Tatiana Sedelnikov

Implementing an Incremental Type Checker

Computer Science Tripos – Part II
Queens' College

13th May 2022

Declaration

I, Tatiana Sedelnikov of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Tatiana Sedelnikov of Queens' College, am content for my dissertation to be made available to the students and staff of the University.

Signed: Tatiana Sedelnikov
Date: 13th May 2022

Proforma

Candidate number:	**2412E**
Project Title:	**Implementing an Incremental Type Checker**
Examination:	**Computer Science Tripos – Part II, 2022**
Word Count:	**8496** [1]
Code Line Count:	**4066** [2]
Project Originator:	**The Dissertation Author and Mistral Contrastin**
Project Supervisor:	**Mistral Contrastin and Neel Krishnaswami**

Original Aims of the Project

An incremental type checker is a software development tool which type checks a program after every edit made to it — only performing computation relevant to the changes made without having to re-check the complete program. The main goal of this project was to implement such a type checker targeting an imperative programming language. I aimed to explore the approach of writing the type checking analysis in an incrementally evaluated dialect of the logic programming language Datalog to achieve incrementalisation.

Work Completed

The goal of this project has been achieved successfully: I have implemented the *Cerium* type checker, an incremental type checker for a syntactic subset of C called *Ceres*. This involved implementing a type checking analysis in Datalog, as well as algorithms in Rust for transforming programs so they could be passed to Datalog and identifying changes between two programs. I also showed that it far outperforms a standard type checker when it comes to re-checking programs after small changes and explored limitations to the approach taken.

Special Difficulties

Frequent cases of illness throughout the year the meant that while the core aims of the project have been achieved, any time for extensions included in the original timetable had to be used as additional buffer time.

[1] This word count was computed using `texcount`.

[2] This code line count was computed find . −name '∗.x' | xargs wc −l where x ∈ rs, dl, toml.

Contents

List of Figures

Chapter 1

Introduction

An incremental type checker can be defined as a type checker which, when repeatedly type checking a changing program, aims to only perform **computation proportional to the size of the change** which triggered the analysis. This makes it a useful software development tool in cases where programmers require instant feedback during code editing.

This project aims to explore an approach to implementing such a type checker where the type checking analysis is implemented using an **incrementally evaluated dialect of Datalog**. Datalog is a logic programming language which can be used to represent code as a database of facts from which further information can be inferred, This has made it a popular choice for program analysis tools such as *Doop* [1] and *Souffle* [2].

Since it also lends itself well to being incrementalised by continuously updating its evaluation in response to input changes, this has recently lead to the idea of using it for incremental program analysis. Notably [3] introduces a systematic approach to deriving incremental type checkers in Datalog. As the paper primarily focuses on functional programming languages and theoretical formalisations, the novelty in this project lies in applying the approach by creating a **practical tool for an imperative language**.

1.1 Motivation

An important part of programming and software development these days are **integrated development environments (IDEs) and code editors**. According to the *JetBrains State of Developer Ecosystem 2021* survey (see [4], answered by 31,743 developers from 183 countries) 78% percent of developers use a standalone IDE and 76% use a desktop editor.

A key functionality that IDEs provide is **real-time feedback** on the correctness of a program after every keystroke. This however means that all static analysis — meaning analysis we can perform on source code without having to run it — has to be constantly re-run from scratch, even though each change is usually very small and only affects certain parts of the code. Given a potentially very large codebase, this becomes expensive in terms of time complexity. To avoid redundant computation and speed up analysis, you want to ideally only perform computation that is relevant to a particular change.

This project focuses on type-checking as the analysis we want to perform incrementally as it plays an important part in ensuring **code safety** in statically typed languages. Ensuring that any typing-related bugs or inconsistencies are found as early and quickly as possible during development is critical.

Compared to other approaches, which mostly involve modifying standard typing algorithms directly (see [5] as an example), implementing a type checker using incremental Datalog has multiple advantages:

- **Separation of incrementalisation and analysis**: One major benefit is not having to deal with the details of incrementalising the analysis algorithm: we can write the analysis code as we would do in the non-incremental case and the Datalog engine will automatically incrementalise it. This also means that any improvements in the engine will automatically apply to any analysis.
- **Extensibility**: Following on from the point above, once you have a working framework which computes a program delta and performs one kind of static analysis on it, it can easily be extended to allow for other kinds of analysis. This is particularly useful since already existing analyses can be incrementalised this way.
- **Correspondence between logic programming and typing rules**: While translating typing rules to Datalog is not completely straightforward, generally since there exists a correspondence between type systems and logic, expressing them in a logic programming language is relatively intuitive.

As the basis for the imperative language the type checker targets, **C** was an obvious choice as it is a popular general-purpose language, allowing me to use existing tool when needed while having the flexibility to chose a subset appropriate for the scope of the project.

1.2 Project Accomplishments

The result of this project is the *Cerium* type checker, an incremental type checker for a syntactic subset of C called *Ceres*. It consists of the following components:

- A **framework** which transforms code to Datalog relations and performs tree differencing to find the the program delta after any change has been made to the source code that is being analysed, and
- the core **type checking analysis** written in Datalog.

Additionally I also present empirical evidence for *Cerium* satisfying the definition of an incremental type checker by performing a series of **benchmarks** against a baseline type checker on a dataset of example programs and changes I created. Limitations of the approach in terms of creating a performance bottleneck through tree differencing are also discussed.

Chapter 2

Preparation

This chapter provides background knowledge for the project as well as detailing the preparatory work undertaken before starting the implementation. After defining the project requirements and justifying any decisions made with regards to implementation languages and libraries, I start by formalising the static semantics of a small subset of C to have a suitable imperative language the type checker can target in the scope of the project. This is followed by a brief introduction to the Datalog language and in particular Differential Datalog (DDlog), a dialect of Datalog which can automate incremental computation. The chapter concludes by describing the approach taken to ensure good software engineering practices and stating the starting point of this project.

2.1 Requirement Analysis

In order to meet the high level success criteria as stated in the project proposal (see Appendix C), the following core requirements can be identified:

1. **Target language specification**: As part of the preparation, specify a set of features and corresponding type system for a C-like imperative language. This should at the very least include variables (as they require context lookup) in order to be able to demonstrate some of the difficulties of translating typing rules to Datalog.

2. **Source program encoding**: Moving on to implementation, create an pipeline which converts a given program into a suitable encoding which can be passed into a type checker written in Datalog.

3. **Type checking analysis**: Implement a type checker for the previously specified language using an incrementally evaluated dialect of Datalog.

4. **Program delta computation**: Another crucial part of making any incremental computation work is obtaining an (approximate) difference between the original and the modified program.

5. **Benchmarking**: Finally for evaluation, manually create some example programs and implement a standard type checker that the incremental type checker can be benchmarked against in order to demonstrate the usefulness of the project.

2.1.1 Choice of Tools

Based on the requirements above, there were multiple core choices to be made with regards to the underlying technology to be used for the project:

Component	Choice	Reasoning
Datalog Engine	**DDlog**	Though there have been incrementalised versions of Datalog for multiple decades (see [6] for example), only few are sufficiently maintained and support advanced enough features to be used for this project. I decided against IncA (used in [3]) as it does not use standard Datalog syntax and is too tightly integrated with the JetBrains MPS workbench. Instead I went with DDlog [7], which is based on Differential Dataflow [8] (a distributed data-parallel compute engine). Note that despite it being more widely supported than other options, there is still relatively little documentation available, so one of the challenges of this project was learning it with little resources to fall back on in case of questions (besides asking the DDlog creators).
Framework Language	**Rust**	It is possible to integrate DDlog as a library with either Rust, Java or C. However as DDlog is written in Rust it is easiest to integrate it with Rust. Rust also provides features such as pattern matching and macros that are useful for working with abstract syntax trees.
Parser	**lang-c**	I chose to go with an existing C parser (available as a Rust crate) as opposed to writing my own or using a parser generator as the focus of this project is not on parsing. In terms of using a specific parser, I compared multiple C parsers available for Rust (*lang-c* [9], a Rust *clang* wrapper [10], and *tree-sitter* Rust bindings [11]) and settled on *lang-c* because it was lightweight and provided the right level of output detail I needed. It is important to note that due to the modular nature of my project, the core functionality is independent of the parser so the parser can be changed (as long as its output is converted to the internal representation of the framework).
Development Environment	**VSCode**	Since I am using multiple languages it is advantageous to use an editor that isn't language specific but still provides various plugins for debugging and syntax highlighting.
Benchmark Library	**Criterion**	This library is a widely used statistics-driven benchmarking library for Rust enabling more meaningful results without having to run the benchmarks many times manually.

2.2 Target Language Specification

I initially thought I would be able to base my specification on an existing formalisation of the C language — there exist many different versions of formalised C semantics due to the official ISO/IEC International C Standard (see [12] for reference) being written in natural language, which introduces ambiguities and undefined behaviour that can be problematic when it comes to any formal reasoning about C programs. However, no formalisations turned out to be suitable as most of them either focus on runtime semantics (notably [13] for example) or are too complex for the needs of this project (for example see [14] or [15]).

As a result, I ended up defining my own type system for a simple imperative language called *Ceres*. As it is syntactically a subset of C, this has the advantage of being able to utilise existing C tools when needed (such as compilers to check for correctness) while also being in control of defining the static interpretation of the language.

2.2.1 Feature Selection

The first step was choosing a subset of C features that could feasibly be targeted in the time allowed, while also being substantial enough to write full programs in it. In order to facilitate iterative development, support for features was added in multiple stages: in the **core** stage I could focus on making a complete incremental type checker work for a minimal set of features, then in the **extension** stage *Ceres* could be extended to be a more complete language. An overview of all features is shown below (the syntax for each is standard C syntax):

- Core — **Primitive types**: `int` (not including `signed` and `unsigned` specifiers), `float`, `char`, and `void`.
- Core — **Binary operators**: standard arithmetic with +, −, /, and ∗.
- Core — **Variables**: allows me to deal with context lookup in Datalog.
- Core — **Functions**: not including storage or access specifiers.
- Extension — **Conditionals**: >, <, ||, &&, and == are added to the binary operators in order to enable `if`–`else` statements (using integers to represent true and false).
- Extension — **Loops**: together with conditionals, `while`-loops allow me to represent most kinds of control flow.

2.2.2 Type System

A type system consists of typing judgments that can be composed together to inference rules which can then be used to inductively infer properties about a program - in this case answering the question of whether the program is well-typed.

Typing judgments for *Ceres* will take the form $\Gamma, \Phi \vdash e : \tau$ where Γ is the variable context, Φ the function context (as functions are not first-class types in C), e an expression and $\tau \in \{\texttt{int}, \texttt{float}, \texttt{char}, \texttt{void}, \texttt{OK}\}$ its type under the given contexts. `OK` is introduced as a type in order to define the well-typedness of function definitions and statements which are otherwise untyped.

