checker. However, there are other circumstances where incremental type checkers can provide a practical speedup. One of these is when type checkers are used in Continuous Integration (CI) pipelines. Here, just as in IDE's, programs are often iteratively type checked with small changes. Existing incrementalization approaches focused for IDE's may be able to be leveraged for CI's as well. CI-specific optimizations might also exist, such as caching and sharing type checker among different branches in the version control system of the same project.

Moreover, further research can be done into the interaction between parallelization and incrementalization. Parallelization through incrementalization is implemented by Aerts [1], and recognized as a possibility by Erdweg et al. [7, p. 893]. Incrementalization through parallelization is implemented by Zwaan et al. [26]. Since these are both highly promising improvements and clearly have overlap in their implementation techniques, it would be interesting to create a general framework to implement both incrementalization and parallelization in a type checker.

References

[1] Taico Aerts. Incrementalizing Statix. Master's thesis, Delft University of Technology, 2019.

[2] Ben Bellamy, Pavel Avgustinov, Oege de Moor, and Damien Sereni. Efficient local type inference. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. ACM, October 2008.

[3] Sander Bosma. Incremental type checking in IncA. Master's thesis, Delft University of Technology, 2018.

[4] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. Using standard typing algorithms incrementally. In *Lecture Notes in Computer Science*, pages 106–122. Springer International Publishing, 2019.

[5] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. Mechanical incrementalization of typing algorithms. *Science of Computer Programming*, 208:102657, August 2021.

[6] Giuseppe Castagna, Guillaume Duboc, Victor Lanvin, and Jeremy G. Siek. A space-efficient call-by-value virtual machine for gradual set-theoretic types. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. ACM, September 2019.

[7] Sebastian Erdweg, Oliver Bračevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, October 2015.

[8] Etienne M Gagnon, Laurie J Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In *Static Analysis: 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29-July 1, 2000. Proceedings 7*, pages 199–219. Springer, 2000.

[9] Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, May 2017.

[10] Edlira Kuci, Sebastian Erdweg, and Mira Mezini. Toward incremental type checking for java. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. ACM, October 2015.

[11] Magnus Madsen. GitHub pull request #517 in Flix, "Parallelize the typer". https://github.com/flix/flix/pull/517, August 2017.

[12] Magnus Madsen. The principles of the Flix programming language. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, November 2022.

[13] Magnus Madsen. Personal communication, June 2023.

[14] Lambert Meertens. Incremental polymorphic type checking in B. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '83*. ACM Press, 1983.

[15] Cameron Moy, Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Corpse reviver: sound and efficient gradual typing via contract verification. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, January 2021.

[16] Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, October 2017.

[17] Ryan R. Newton, Ömer S. Ağacan, Peter Fogg, and Sam Tobin-Hochstadt. Parallel type-checking with Haskell using Saturating LVars and Stream Generators. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, February 2016.

[18] André Pacak, Sebastian Erdweg, and Tamás Szabó. A systematic approach to deriving incremental type checkers. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, November 2020.

[19] Nithin Vadukkumchery Rajendrakumar and Annette Bieniusa. Bidirectional typing for erlang. In *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang*. ACM, August 2021.

[20] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, January 2015.

[21] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, August 2016.

[22] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, January 2016.

[23] Nachiappan Valliappan and John Hughes. Typing the wild in erlang. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang*. ACM, September 2018.

[24] Hendrik van Antwerpen and Eelco Visser. Scope states: Guarding safety of name resolution in parallel type checkers. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[25] Guido H. Wachsmuth, Gabriël D. P. Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. A language independent task engine for incremental name

and type analysis. In *Software Language Engineering*, pages 260–280. Springer International Publishing, 2013.

[26] Aron Zwaan, Hendrik van Antwerpen, and Eelco Visser. Incremental type-checking for free: using scope graphs to derive incremental type-checkers. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):424–448, October 2022.