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

Martijn Staal [1]

[1]Delft University of Technology

TU Delft

Introduction

- Type checkers are invaluable tools which can help programmers write correct programs.
- Practical and fast type checkers are essential to widespread adoption of the latest advancements in type system theory into daily programming practice.
- Currently, there is no clear overview of approaches to improve type checker efficiency.
- The aim of this study is to provide an **explorative overview of proposed efficiency improvements in type checkers**.

Purposes of improving type checker efficiency

Some reasons of why it is important to improve type checker efficiency are:

- **Enable adoption**: New theoretical advances in typing theory need practical tools to be adopted in the field.
- **Real-time feedback**: Fast type checkers are required to provide real-time type correctness feedback in integrated development environments (IDEs).
- **Environmentally-aware**: Resource efficient typing tools require less energy and less hardware, thus decreasing the environmental impact of software development.

Research question

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?"**

This main question is divided into these 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?

Methodology

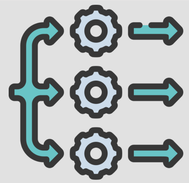
- Explorative search in current academic literature to present and discuss a representative overview of efficiency improvements.
- Only papers that describe an efficiency improvement, an implementation for this improvement and performance benchmarks are selected for discussion.

Categories of efficiency improvements

Incrementalization

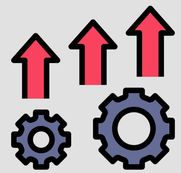
- An incremental type checker may re-use the results from a previous run, so that the amount of re-analyzed code is minimized. [8, 4, 3, 1, 9]
- Incrementalization is particularly useful when integrating a type checker in an integrated development environment (IDE) to realize real-time feedback on type correctness.
- Incrementalization can be a first step to implementing a parallel type checker [4, 1].

Parallelization



- Nowadays, nearly all computers have multiple CPU cores, but writing type checkers to fully leverage these is a challenge.
- Type checkers can be made to run in parallel via various strategies [6, 5, 1].
- Parallelization can be a first step to implementing an incremental type checker [9].

Algorithmic improvements



- All improvements in type checkers due to using more efficient algorithms, or more efficient implementations of the same algorithms.
- Extremely varied category of mostly type checker-dependent improvements.

Comparison and reported performance results

- Many different approaches exist to implement these general techniques, but these approaches often depend on a specific structure of the type checker.
- Algorithmic improvements in general depend on the existing type checker.

Type of improvement	Reported speedups in real-world benchmarks	Reported speedups in synthetic benchmarks
Incrementalization	up to 6x, 10x or 18x [1, 8, 9]	up to 10x, 123x, 147x [4, 3, 9]
Parallelization	up to 45% [1]	up to 80x [6]
Algorithmic improvements	24.72x on average, up to 428.17x [2], between -3.3% and +6% [7]	*None reported*

Table 1. Overview of reported efficiency improvements

Conclusions

- Incrementalization and parallelization are both promising general techniques for improving type checker efficiency.
- Incrementalization is becoming a must have in order to use a type checker in an IDE.
- Since incrementalization can be a step towards implementing parallelization in a type checker, and vice versa, implementing either of these can yield significant performance improvements.

Future work

- Incrementalization specifically optimized for Continuous Integration (CI) pipelines.
- A combined general approach to implement both incrementalization and parallelization.

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] 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, oct 2015.

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

[6] 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, feb 2016.

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

[8] 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.

[9] 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, oct 2022.

m.a.staal@student.tudelft.nl