Figure 2.1 gives on overview of the **core** *Ceres* type system. A key difference to the standard C type system is that I made the choice to **not support implicit type conversion** — this makes it easier to introduce typing errors in the base version of the language.

$$\frac{n \text{ is an integer}}{\Gamma, \Phi \vdash n : \texttt{int}} \text{ (int-literal)} \quad \frac{n \text{ is a float}}{\Gamma, \Phi \vdash n : \texttt{float}} \text{ (float-literal)} \quad \frac{n \text{ is a character}}{\Gamma, \Phi \vdash n : \texttt{char}} \text{ (char-literal)}$$

$$\frac{\Gamma, \Phi \vdash e_1 : \tau \quad \Gamma, \Phi \vdash e_2 : \tau}{\Gamma, \Phi \vdash e_1 \ \texttt{op} \ e_2 : \tau} \text{ (binary-op)} \quad \frac{x \in \Gamma \quad \Gamma(x) = \tau}{\Gamma, \Phi \vdash x : \tau} \text{ (var)}$$

$$\frac{\Gamma, \Phi \vdash e : \tau}{\Gamma \cup \{x \mapsto \tau\}, \Phi \vdash \tau \ x \ = \ e : \tau} \text{ (assign)} \quad \frac{\Gamma, \Phi \vdash e_1 : \texttt{OK} \ldots \Gamma, \Phi \vdash e_n : \texttt{OK}}{\Gamma, \Phi \vdash e_1 \ldots e_n : \texttt{OK}} \text{ (unit)}$$

$$\frac{\Gamma \cup \{a_1 \mapsto \tau_{a_1}, \ldots, a_n \mapsto \tau_{a_n}\}, \Phi \cup \{f \mapsto (\text{return} : \ \tau_r, \text{args} : \ [\tau_{a_1}, \ldots, \tau_{a_n}])\} \vdash e : \tau_r}{\Gamma, \Phi\{f \mapsto (\text{return} : \ \tau_r, \text{args} : \ [\tau_{a_1}, \ldots, \tau_{a_n}])\} \vdash \tau_r \ f(\tau_{a_1} a_1, \ldots, \tau_{a_n} a_n)\{e\} : \texttt{OK}} \text{ (fun-def)}$$

$$\frac{f \in \Phi \quad \Phi(f).\text{return} = \tau \quad \forall i \in [1, n].\ \Gamma, \Phi \vdash a_i : \Phi(f).\text{args}[i]}{\Gamma, \Phi \vdash f(a_1, \ldots, a_n) : \tau} \text{ (fun-call)}$$

$$\frac{\Gamma, \Phi \vdash e_1 : \tau_1 \quad \Gamma, \Phi \vdash e_2 : \tau_2}{\Gamma, \Phi \vdash e1; \ e2 : \tau_2} \text{ (compound)} \quad \frac{\Gamma, \Phi \vdash \texttt{return} e_1; \ e2; : \tau}{\Gamma, \Phi \vdash \texttt{return} \ e : \tau} \text{ (compound-return)}$$

$$\frac{\Gamma, \Phi \vdash e : \tau}{\Gamma, \Phi \vdash \texttt{return} \ e : \tau} \text{ (return)} \quad \frac{\Gamma, \Phi \vdash e : \texttt{OK}}{\Gamma, \Phi \vdash e : \texttt{void}} \text{ (void-return)}$$

Figure 2.1: Overview of the *Ceres* type system.

2.3 Introduction to Differential Datalog

Datalog is a **declarative logic programming language** which is mainly used as query language for deductive databases. In reality Datalog an umbrella term for many different dialects, many of them enriched with features beyond the classic textbook form. In this case we will use DDlog-specific syntax and features for all the examples below but the general evaluation principles apply to any dialect. For a more comprehensive introduction see [16] and [17].

To illustrate the main ideas behind Datalog, consider as an example a set of side-by-side rowing boat races. How could we try to infer whether a certain crew is faster than another even if they don't all race each other directly? We start by defining the type signatures for appropriate **input and output relations**:

```
input relation WonRace(winner: Crew, loser: Crew)
output relation IsFaster(faster: Crew, slower: Crew)
```

A relation is a tuple which represent a fact that holds universally in the program. Input relations are passed to the program and can be used to infer new information represented

by output relations. In this case `WonRace` represents the result of a race while `IsFaster` is the information we want to infer (all pairs of crews where one crew is faster than another).

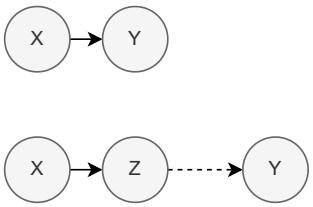
The example also demonstrates a useful feature DDlog is enriched by: **custom datatypes**. These will be useful for defining *Ceres* types later on, and can range from simple aliases to more complicated definitions with type constructors. The code below shows two possible ways we could define the `Crew` type:

```
% Aliasing:
typedef Crew = String

% Disjoint union type with fields:
typedef Crew = Beginner{name: String, ranking: signed<32>}
             | Intermediate{name: String, ranking: signed<32>}
             | Championship{name: String, ranking: signed<32>}
```

Next we have to define the **rules** used compute the output relations. Datalog rules take the form of a head tuple followed by a body of tuples — if each tuple in the body holds, so does the head. Figure 2.2 shows that the relation `IsFaster`(X, Y) holds in two possible scenarios: either crew X has directly won a race against Y, or it has won a race against some other crew Z which is faster than Y.

```
1 IsFaster(X, Y) :-
2     WonRace(X, Y).
3
4 IsFaster(X, Y) :-
5     WonRace(X, Z),
6     IsFaster(Z, Y).
```



$$\frac{\texttt{WonRace}(X, Y)}{\texttt{IsFaster}(X, Y)}$$

$$\frac{\texttt{WonRace}(X, Z) \quad \texttt{IsFaster}(Z, Y)}{\texttt{IsFaster}(X, Y)}$$

Figure 2.2: Datalog example rules with graph and logic interpretations for comparison.

Note that if we interpret the scenario as a graph where a direct edge from Y to X represents X being faster than Y, this computes a transitive closure over the graph. Also note the logical interpretation of each rule, as this is what enables the translation of typing rules later on.

We will now consider a concrete example of six different crews racing each other in order to illustrate **naive bottom-up evaluation**. Figure 2.3 visualises the example as a graph.

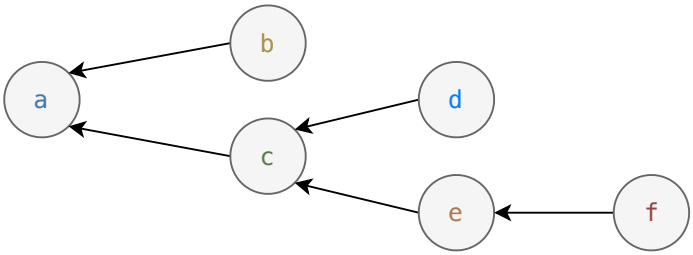


Figure 2.3: Example scenario graph visualisation.

Evaluation proceeds by iterating over all existing relations and generating new ones until it reaches a **fixpoint**. Figure 2.4 shows the relations generated for the example scenario in each

step. In practice there are many optimisations techniques which are used to make evaluation more efficient, but from the perspective of the programmer the principle of interpreting programs bottom-up (as opposed to having functions which pass arguments top-down as is the case with most standard programming languages) still holds.

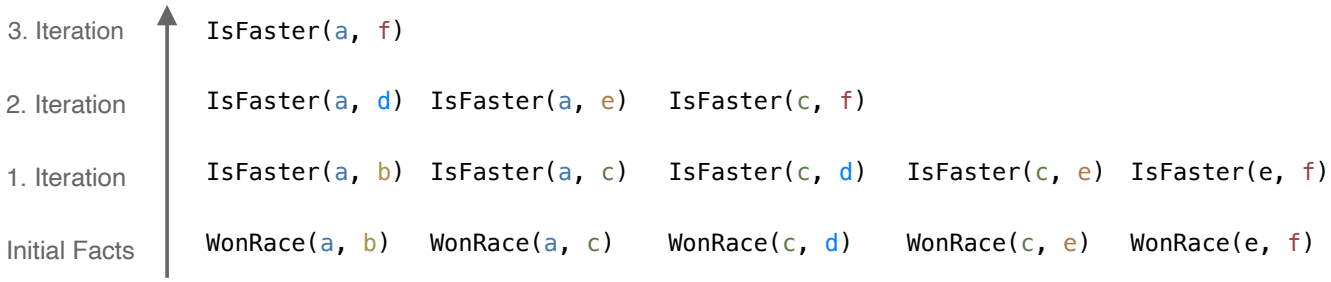


Figure 2.4: Naive bottom-up evaluation of example scenario.

Another important feature that we will need is is **negation**. In this case, as figure 2.5 shows, we can use it to find the fastest boat (or boats if we only had a partial ordering or disjointed components).

```
1 output relation Undefeated(crew: Crew)
2
3 Undefeated(X) :-
4     not IsFaster(_, X).
```

$$\frac{\forall y.\, \neg \texttt{IsFaster}(y, X)}{\texttt{Undefeated}(X)}$$

Figure 2.5: Example rule using negation and corresponding logic interpretation.

2.4 Software Engineering Methodology

Throughout the project I made sure to maintain a good software engineering approach, which can be demonstrated through multiple aspects:

- **Development model**: In order to make consistent progress during the project I used an iterative development model with multiple phases which are described in figure 2.6. I re-evaluated my work plan in regular intervals to adapt to changing circumstances.
- **Unit testing**: I utilised Rust's inbuilt functionality for automated test modules to write unit tests. This enabled me to check whether components and functions were working as expected in isolation during development without requiring me to have finished a complete pipeline.
- **Code quality**: To ensure consistent code formatting I used the formatter included with the VSCode Rust plugin. I also made sure my code was readable and documented through many comments.
- **Version control**: I maintained separate GitHub repositories for both my project code and write-up and made sure to back up through git regularly.
- **Licensing**: I chose to license my project under the MIT license to make it available open source. I then also ensured any libraries and any code I used for my evaluation dataset was also available open source.

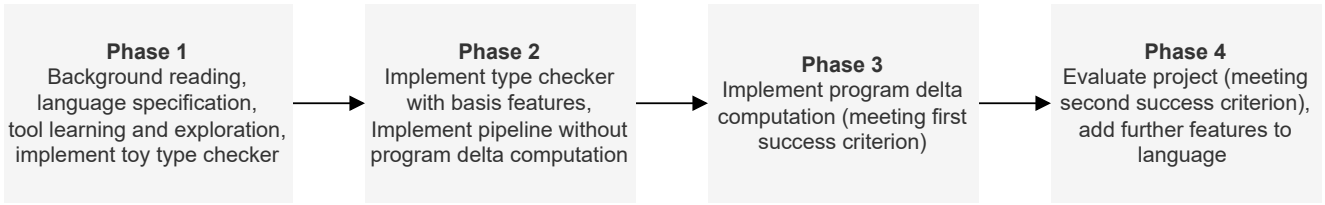


Figure 2.6: Overview of development phases.

2.5 Personal Starting Point

At the start of the project, I had no previous experience or specific knowledge related to the area of incremental type checking at all, so I had to spend a lot of time at the start reading up on it. Multiple courses in the Tripos are relevant to this project: I gained some familiarity with logic programming in the *Part IB Prolog* course, however Prolog is quite different to Datalog as it differs fundamentally in its evaluation, and I had to learn Differential Datalog before starting my implementation. *Part IB Programming in C* was useful for providing a background in C while *Part IB Semantics of Programming Languages* and *Part II Types* where helpful in formalising a subset of it. *Part IB Compiler Construction* and *Part II Optimising Compilers* are also relevant. Finally I had to learn Rust from scratch as I had never used it before this project.

2.6 Summary

- Appropriate tools have been chosen for the project based on project requirements, notably I decided to use **DDlog** as a dialect of Datalog which will be automatically incrementalised, and **Rust** as the implementation language for the rest of the framework.
- A **type system has been formally specified** for syntactic subset of C. This will form the basis of the type checking analysis implemented in the next chapter.
- The key takeaway from the Datalog introduction is the approach to solving problems in Datalog by **specifying input and output relations**, and that it is evaluated **bottom-up**.
- **Good software engineering practices** have been used throughout the project.

Chapter 3

Implementation

The following chapter describes the implementation of the *Cerium* type checker. It starts by providing an overview of the project architecture: a pipeline of consisting of multiple components, each implementing one part of the functionality of the project. This is then followed by more detailed look at each component and the challenges encountered during implementation. Starting from an input program, I explain how a program is converted to DDlog relations through abstract syntax tree (AST) flattening, how we can then find the difference between two programs through tree differencing in order to specify the changes to the DDlog incrementalisation engine, and finally how typing rules can be translated to DDlog in order to perform the core functionality of the project, type checking.

3.1 Pipeline Overview

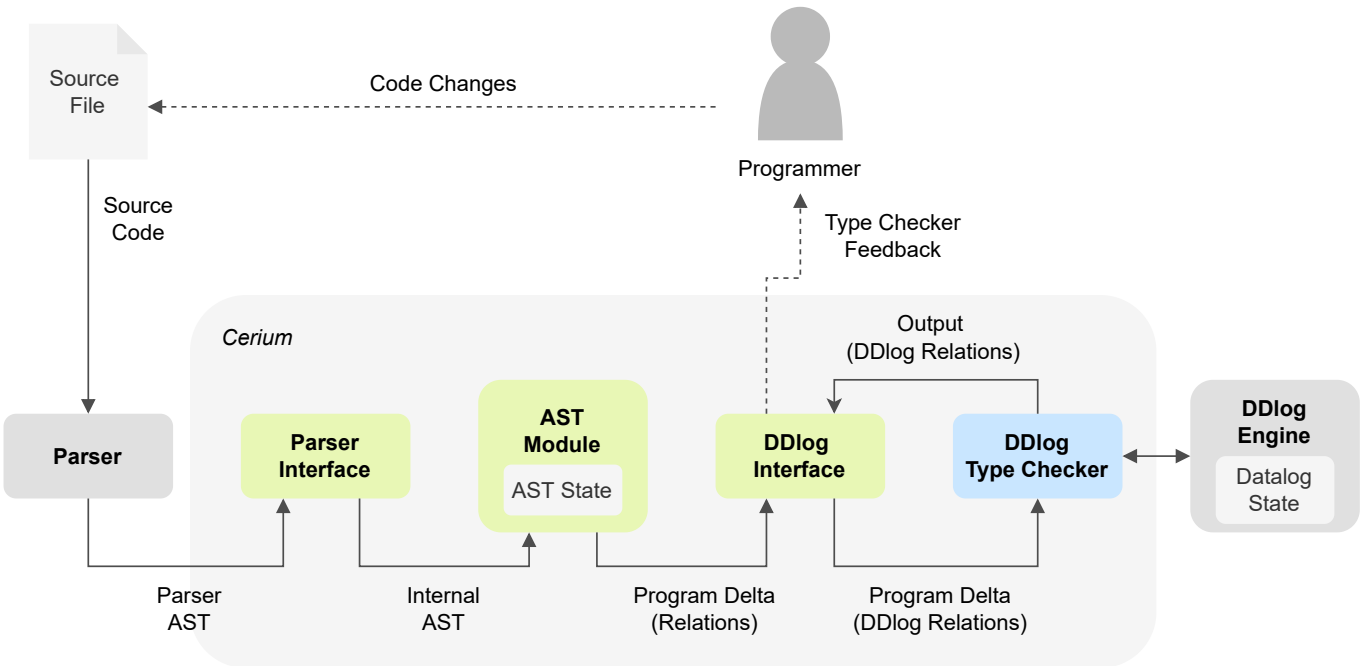


Figure 3.1: Architecture overview (including interaction with a user).

Figure 3.1 provides an overview of the *Cerium* pipeline. While designing in pipeline I made sure components could easily be extended or replaced by connecting them through interfaces. The type checker is run as command line tool: once it is started by a user (giving a file name as input), it continuously cycles through the pipeline on every file save, providing feedback on whether the program is "correctly typed ✓" or contains a "typing error ✗".

The project architecture is also reflected in the repository structure shown in figure 3.2: the colours of different files correspond to the ones used in the pipeline diagram (with gray used to indicate folders). In order to use the DDlog type checker as a library from within the framework, `type_checker.dl` is compiled into an binary Rust executable that can be linked against any other Rust project.

```
cerium/
├── framework/
│   ├── benches/ ..... evaluation dataset and benchmark code
│   ├── convert_variant_derive/ ..... macro for conversion to DDlog relations
│   ├── src/
│   │   ├── ast.rs ..... AST data structure and differencing
│   │   ├── ddlog_interface.rs ..... interface to DDlog type checker
│   │   ├── definitions.rs ..... AST relation definitions
│   │   ├── lib.rs ..... top-level functions
│   │   ├── main.rs ..... executable entry-point
│   │   ├── parser_interface.rs ..... parser output conversion to relations
│   │   └── standard_type_checker.rs .. recursive descent type checking for benchmarking
│   └── tests/ ..... code examples for unit tests
└── type_checker.dl ..... type checking analysis
```

Figure 3.2: Repository overview (only including core files and folders).

3.2 Abstract Syntax Tree Flattening

Recall that the first step in solving a problem using Datalog is converting your problem input into relations. Hence the first step in the framework pipeline is encoding the input program into appropriate Datalog relations that will represent the set of all facts that we know about the program.

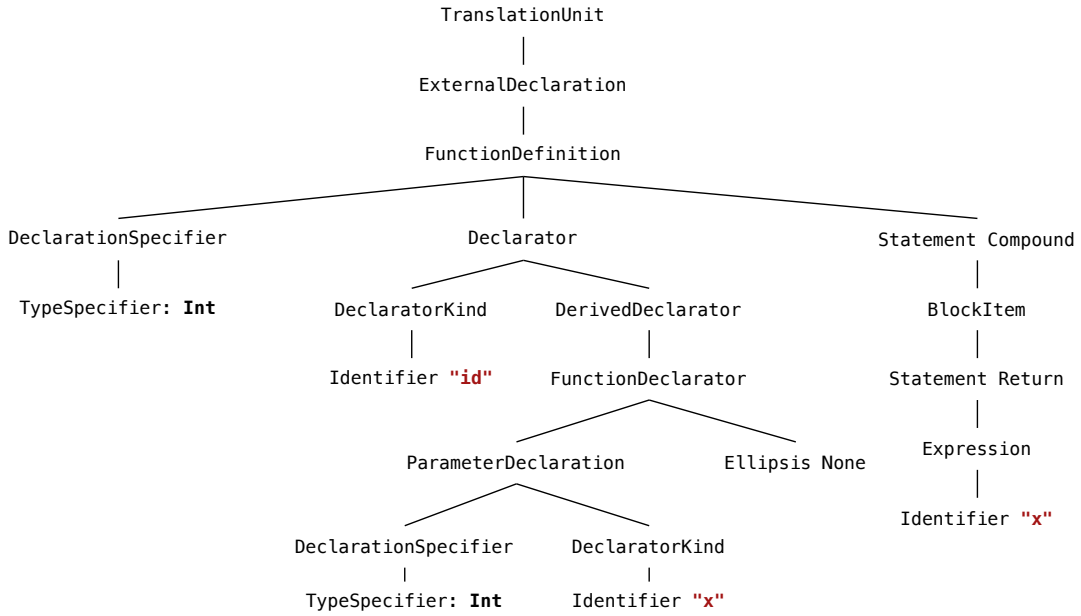


Figure 3.3: Output of the *lang-c* parser for the identity function visualised as a tree.

Consider the following simple function as an example program to encode:

```
int id(int x) {
    return x;
}
```

Figure 3.3 shows the output we get by parsing this code using the *lang-c* parser library. This abstract syntax tree (AST) is the starting point for the transformation and the goal is to flatten it down to relations. For this note that a tree can be represented by assigning each node a unique identifier and then implicitly linking its children by storing a list of corresponding child identifiers in each node. This means that we can represent each AST node type A with n children (named a_1 to a_n) as a relation with the following type signature:

$$A(\text{id} : \texttt{ID}, a_1 : \texttt{ID}, \ldots, a_n : \texttt{ID})$$

Any further information such as variable or function names can be added as another tuple element of type `string`, and varying amounts of children, such as in the case of argument lists for example, can be represented by a vector of identifiers.

It would be possible to define a relation type for each AST node type in the *lang-c* parser specification, however as I didn't want to be dependant on a particular parser I opted for defining my own internal representation and writing a parser interface which converts the parser output to it instead. Figure 3.4 shows the simplified internal AST for the example above together with the corresponding program relations. Appendix A lists the type signatures of all supported program relations.

Since the time complexity of many of the following steps such as tree differencing and type checking depends on the tree size (and hence number of input relations), this approach also has the advantage of being able to make the AST simpler.

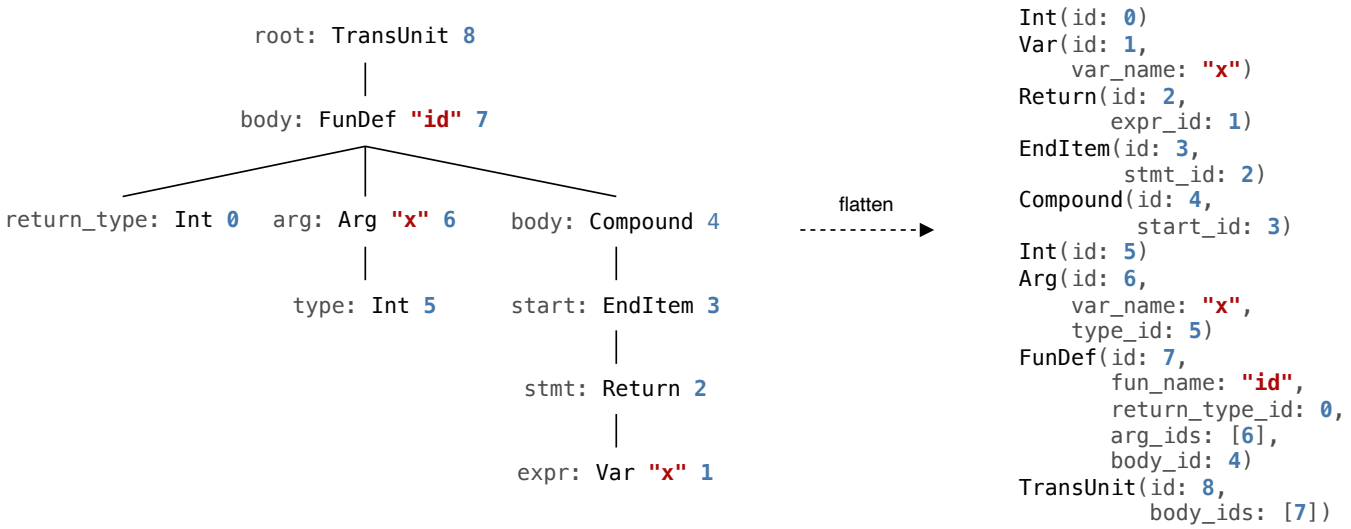


Figure 3.4: Internal AST and program relations for the identity function.

The parser interface obtains the internal tree representation by recursively traversing the parser output once and simultaneously constructing both a new tree (see the struct defintions for the AST data struture defined in the AST module below) as well as the relations (defined as an `AstRelation` enum) corresponding to each node.

```
pub struct Tree {
    arena: HashMap<ID, AstNode>,
    max_id: ID,
    root_id: ID,
}

pub struct AstNode {
    node_id: ID,
    relation: AstRelation,
    children: Vec<ID>,
}
```

Identifiers are simply numerical values which are allocated by keeping track of the largest identifier used so far and assigning the next possible value every time a new node is created. The final flattening step traverses the map of tree nodes in order to extract the relations into a set.

3.3 Structural Tree Differencing

Having obtained an internal AST with corresponding input program relations, we can now pass the relations to the DDlog type checker. However we only pass all relations to the type checker during the initial pass. Every successive type checking pass afterwards we need to compute a program delta in order to only pass relations that have changed since the previous pass in order for the DDlog engine to be able to incrementalise the analysis. We can compute the program delta by identifying the structural difference between the previous and modified AST.

This adds a significant overhead to the program: if we consider adding a node, deleting a node, updating a node and moving a node as possible edit actions that need to be identified trying to find an accurate delta becomes a NP-hard problem (see [18]). We therefore only consider **additions** and **deletions** (with updates being represented by a deletion followed by an addition). We also make some assumptions about the nature of changes being made to a program which helps to determine an **approximate** delta for the sake of performance.

Program differencing can be seen as as a type of comparative program analysis, and as such we have to be able to to guarantee **safety** properties as the type checking analysis performed using the approximate differencing results need to be safe. The following should hold:

$$\text{real } \Delta \subseteq \text{computed } \Delta$$

In other words, it is fine if we over-approximate unchanged code as changed, but all changed code definitely needs to be marked as changed. As long as the this safety property holds, there then exists an inherent trade-off between the **exactness** of the delta and the **performance** of the algorithm in terms of time complexity.

Exactness	Performance
The more exact the delta is, the more effective the incremental computation that builds on it becomes. However the more fine-grained the comparison between ASTs, the more complex the algorithm has to be in order to identify all the differences, which is to the detriment of performance.	The faster and less complex the algorithm is, the better the overall performance of the type checker becomes. However this often comes at the cost of sacrificing exactness by only approximating some of the differences.

There are multiple algorithms proposed for structural differencing in literature (such as [19], [20] and [21] to name notable examples), however in this case I have opted to devise my own algorithm as we are not aiming to be able to find the difference between any kind of tree, and can therefore make assumptions about the general structure of the trees as well as about the type of changes we expect to occur more often, leading to heuristics which can simplify the algorithm.

3.3.1 Program Change Assumptions

Based on my experience as a programmer there are multiple assumptions that can reasonably be made about incremental changes made in the process of building up a program:

	Assumption	Consequence
1	On average in a given change the programmer is more likely to add or move around function declarations rather than rename a function.	We can approximate any functions with a new name as being completely new functions, any function names that do not appear anymore as deleted, and then only look at subtrees in more detail if the function name at its root has previously occurred.

2	On average the program will grow, i.e. a change inside a program is more likely to be an addition rather than a deletion.	When comparing items inside a function compound, if two nodes do not match, instead of assuming a node in the original tree has been deleted we assume that a new node has been inserted between the previous and this node.
3	In general the programs won't use many levels of nested scopes.	We do not try to go beyond the top-level compound when comparing items inside a function compound. This means we do not try to find any fine-grained changes inside statements and just consider the binary question of whether the statement contained in the item has changed in any way or not.

3.3.2 Overview of Algorithm

This section presents an overview of the tree differencing algorithm implemented in the AST module of the pipeline.

- **Algorithm input**: Two ASTs, from now on referred to as t_p (previous) and t_n (new).
- **Algorithm output**: A modified version of t_p, t_m, that is structurally identical to t_n but all node identifiers are consistent with the initial version of t_p. Two sets of AST relations, S_d (containing all relations to be deleted from the running DDlog type checking program) and S_i (containing all relations to be inserted to the running DDlog type checking program).

The following terms will be used as terms referring to a collection of actions being performed:

- **Insert**: A new node is added to the AST, being allocated the largest identifier not yet in use. Any parent nodes are replaced in order to link to this node. A corresponding AST relation is added to the insertion set.
- **Delete**: An existing node is deleted from the AST, its identifier being de-allocated if it was the largest identifier at this point. Any parent nodes are replaced in order to no longer link to it. A corresponding AST relation is added to the deletion set.
- **Replace**: An existing node is deleted from the AST, but instead of de-allocating the identifier the same identifier is used for added a new node to the AST (including any necessary changes such as parent or child replacements). The AST relation corresponding to the deleted node is placed in the deletion set while the relation corresponding to the added node is placed in the insertion set.

Starting at the root node of both trees (the translation unit) we traverse both trees in the following manner in order to compare them:

1. Each child of a root node is a function definition. This is where assumption 1 comes in play: iterate through both sets of children comparing function names. If two function names match, continue to step 2. Else keep track of which names occur in t_p and t_n.

2. For each pair of function definition nodes with matching names:

   (a) Compare the child node corresponding to the return type and replace the node if they do not match.

   (b) Compare all children corresponding to the list of parameters, replacing, inserting and deleting nodes when necessary.

   (c) Compare the function compounds item by item. This utilises both assumption 2 and 3. An example for this is shown in figure 3.5 - deletions are more expensive.

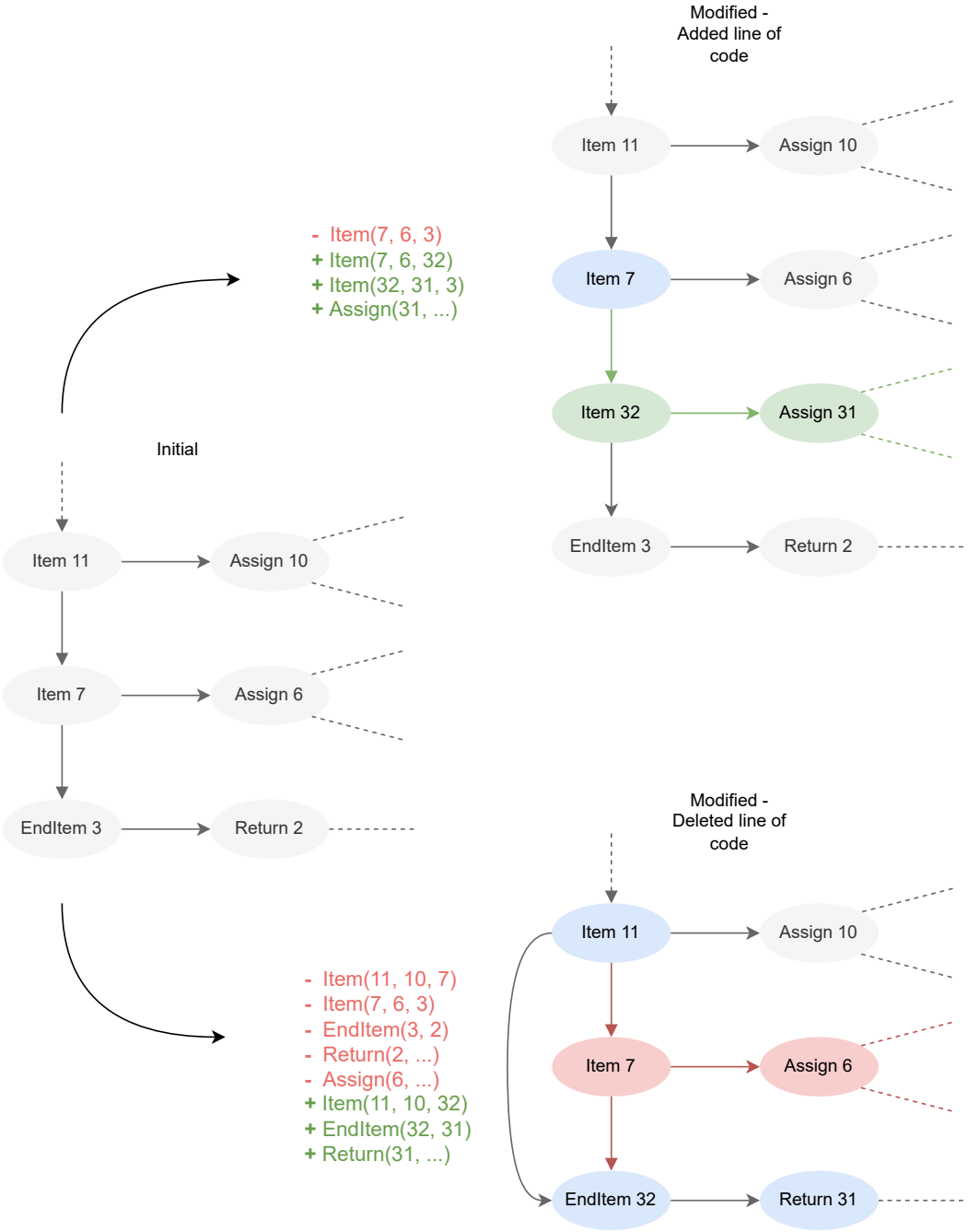


Figure 3.5: Example diagram of a statement addition vs. deletion (identifiers are arbitrary).

3. Delete all function subtrees only appearing in t_p. Insert all function subtrees only appearing in t_n. If the list of translation unit children has changed in any way, replace the translation unit node to reflect the changes.

After the algorithm concludes, t_m is retained as program state in order to be used as t_p during the next iteration of the pipeline while S_d and S_i are passed to the DDlog interface which in turn passes them them to the DDlog type checker.

3.4 Translating Typing Rules into Datalog

The final part of the project pipeline is the type checking analysis implemented in DDlog. Since the DDlog engine automatically incrementalises the analysis, we can assume we are given a full set of input relations representing a program and then write output relation rules in order to infer whether the program is correctly typed or not. We can define the program with translation unit with identifier x as its root as correctly typed if and only if there exists a derivation of the output relation `OkProgram`(x).

The general approach for deriving `OkProgram`(x) consists of having a `Typed`(x, τ) output relation for each AST input relation. The `Typed`(x, τ) relations are then iteratively built up starting with relations corresponding to the bottom of the AST. A parent node is only well-typed is the relations corresponding to its children are well-typed. However, the translation from theoretical typing rules to Datalog typing rules is not completely straightforward. This section explains multiple techniques that have to be used to enable the translation.

We continue using the identity function code from the previous sections as an example (the internal AST with each node representing an input relation is repeated in figure 3.6).

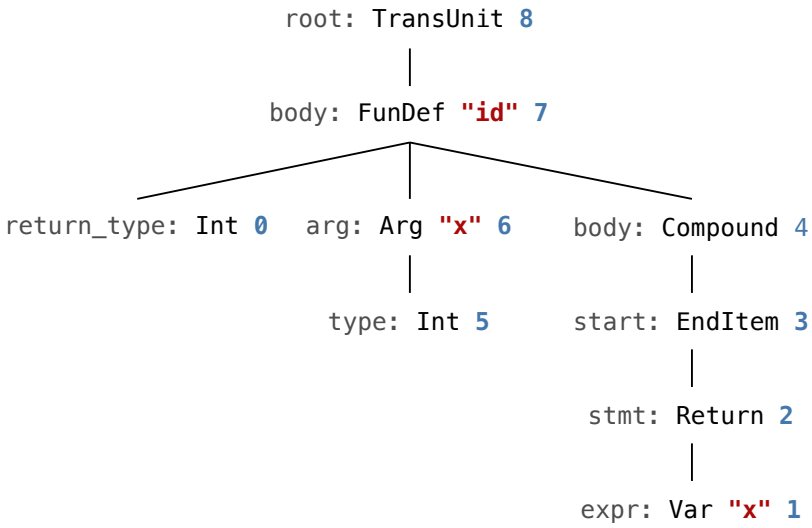


Figure 3.6: Internal AST for the identity function.

Consider the type derivation we obtain by applying the typing rules defined in section 2.2.2:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x \in \{x \mapsto \texttt{int}\} \quad \{x \mapsto \texttt{int}\}(x) = \texttt{int}}{\{x \mapsto \texttt{int}\}, \{\text{id} \mapsto (\text{var} : \texttt{int}, \text{args} : [\texttt{int}])\} \vdash x \ : \ \texttt{int}}\ (\text{var})}{\{x \mapsto \texttt{int}\}, \{\text{id} \mapsto (\text{return} : \texttt{int}, \text{args} : [\texttt{int}])\} \vdash \texttt{return}\ x; \ : \ \texttt{int}}\ (\text{return})}{\emptyset, \{\text{id} \mapsto (\text{return} : \texttt{int}, \text{args} : [\texttt{int}])\} \vdash \texttt{int}\ \ \text{id}(\texttt{int}\ x)\{\ \texttt{return}\ x;\ \} : \texttt{OK}}\ (\text{fun-def})}{\emptyset, \emptyset \vdash \texttt{int}\ \ \text{id}(\texttt{int}\ x)\{\ \texttt{return}\ x;\ \} : \texttt{OK}}\ (\text{unit})$$

Now compare the above to the type derivation we obtain as a result of the Datalog evaluation in figure 3.7 (the proof tree split is into multiple parts — which roughly correspond to each line in the theoretical derivation — for readability). Note that the `OK` type is implicit in Datalog in that all derived tuples will always be `OK` if they don't contain an explicit type. Expressions that have an explicit type take one of the following values:

```
typedef Type = VoidType
             | IntType
             | FloatType
             | CharType
```

The following subsections detail the key concepts needed to understand the differences between between the two proof trees as highlighted in the Datalog derivation:

- grounding,
- errors in variable-length lists,
- and context search.

It is important to note that every type system feature added to a language could potentially require different techniques on a case by case basis in order to be represented in Datalog, but the ones outlined below are sufficient for *Ceres*.

3.4.1 Grounding

As relations are implicitly linked, we don't have a concrete data structure to traverse. Instead we have to ground each typing rule — i.e. make sure it can be derived by conditioning it on the existence of an input relation with a matching identifier. Otherwise the typing relations would consider the set of all possible programs as input. Since this set is of infinite size, evaluation wouldn't terminate.

Axioms (rules representing the leaves of the proof tree) are derived directly from input relations. In this case we just have primitive types as axioms:

```
TypedLiteral(id, VoidType) :- Void(id).
TypedLiteral(id, IntType) :- Int(id).
TypedLiteral(id, FloatType) :- Float(id).
TypedLiteral(id, CharType) :- Char(id).
```

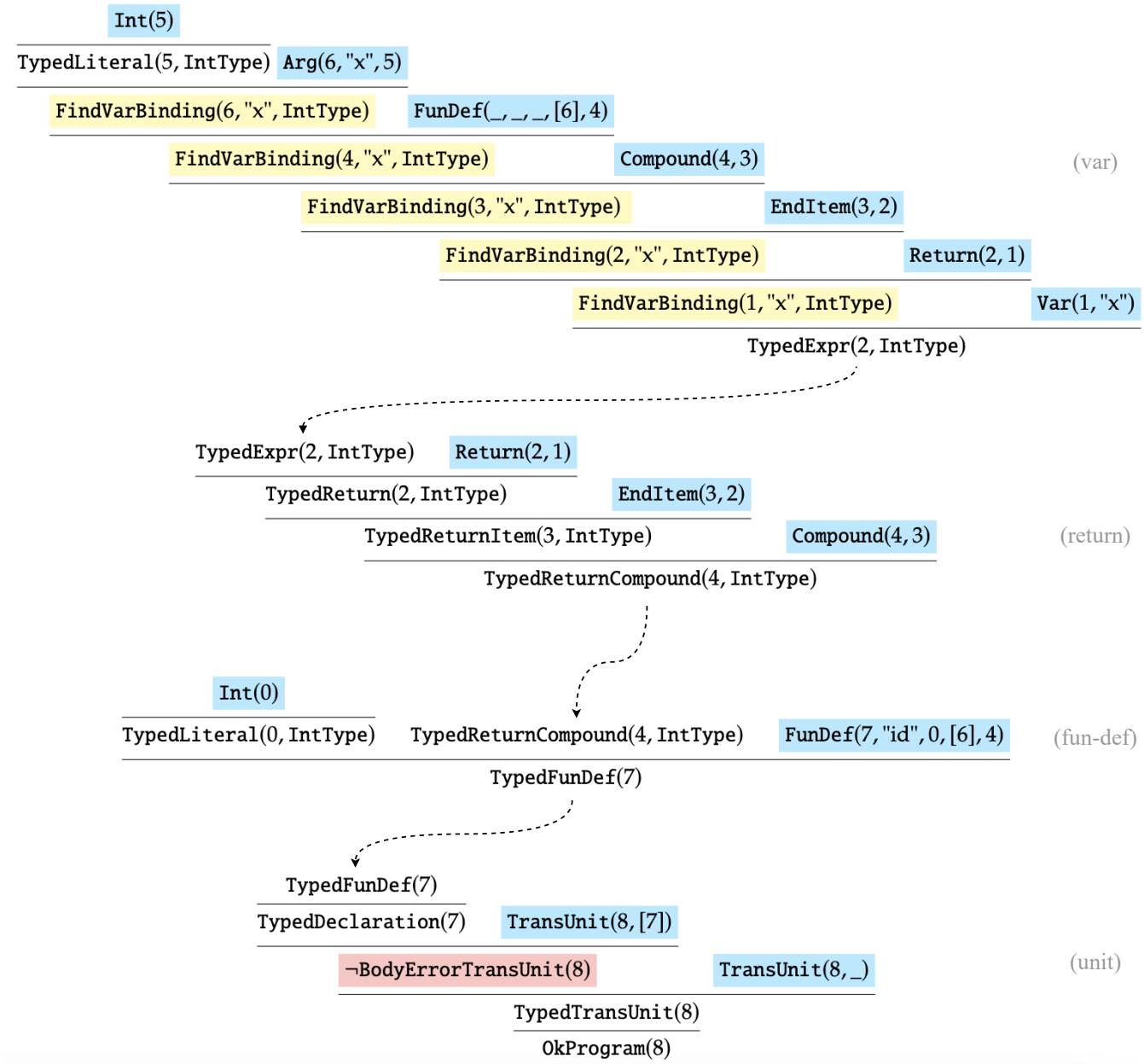


Figure 3.7: Datalog type checker evaluation visualised as proof tree.

Rules further down the tree build on both input relations and already derived typing relations. As an example consider the typing relation for function definitions:

```
TypedFunDef(id) :-
    % Ground relation with input AST relation.
    FunDef(id, fun_name, return_type_id, arg_ids, body_id),
    % Check return type matches type returned by function body.
    TypedLiteral(return_type_id, return_type),
    % Check function body is well-typed.
    TypedReturnCompound(body_id, return_type).
```

3.4.2 Errors in Variable-Length Lists

If a relation contains an element of type `Vec < ID >` that can take different lengths (which happens in case an AST node has an unknown number of children) — such as in the case of number of function declarations in a translation unit or number of arguments passed to a function — we cannot get the identifiers in the list as we do with separate named elements. We also cannot deduce that there was an error based on the absence of relevant relations, so instead we have to explicitly check that there has not been an error.

For the first problem we can use an inbuilt DDlog function called `FlatMap` which enables you to convert vectors into relations that can be referred to by variables at runtime. For the second problem we can utilise negation. This can be illustrated with the typing relation for a translation unit:

```
TypedTransUnit(id) :-
    TransUnit(id, _),
    not BodyErrorTransUnit(id).

BodyErrorTransUnit(id) :-
    TransUnit(id, body_ids),
    var body_id = FlatMap(body_ids),
    not TypedDeclaration(body_id).
```

Effectively the above says that the translation unit is well-typed if there is no error in it. Then an error in a translation unit is defined to exist if at least one of the function declarations is not well-typed.

3.4.3 Context Search

Context search combines two ideas introduced in [3], co-functional dependencies and context fusion into one. If you are trying to automatically generate Datalog code based on typing rules, it easier to apply them one after another however since I am writing the type checker by hand it is possible to achieve more concise and human-readable code by directly applying the core ideas in one step.

Notice that a big difference between typing relations and typing judgments is that the **relations do not contain contexts**. During a standard top-down type checking analysis whenever you need to look up a variable or function name, you can simply look it up in the context which has been constructed while traversing the tree towards that node. However this is not possible in Datalog because we are building the typing derivation from the bottom-up. From the perspective of the node which requires a context lookup, there are multiple possible types a variable could have and infinitely many possible type signatures a function could have.

Co-functional dependencies are based on the observation that a context is an input which can always be uniquely determined given the AST and can therefore be computed from scratch at any point of the tree. Context fusion is then the observation that a lookup in a context is also effectively a search. Combining both we come to the conclusion that each time we need to lookup a type during a Datalog derivation, we can effectively **perform a top-down context construction** by searching for variable or function usage starting from each variable or function definition.

In practice this means having the following relations in order to encode the context search:

```
relation FindVarBinding(location: ID, var_name: string, type: Type)
relation FindFunBinding(location: ID, fun_name: string, type: Type)
```

In case of variables we need rules for each possible combination of child and parent input relation in order to iteratively build up the path from variable definition to usage (if such a path exists). This does mean that some `FindVarBinding` tuples will be generated despite never being used because all possible paths to any scope where a variable might be used will be computed. However, variables in the same scope do share parts of the derivation tree so it is not a separate construction for each.Variables can be defined through assignment or as function arguments:

```
% Case: found assignment and names match.
FindVarBinding(id, var_name_found, t) :-
    Assign(id, var_name_found, type_id, expr_id),
    TypedLiteral(type_id, t).

% Case: found function definition so need to check arguments.
FindVarBinding(id, var_name, t) :-
    FunDef(_, _, _, arg_ids, id),
    var next_id = FlatMap(arg_ids),
    FindArgVarBinding(next_id, var_name, t).

% Case: found in argument declaration and names match.
 FindArgVarBinding(id, var_name_found, t) :-
    Arg(id, var_name_found, type_id),
    TypedLiteral(type_id, t).

% Otherwise continue searching.
```

For functions we just need to search all function definitions starting from the translation unit:

```
% Case: need to search for definition in translation unit.
FindFunBinding(id, fun_name, return_type, arg_type_ids) :-
    TransUnit(id, body_ids),
    var next_id = FlatMap(body_ids),
    FindFunBinding(next_id, fun_name, return_type, arg_type_ids).

% Case: found right function.
FindFunBinding(id, fun_name, return_type, arg_type_ids) :-
    FunDef(id, fun_name, return_type_id, arg_type_ids, _),
    TypedLiteral(return_type_id, return_type).
```

3.5 Summary

- The *Cerium* type checker can be seen as a **pipeline** of multiple stages which are executed in repeated passes every time the input program is edited.
- The first stage consists of transforming the parser AST into an **internal AST** which in turn can be directly **flattened into relations** which implicitly preserve the tree structure by linking to child relations through identifiers.
- After the initial pass, each pass also includes a **tree differencing** stage where be obtain an approximate program delta using based on **assumptions** made about the type of changes usually made to a program. This enables the DDlog engine to automatically incrementalise the final stage.
- The final stage type-checks the program by applying Datalog typing relations to the input AST relations. The Datalog relations are obtained from the *Ceres* type system by applying techniques such as **grounding**, **error detection in lists** and **context search**.

Chapter 4

Evaluation

This chapter provides evidence for the success of the project. It starts by outlining the approach I used for benchmarking *Cerium* and then goes on to demonstrate that it meets the definition of an incremental type checker. Next, I discuss some of the limitations of my implementation and put them in a wider context of moving from a standalone tool to functionality incorporated into an IDE. Finally, I review the requirements and success criteria set out at the start of the project.

4.1 Methodology

Several elements are needed in order to evaluate this project: a non-incremental baseline the incremental type checker can be compared against, a dataset of *Ceres* programs and small program changes, and a benchmarking library which can measure execution time.

4.1.1 Baseline

As a baseline I implemented a non-incremental **recursive descent** type checker for *Ceres* in Rust. Recursive descent is the standard approach used for type checking and consists of recursively traversing the AST starting from the root and checking each node. For example, consider part of the function used to check statements which deals with function calls:

```
fn type_check_statement(
    node: AstRelation,
    ast: &Tree,
    var_context: HashMap<String, Type>,
    fun_context: HashMap<String, FunType>,
    current_fun: String,x
) -> (Type, HashMap<String, Type>) {
    // Pattern-match over all AST nodes that can appear inside a function.
    match node {
        AstRelation::FunCall { id: _, fun_name, arg_ids, } => {
            // Retrieve function type from context that has been built up.
            let fun_type = fun_context.get(&fun_name).unwrap();
            let fun_types = fun_type.arg_types.clone();
```

```
            // Recursively check that all arguments are well-typed.
            let mut arg_index = 0;
            for arg_id in arg_ids {
                let (arg_type, var_context) = type_check_statement(...);
                // Argument and parameter types don't match.
                if fun_types[counter] != arg_type {
                    return (Type::ErrorType, var_context);
                }
                arg_index = arg_index + 1;
            }
            // Function call is well-typed.
            return (fun_type.return_type.clone(), var_context);
        }
        // Similar code for all other types of AST nodes.
        ...
    }
}
```

4.1.2 Dataset

The second element required for evaluation is a dataset of example programs and changes. Besides writing some short programs myself, I also created some examples based on solutions to Project Euler problems available as part of *The Algorithms*, an open-source algorithm library hosted on GitHub.

Since Project Euler solution code often does not require any advanced C features due to the mathematical nature of the problems, a lot of the code only requires minimal rewriting in order to just utilise features supported by *Ceres* — mostly changing it to only use a subset of primitive types and rewriting `for`-loops to `while`-loops. Note that the runtime meaning of all the example programs is irrelevant as we are only interested in the correctness of static properties. See below an example for the type of code we obtain after rewriting:

```
int problem1(void) {
    int t = 10;
    int sum = 0;
    while (t) {
        t = t - 1;
        int p = 0;
        int sum = 0;
        int N = 1000;
        p = (N - 1) / 3;
        sum = ((3 * p * (p + 1)) / 2);
        p = (N - 1) / 5;
        sum = sum + ((5 * p * (p + 1)) / 2);
        p = (N - 1) / 15;
        sum = sum - ((15 * p * (p + 1)) / 2);
    }
    return sum;
}
```

Larger programs were built up by using smaller examples, resulting in 10 initial base programs (ranging from 8 to 2592 AST nodes in terms of size) with possible program changes for each of them. Each change is small enough to simulate the types of changes that realistically occur in order for analysis to be rerun in an IDE. Number of AST nodes is used as a metric for program size as it is code style independent (in contrast to number of code lines) and therefore more accurately reflects the size in terms that matter for static analysis.

4.1.3 Testing Environment

Benchmarks were performed using the *criterion* benchmarking library for Rust. The library enables you to measure the time it takes for a function to run. In order to obtain statistically significant results each benchmark is split into two phases: warmup and measurement. During the warmup the function is repeatedly executed in order to allow the CPU and OS caches to adapt to the workload. Then during the measurement stage the function is timed in samples, where each sample contains multiple executions of the function. This allows the library to use linear regression on the sample measurements with respect to the sample size in order to obtain an approximate execution time for the function.

The values used for the graphs in this chapter are the **means of the probability distribution functions for execution time** obtained by running 100 measurements. All performance measurements were collected on a laptop with the following specifications:

- Processor: 1.4 GHz Quad-Core Intel Core i5
- Memory: 16 GB 2133 MHz LPDDR3
- OS: macOS Big Sur v11.6.4
- Software versions: *rustc* v1.56.0, DDlog v0.49.0, *criterion* v0.3

4.2 Type Checking Performance

Recall the definition of an incremental type checker as defined in the introduction: **a type checker, which given a program delta, aims to only perform an amount of computation that is proportional to the size of the program delta, instead of the size of the program**.

We can show that this holds for a given type checker by fixing a small program delta and then measuring the time it took to perform the type checking analysis while increasing the program size - we expect the execution time to remain approximately constant if the type checker is incremental.

Figure 4.1 demonstrates that this is indeed the case for *Cerium*. For this benchmark I kept some code constant in every file and introduced one change to it: changing an assignment from `int` sum = 0; to `float` sum = 0; . Then I added more unrelated code that was unaffected by the change to the program (up to 2592 AST nodes) and ran both the standard type checker and *Cerium* on it (the latter after it already performed an initial pass on the unchanged program in order to build up the internal program state).

We can see that while the execution time for the standard type checker increases with increasing program size, it remains close to constant (at around 47 microseconds) in the incremental case (only performing worse on the smallest program size of 162 nodes).

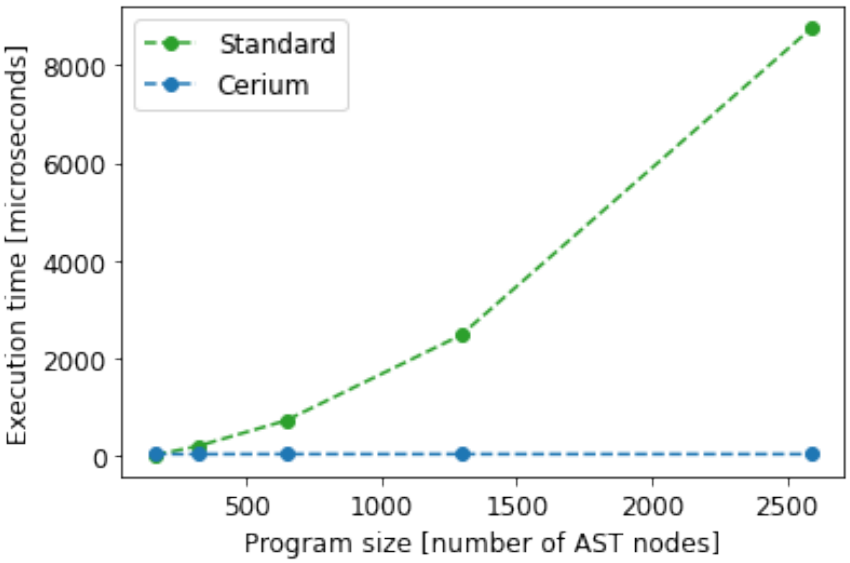


Figure 4.1: Performance comparison based on constant delta and increasing program size.

Furthermore, we can demonstrate that the *Cerium* type checking analysis is faster than the standard one for a range of different changes affecting different parts of a program. The table below gives an overview of changes separately introduced to a small program and figure 4.2 shows that *Cerium* outperforms the standard type checker in each case.

Program	Description
Initial program	Correct program with multiple functions consisting of 30 AST nodes.
Change 1	Parameter type of one function is changed - *throws typing error*.
Change 2	Return type of one function is changed - *throws typing error*.
Change 3	Argument type of one function call is changed - *throws typing error*.
Change 4	New function is added to the program - *correctly type-checks*.
Change 5	One of the functions in the program is deleted - *correctly type checks*.
Change 6	One of the functions in the program is renamed - *correctly type checks*.
Change 7	One of the function parameters is renamed - *correctly type checks*.

Note how some of the execution times also reflect assumptions we made during tree differencing. For example consider change 6, renaming a function — as we identify functions by their names during tree differencing, renaming from the perspective of the type checker becomes a combination of deleting and adding a function. In general the type checker performing well indicates that the tree differencing algorithm found an appropriate delta. If it did not, we would get times closer to the time it took to perform the initial pass.

Differing times for the standard type checker (see change 1 and 2 for example) are explained by the fact that it abandons the recursive traversal whenever an error is found.

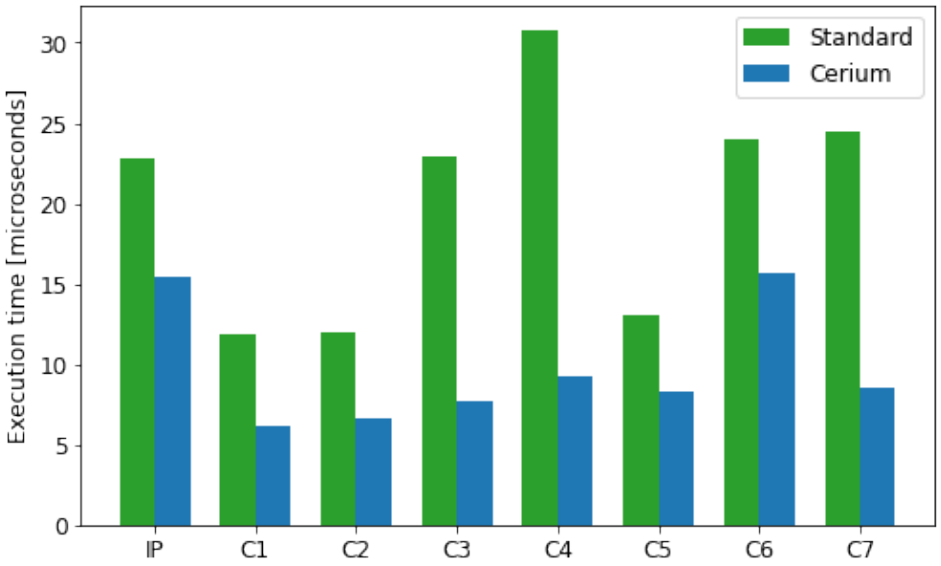


Figure 4.2: Performance comparison based on different types of changes.

4.3 Framework Limitations

The measurements in the previous section only take into account the time it takes for the type checking analysis to be executed. However, as figure 4.3 exemplifies, once we consider the complete *Cerium* framework pipeline we can observe that in practice type checking only takes a fraction of the overall execution time.

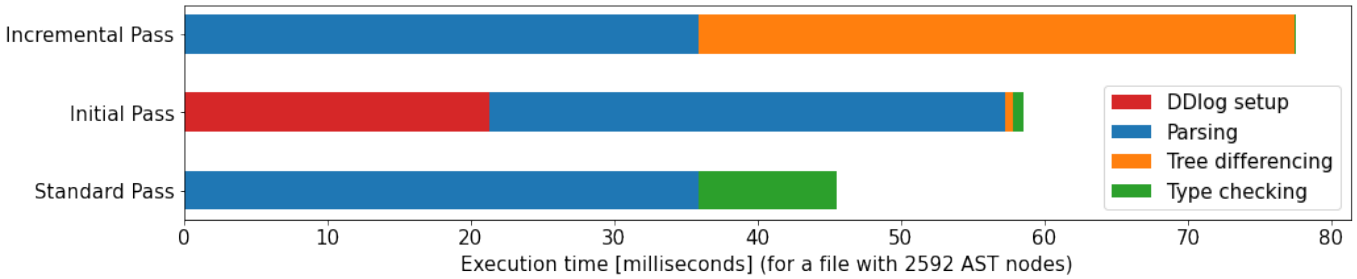


Figure 4.3: Comparison of execution time for different pipeline stages.

This also means that used as standalone tool, *Cerium* is slower than the standard type checker in most cases. While the overhead from the DDlog setup and initial type checking pass is negligible as it only occurs once at the start of a programming session, and parsing (including the conversion to the internal AST) has to be performed for both type checkers, we can primarily identify **tree differencing as the limiting factor for performance**.

This is true both in the sense that computing the delta takes up a lot of execution time that is not required in a standard type checking pass, as well as in the sense that the DDlog execution time is dependant on the quality (i.e. exactness) of the delta. The issue is further complicated by the trade-off between exactness and complexity that makes it hard to optimise for both.

To some extent this is a reflection on my algorithm as better, more optimised differencing algorithms do exist. However as any differencing algorithm will involve some form of tree traversal which will take more time the larger the input programs it compares are, this will always be an inherent bottleneck to any incremental computation.

I found that this limitation is often not made clear in a lot of literature around incremental computation: papers such as one of the main papers this project is based on (see [3]) tend to assume a program delta is given and neglect to mention that any performance results presented are only useful in practice if a suitable delta can be obtained.

Though tree differencing presents a limitation this does not mean incremental type checking is not useful in general: integrated into an IDE there are many analyses which can be incrementalised, as well as other uses for program deltas (such as providing a more accurate history of file modifications that is based on ASTs and not lines of code which often don't capture any dependencies), so that overall the time needed for tree differencing becomes worthwhile.

4.4 Review of Success Criteria

All the core requirements set out at the beginning of the project have been met:

- ✓ A a type system for an imperative language has been specified.
- ✓ A pipeline transforming source code to DDlog relations has been implemented.
- ✓ A tree differencing algorithm finding an approximate program delta has also been implemented as part of the pipeline.
- ✓ A type checking analysis has been written in DDlog.
- ✓ A standard type checker has been implemented as a baseline and benchmarks have been performed on manually created example programs and changes.

Consequently, the success criteria defined in the project proposal can be seen as satisfied:

1. An incremental type checker, which determines whether a program is correctly typed or not, is implemented.

In the proposal this was also followed up by "'Correctly' in this case is defined to be an empirical measure: the result of the type checker should overlap with the program being rejected or accepted by a standard C compiler.". This part of the criterion wasn't possible to achieve as during the course of the project I chose to specify a type system which differs to the standard C type system. However as the type checker is generally working and all *Ceres* features have been used in at least one example program with output being as expected, I would still consider this criterion met. Nevertheless, ideally the type checker would be tested further by more extensive unit tests covering all cases to guarantee correctness.

2. The incremental type checker should, on average, perform faster than the non-incremental type checker when tested on the corpus of example code changes.

As evidenced in this chapter, if you exclusively consider the time it takes to execute the type checking analysis (after already having performed an initial pass of the whole program), the *Cerium* outperforms a non-incremental type checker in the majority of the benchmarks performed. This criterion is therefore also met.

4.5 Summary

- Benchmarks were performed using a recursive descent type checker as a baseline, example programs and changes which partially based on open source C code as an input dataset, an *criterion* as a library for measuring execution time.
- The benchmarks show that the *Cerium* type checking analysis **satisfies the definition of an incremental type checker**: its performance is proportional to program changes, outperforming the baseline type checker.
- Evaluating the pipeline as whole, we can identify **tree differencing as a bottleneck**.
- Overall the project **satisfies all core requirements and success criteria**.

Chapter 5

Conclusions

Overall the project was a success: I met the success criteria set up in the project proposal by implementing *Cerium*, an incremental type checker targeting *Ceres* (a syntactic subset of C).

In order to achieve this I formally specified a target type system, implemented a framework in Rust which is able to convert a given *Ceres* input program into Datalog relations and find the difference between two programs, and wrote a type checking analysis in DDlog (an incremental Datalog dialect).

I demonstrated *Cerium* satisfies the definition of an incremental type checker by benchmarking it against a standard type checker and showing that its performance is proportional to program changes and outperforms the standard type checker.

5.1 Lessons Learnt

The main oversight made during planning before the start of the project was **not identifying the complexity of implementing a tree differencing algorithm early enough** and therefore not including sufficient time for it in the project plan. This was mainly due to me being unfamiliar with the wider subject area and a lot of the literature related to the approach I used for incremental type checking being quite recent and disjointed.

Together with external circumstances affecting my project progress, this meant I spent a lot less time exploring more complex aspects of type checking in Datalog than I would have liked too. Nonetheless it was a useful lesson in **adapting to changing circumstances** and plans.

In hindsight I think I also could have spend less on the tree differencing algorithm by adapting a more **test-driven development** upfront as I spent a lot of time debugging the algorithm only to find edge cases I had not considered before. More test would have also allowed me to provide better evidence for the correctness of the type checker.

5.2 Future Work

The project opens up many possible opportunities for future work:

- **Extending *Ceres* and adding type checker features**: The next step in my implementation would have been adding error collection to the type checker in order to provide more useful error messages. Furthermore, there are many C type features not included in *Ceres* which might provide insights into new techniques for translating type systems to Datalog.
- **Integrating *Cerium* with an IDE**: A type checker is most useful to a programmer if any errors are visualised directly in the code.
- **Improving tree differencing**: Beyond considering other tree differencing algorithms, there is also the possibility of changing the approach to differencing completely:
  - **Combining incremental parsing and type checking**: An interesting idea I briefly considered but had to quickly abandon was modifying an incremental parser such as [11] to identify the program delta during parsing, eliminating the need for additional differencing overhead and moving closer to a fully incremental system.
  - **Exploring new ways of code identification**: Going one step further, languages such as *unison*, which stores functions as hashes of their syntax trees, are showcasing new way of thinking of code structure. This makes it easier to identify code that has changed or remains semantically the same with only superficial changes such as renaming, which in turn makes it easier to find program deltas.
- **Explore human-computer interaction aspects of IDEs**: Collecting concrete data about the types of changes programmers make to code and how a program is built up from scratch as a series of changes can improve any assumptions made when designing algorithms during implementation and program changes during evaluation.
- **Improve benchmarking dataset through automatic change creation**: In order to be able to evaluate incremental type checkers on a larger scale, it would be helpful to be able to create code and code changes automatically, possibly by training a machine learning model on an existing manual dataset.

Bibliography

[1] Yannis Smaragdakis and Martin Bravenboer. "Using Datalog for Fast and Easy Program Analysis". In: *Proceedings of the First International Conference on Datalog Reloaded*. Springer-Verlag, 2010, pp. 245–251. DOI: 10.1007/978-3-642-24206-9_14.

[2] Bernhard Scholz et al. "On Fast Large-Scale Program Analysis in Datalog". In: *Proceedings of the 25th International Conference on Compiler Construction*. Association for Computing Machinery, 2016, pp. 196–206. DOI: 10.1145/2892208.2892226.

[3] André Pacak, Sebastian Erdweg and Tamás Szabó. "A Systematic Approach to Deriving Incremental Type Checkers". In: 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428195.

[4] *The State of Developer Ecosystem 2021*. URL: https://www.jetbrains.com/lp/devecosystem-2021/ (visited on 07/05/2022).

[5] Sebastian Erdweg et al. "A Co-Contextual Formulation of Type Rules and Its Application to Incremental Type Checking". In: *SIGPLAN Not.* 50.10 (Oct. 2015), pp. 880–897. ISSN: 0362-1340. DOI: 10.1145/2858965.2814277.

[6] Guozhu Dong, Jianwen Su and Rodney Topor. "Nonrecursive Incremental Evaluation of Datalog Queries". In: *Annals of Mathematics and Artificial Intelligence* 14 (Oct. 1995). DOI: 10.1007/BF01530820.

[7] Leonid Ryzhyk and Mihai Budiu. "Differential Datalog". In: (2019).

[8] *Differential Dataflow (GitHub repository)*. URL: https://github.com/frankmcsherry/differential-dataflow (visited on 04/05/2022).

[9] *lang-c (GitHub repository)*. URL: https://github.com/vickenty/lang-c (visited on 04/05/2022).

[10] *clang-rs (GitHub repository)*. URL: https://github.com/KyleMayes/clang-rs (visited on 04/05/2022).

[11] *Tree-sitter (GitHub repository)*. URL: https://github.com/tree-sitter/tree-sitter (visited on 04/05/2022).

[12] *ISO/IEC 9899:2018*. URL: https://www.iso.org/standard/74528.html (visited on 04/05/2022).

[13] Sandrine Blazy and Xavier Leroy. "Mechanized Semantics for the Clight Subset of the C Language". In: *J. Autom. Reason.* 43.3 (2009), pp. 263–288. DOI: 10.1007/s10817-009-9148-3.

[14] Chucky Ellison and Grigore Rosu. "An executable formal semantics of C with applications". In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. Ed. by John Field and Michael Hicks. ACM, 2012, pp. 533–544. DOI: 10.1145/2103656.2103719.

[15] Nikolaos Papaspyrou. "A Formal Semantics for the C Programming Language". In: (Apr. 1998).

[16] Todd J. Green et al. *Datalog and Recursive Query Processing*. 2013.

[17] Serge Abiteboul, Richard Hull and Victor Vianu. *Foundations of Databases*.

[18] Philip Bille. "A survey on tree edit distance and related problems". In: *Theoretical Computer Science* 337.1 (2005), pp. 217–239. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2004.12.030.

[19] Jean-Rémy Falleri et al. "Fine-Grained and Accurate Source Code Differencing". In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, 2014, pp. 313–324. DOI: 10.1145/2642937.2642982.

[20] Victor Cacciari Miraldo and Wouter Swierstra. "An Efficient Algorithm for Type-Safe Structural Diffing". In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341717.

[21] Sebastian Erdweg, Tamás Szabó and André Pacak. "Concise, Type-Safe, and Efficient Structural Diffing". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Association for Computing Machinery, 2021, pp. 406–419. DOI: 10.1145/3453483.3454052.

Appendix A

Input Program Relations

```
FunDef(id: ID, fun_name: string, return_type_id: ID, arg_ids: Vec<ID>, body_id: ID)
Assign(id: ID, var_name: string, type_id: ID, expr_id: ID)
IfElse(id: ID, cond_id: ID, then_id: ID, else_id: ID)
FunCall(id: ID, fun_name: string, arg_ids: Vec<ID>)
Item(id: ID, stmt_id: ID, next_stmt_id: ID)
BinaryOp(id: ID, arg1_id: ID, arg2_id: ID)
Arg(id: ID, var_name: string, type_id: ID)
While(id: ID, cond_id: ID, body_id: ID)
If(id: ID, cond_id: ID, then_id: ID)
TransUnit(id: ID, body_ids: Vec<ID>)
Compound(id: ID, start_id: ID)
Var(id: ID, var_name: string)
EndItem(id: ID, stmt_id: ID)
Return(id: ID, expr_id: ID)
Void(id: ID)
Int(id: ID)
Float(id: ID)
Char(id: ID)
```

Appendix B

Output Typing Relations

```
FindFunBinding(current_id: ID, fun_name: string, return_type: Type,
               arg_type_ids: Vec<ID>)
FindArgVarBinding(current_id: ID, var_name: string, t: Type)
TypedReturnCompound(id: ID, fun_return_type_id: Type)
TypedReturnItem(id: ID, return_type: Type)
TypedIfElseStatement(id: ID, t:Type)
TypedIfStatement(id: ID, t: Type)
ArithmeticType(id: ID, t: Type)
TypedArgument(id: ID, t: Type)
TypedLiteral(id: ID, t: Type)
TypedReturn(id: ID, t: Type)
TypesMatch(id1: ID, id2: ID)
TypedExpr(id: ID, t: Type)
BodyErrorTransUnit(id: ID)
TypedDeclaration(id: ID)
TypedTransUnit(id: ID)
TypedStatement(id: ID)
TypedCompound(id: ID)
TypedFunDef(id: ID)
TypedItem(id: ID)
```

Appendix C

Project Proposal

Implementing an Incremental Type Checker - Project Proposal

October 2021

1 Introduction and Project Description

Statically type checking a program without executing it by analysing the source code is an essential tool for ensuring type safety during development. Particularly in an integrated development environment (IDE), programmers often want immediate feedback on the correctness of their typing after every one of their changes. However, re-checking a potentially very large code base from scratch for every small change can be very expensive in terms of time complexity. Ideally the type checking should therefore happen incrementally, meaning that only computation that is relevant to the code change is performed.

There exist multiple possible approaches that can be taken to incrementalise static analyis and type checking in particular, ranging from using caching and memoization to typing algorithms specifically designed for incremental systems (see [1] and [2] for example). One possible method is outlined in [3]: The idea is to transform typing rules in a way that allows them be expressed in Datalog, a declarative logic programming language for which there are dialects that can be compiled incrementally and which has recently been used extensively for static analysis.

The aim of this project is to use the above approach (i.e. using an incremental evaluator) to implement an incremental type checker for a C-like imperative programming language. While [3] mainly focuses on functional languages, there have been other projects utilising the same or a similar approach for a variety of other language types, such as for example [4] (which describes the implementation of an incremental type checker for Rust) and [5] (which explores it in the context of object-oriented features), showing that this is a viable project.

2 Starting Point

I have no previous experience in implementing type checkers, however I have taken multiple courses that could be relevant to this project: Semantics of Programming Languages, Compiler Construction, Prolog (since Datalog is a subset of it), and Programming in C. I also plan on using Rust for part of the implementation, which I haven't used before.

3 Project Structure

3.1 Core

The project can be split into the following components:

- **Preparation**: The first step is to determine a set of typing rules based on CLight [6], a subset of C with existing formalized semantics. I will simplify Clight even further to begin with in order to have a basic imperative language to get started, then only later is the aim is to investigate to which extent more features can be added. Furthermore some of the typing rules will then have to be transformed in a way such that they can be implemented with the incremental evaluator.
- **Type Checker**: The plan is for the type checker itself to be written in a language that is incrementally compiled. I currently plan on using DDlog [7]. In the core implementation the checker will only have to return true or false depending on whether a program is typed correctly or not (as opposed to collecting every error).
- **Framework**: In order to be able to run the type checker and make it a usable tool, I will also need to create several supporting components. Since DDlog is written in Rust and Rust is one of the recommended languages to use to integrate DDlog as a library, the intention is for the framework to be primarily written in Rust.
  1. Generate a **parser** for parsing input source code into a suitable syntax tree representation. There are multiple parser generators available in Rust. It might also be interesting to potentially use an incremental parser such as [8] in conjunction with the incremental type checker.
  2. Implement an **encoder** as the interface between the parser and the checker which:
     a) Computes the difference between the syntax tree before and after a change in the source code.
     b) Encodes it in a suitable representation such that the update can be passed to the type checker.
  3. Package all components together as one **pipeline** that can be run with a script and gives suitable feedback.

3.2 Evaluation

In order to evaluate the above, I need to:

- Create a handwritten **non-incremental type checker**.
- Create a corpus of **example programs** with corresponding code changes. For this I will find suitable C source code published open source in GitHub repositories as a basis for the programs, and then manually create possible code changes. In particular the source code for the CompCert compiler [9] (which is a formally verified compiler for CLight) includes a collection of programs for testing purposes.

The incremental version can then be benchmarked against the non-incremental one using the corpus of programs.

3.3 Possible Extensions

1. **Error collection**: To really be useful, a type checker shouldn't just stop when it identifies an incorrect derivation, and instead continue checking to find and display all typing errors in the program.

2. **Further language features**: Investigate whether any features beyond the basic set of rules implemented in the core project can be added to the type checker, exploring whether there are any limitations to this approach to type checking.

3. **IDE integration**: Instead of using the type checker as a standalone tool, they are usually used as part of on IDE, enabling the visual highlighting of errors. This can be achieved by leveraging the API of text editors such as VSCode, for example.

4 Success Criteria

1. An incremental type checker, which determines whether a program is correctly typed or not, is implemented. 'Correctly' in this case is defined to be an empirical measure: the result of the type checker should overlap with the program being rejected or accepted by a standard C compiler.

2. The incremental type checker should, on average, perform faster than the non-incremental type checker when tested on the corpus of example code changes.

5 Timetable and Milestones

Michaelmas Term

14/10/21 - 27/10/21 (2 weeks)

- *Milestone:* Submit proposal.
- Decide on typing rules for target programming language.
- Read up on and understand relevant background papers and materials.

28/10/21 - 17/11/21 (3 weeks)

- Determine and perform transformations necessary for implementing typing rules. → *Milestone:* Finish preparation part of project.
- Determine and familiarise myself with libraries and languages needed for the framework (parser generator, Rust, DDlog) by using them for a very simple toy language type checker.

18/11/21 - 01/12/21 (2 weeks)

- Write some example programs for testing the type checker during development.
- Start working on implementing the core type checker and supporting framework.

Christmas Vacation:

02/12/21 - 19/01/22 (7 weeks)

- *Christmas break + buffer time.*
- Continue working on implementing the core type checker and supporting framework. → *Milestone*: Finish the core implementation for the type checker and framework.

Lent Term:

20/01/22 - 02/02/22 (2 weeks)

- Write progress report. → *Milestone*: Submit progress report.
- Implement non-incremental type-checker and create corpus for benchmarking.

03/02/22 - 16/02/22 (2 weeks)

- Focus on evaluating the project.→ *Milestone*: Finish evaluation phase of core project.
- Start writing the dissertation.

17/02/22 - 16/03/21 (4 weeks)

- Continue writing the dissertation.
- Work on extensions if time permits.

Easter Vacation:

17/03/22 - 27/04/22 (6 weeks)

- *Easter break.*
- Finish writing dissertation.
- Work on extensions if time permits.

Easter Term:

28/04/21 - 18/05/22 (3 weeks)

- *Buffer time in case of unexpected problems.*
- *Milestone:* Submit dissertation and code.

6 Resource Declaration

I will use my personal computer for this project (MacBook Pro, Intel Core i5, 1.4 GHz, 16GB RAM). I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. I will do this by using Git version control with additional Google Drive backups throughout the project, with MCS computers as a fallback option in case of laptop failure.

References

[1] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. "Mechanical incrementalization of typing algorithms". In: *Science of Computer Programming* 208 (2021), p. 102657. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102657. URL: https://www.sciencedirect.com/science/article/pii/S0167642321000502.

[2] Sebastian Erdweg et al. "A Co-Contextual Formulation of Type Rules and Its Application to Incremental Type Checking". In: *SIGPLAN Not.* 50.10 (Oct. 2015), pp. 880–897. ISSN: 0362-1340. DOI: 10.1145/2858965.2814277. URL: https://doi.org/10.1145/2858965.2814277.

[3] André Pacak, Sebastian Erdweg, and Tamás Szabó. "A Systematic Approach to Deriving Incremental Type Checkers". In: 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428195. URL: https://doi.org/10.1145/3428195.

[4] *Incremental Type Checking in IncA*. URL: http://resolver.tudelft.nl/uuid:95fd2998-1b39-4dd1-aba4-b2f73b443828r.

[5] Tamás Szabó et al. "Incremental Overload Resolution in Object-Oriented Programming Languages". In: ISSTA '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 27–33. ISBN: 9781450359399. DOI: 10.1145/3236454.3236485. URL: https://doi.org/10.1145/3236454.3236485.

[6] Sandrine Blazy and Xavier Leroy. "Mechanized Semantics for the Clight Subset of the C Language". In: *Journal of Automated Reasoning* 43.3 (July 2009), pp. 263–288. ISSN: 1573-0670. DOI: 10.1007/s10817-009-9148-3. URL: http://dx.doi.org/10.1007/s10817-009-9148-3.

[7] *Differential Datalog*. URL: hhttps://github.com/vmware/differential-datalog.

[8] *Tree-sitter*. URL: https://github.com/tree-sitter/tree-sitter.

[9] *CompCert: The formally-verified C compiler*. URL: https://github.com/AbsInt/CompCert.